Guide du développeur

Borland®
Delphi™ 6
pour Windows

Reportez-vous au document DEPLOY situé dans le répertoire racine de votre produit Delphi 6 pour obtenir la liste complète des fichiers que vous pouvez distribuer en accord avec les termes du contrat de licence.

Les applications mentionnées dans ce manuel sont brevetées ou en attente de brevet. Ce document ne donne aucun droit sur ces brevets.

Imprimé en Irlande

HDE1350WW21001 1E0R0501
0102030405-9 8 7 6 5 4 3 2 1
D3

Table des matières

Chapitre 6
Conception de l'interface utilisateur des applications 6-1

Chapitre 7
Manipulation des contrôles 7-1

Chapitre 20

Utilisation du moteur de bases de données Borland 20-1

Chapitre

1

Introduction

Ce manuel aborde des notions de développement intermédiaires et avancées. Il traite notamment de la création d'applications de bases de données client/serveur, de l'écriture de composants personnalisés et de la création d'applications serveurs Web Internet. Il vous permet de construire des applications qui respectent les spécifications de nombreux standards comme SOAP, TCP/IP, COM+ et ActiveX. Nombre de fonctionnalités avancées concernant le développement web, les technologies XML de pointe et le développement de bases de données nécessitent des composants ou des experts qui ne sont pas disponibles dans toutes les versions de Delphi.

Ce guide suppose que l'utilisation et les techniques fondamentales de programmation Delphi ont été assimilées. Pour une présentation de la programmation Delphi et de l'environnement de développement intégré (EDI), voir le manuel de *Prise en main* et l'aide en ligne.

Contenu de ce manuel

Ce manuel comporte cinq parties décomposées comme suit :

- **La partie I, "Programmation Delphi",** décrit la manière de concevoir des applications Delphi généralistes. Cette partie donne des détails sur les techniques de programmation utilisables dans toute application Delphi. Elle décrit, par exemple, la manière d'utiliser les objets courants de la bibliothèque de composants visuels (VCL) ou de la bibliothèque de composants Borland multiplate-forme (CLX) qui simplifient le développement de l'interface utilisateur. Ces objets permettent de gérer des chaînes, de manipuler du texte, d'implémenter des dialogues communs, etc. Cette section contient également des chapitres décrivant la manipulation des graphiques et la gestion des erreurs et des exceptions, l'utilisation des DLL, l'automation OLE et l'écriture d'applications internationales.

Un chapitre décrit comment utiliser les objets de la bibliothèque de composants Borland multiplate-forme (CLX) pour développer des applications pouvant être compilées et exécutées sous Windows ou sous Linux.

Le chapitre sur le déploiement aborde les opérations nécessaires pour distribuer votre application auprès de ses utilisateurs. Ce chapitre donne des informations sur les options de compilation, l'utilisation de InstallShield Express, les problèmes de droits de distribution et sur la manière de déterminer les paquets, DLL et autres bibliothèques qu'il faut utiliser pour générer la version distribuée d'une application.

- **La partie II, "Développement d'applications de bases de données",** décrit comment construire des applications de bases de données en utilisant les outils et les composants base de données. Delphi vous permet d'accéder à de nombreux types de bases de données, notamment des bases de données locales, comme Paradox et dBASE, et des bases de données serveur SQL en réseau, comme InterBase, Oracle et Sybase. Vous pouvez choisir parmi divers mécanismes d'accès aux données, dont dbExpress, BDE (moteur de bases de données Borland), InterbaseExpress et ADO. Pour implémenter les applications de bases de données les plus évoluées, vous avez besoin de fonctionnalités qui ne sont pas disponibles dans toutes les versions de Delphi.
- **La partie III, "Ecriture d'applications Internet",** décrit comment créer des applications distribuées sur internet. Delphi comprend une large gamme d'outils permettant d'écrire des applications serveur web, y compris l'architecture Web Broker pour créer des applications serveur multiplates-formes, WebSnap pour concevoir des pages web dans un environnement GUI, le support de l'utilisation de documents XML et une architecture pour utiliser les services web basés sur SOAP. Pour un support de bas niveau de la messagerie dans les applications Internet, cette section décrit également la façon de travailler avec des composants socket. Les composants qui implémentent un grand nombre de ces fonctionnalités ne sont pas disponibles dans toutes les versions de Delphi.
- **La partie IV, "Développement d'applications COM",** décrit comment construire des applications qui peuvent interagir avec d'autres objets API basés sur COM du système, comme les extensionsdu Shell Windows ou les applications multimédia. Delphi contient des composants qui supportent ActiveX, COM+ et une bibliothèque basée sur COM pour les contrôles COM qui peuvent être utilisés par les applications généralistes ou web. Le support des contrôles COM n'est pas disponible dans toutes les éditions de Delphi. Pour créer des contrôles ActiveX, vous devez disposer de l'édition Professionnelle ou Entreprise.
- **La partie V, "Création de composants personnalisés",** décrit la manière de concevoir et d'implémenter vos propres composants et de les intégrer à la palette des composants de l'EDI. Un composant peut quasiment constituer tout élément de programme manipulable à la conception. L'implémentation de composants personnalisés nécessite la dérivation d'une nouvelle classe à partir d'une classe existante de la bibliothèque de classes VCL ou CLX.

Conventions typographiques

Ce manuel utilise les polices et les symboles décrits dans le tableau suivant pour mettre en évidence des parties particulières du texte :

Tableau 1.1 Polices et symboles utilisés dans ce manuel

Police ou symbole	Signification
`Police à pas fixe`	Le texte apparaissant avec une police à pas fixe sert à représenter le texte tel qu'il apparaît à l'écran ou dans du code Pascal Objet. Il indique aussi les valeurs à saisir au clavier.
[ ]	Les crochets dans le texte ou dans une syntaxe représentent des éléments facultatifs. Ces crochets sont à omettre lors de la saisie.
Gras	Les mots en gras dans le texte ou dans le code servent à représenter les mots réservés du langage Pascal Objet et les options du compilateur.
Italique	Les mots en caractères italiques représentent des identificateurs du langage Pascal Objet comme les variables et les noms de types. L'italique sert aussi à faire ressortir certains mots comme les nouveaux termes.
Touches	Cette police sert à indiquer une touche du clavier. Par exemple, "Appuyez sur *Echap* pour quitter un menu".

Support technique

Inprise offre également de nombreuses options de support pour satisfaire les besoins de l'ensemble de ses développeurs. Pour obtenir des informations sur les services de support, consultez http://www.borland.fr.

D'autres documents d'informations techniques Delphi et les réponses aux questions fréquentes (FAQ) se trouvent également sur ce site.

A partir de ce site Web, vous pouvez accéder à de nombreux groupes de discussion où les développeurs Delphi s'échangent des informations, des astuces et des techniques. Ce site propose également une liste de livres concernant Delphi.

Partie

I

Programmation Delphi

Les chapitres de cette partie présentent les concepts et connaissances nécessaires pour créer des applications Delphi avec n'importe quelle édition du produit. Ils introduisent également des concepts qui seront traités dans d'autres sections de ce manuel.

Chapitre

2

Développement d'applications avec Delphi

Borland Delphi est un environnement de programmation visuelle orienté objet permettant le développement rapide d'applications 32 bits en vue de leur déploiement sous Windows et sous Linux. En utilisant Delphi, vous pouvez créer de puissantes applications avec un minimum de programmation.

Delphi propose une bibliothèque de classes complète appelée la VCL (*Visual Component Library*), la bibliothèque des composants Borland multiplates-formes appelée CLX et une suite d'outils de conception rapide d'applications (RAD) comprenant des modèles d'application, des modèles de fiche et des experts de programmation. Delphi gère réellement la programmation orientée objet :

- la bibliothèque de classes VCL comprend des objets qui encapsulent l'API Windows ainsi que d'autres techniques de programmation utiles (Windows)
- la bibliothèque de classes CLX comprend des objets qui encapsulent la bibliothèque Qt (Windows ou Linux)

Ce chapitre décrit brièvement l'environnement de développement Delphi et comment il s'inscrit dans le cycle de développement. Le reste de ce manuel donne des détails techniques sur le développement d'applications, la gestion des bases de données et les applications Internet ou intranet, ainsi que des informations sur la création de contrôles ActiveX ou COM, et sur l'écriture de composants personnalisés.

Environnement de développement intégré

Au démarrage de Delphi, vous êtes immédiatement placé dans l'environnement de développement intégré, appelé également EDI. Cet environnement propose tous les outils nécessaires à la conception, au test, au débogage et au déploiement d'applications.

L'environnement de développement Delphi contient un concepteur visuel de fiches, l'inspecteur d'objets, l'arborescence des objets, la palette des composants, le gestionnaire de projet, l'éditeur de code source et le débogueur, entre autres outils. Certains d'entre eux ne font pas partie de toutes les versions du produit. Vous pouvez, à votre guise, passer de la représentation visuelle d'un objet (dans le concepteur de fiche ; l'inspecteur d'objets permettant de modifier l'état initial de l'objet lors de l'exécution) à l'éditeur de code source qui permet de changer la logique d'exécution de l'objet. La modification de propriétés liées au code dans l'inspecteur d'objets, comme le nom d'un gestionnaire d'événement, modifie automatiquement le code source correspondant. De plus la modification du code source, par exemple le nom d'une méthode gestionnaire d'événement dans la déclaration de classe d'une fiche, est immédiatement reflétée dans l'inspecteur d'objets.

L'EDI supporte le développement d'applications à tous les stades du cycle de vie du produit, de la conception au déploiement. L'utilisation des outils de l'EDI accélère le prototypage et réduit la durée du développement.

Une présentation plus complète de l'environnement de développement est proposée dans le manuel *Prise en main*, livré avec le produit. En outre, le système d'aide en ligne offre de l'aide sur tous les menus, dialogues et fenêtres.

Conception d'applications

Delphi dispose de tous les outils nécessaires pour commencer à concevoir une application :

- Une fenêtre vide, appelée une *fiche*, dans laquelle concevoir l'interface utilisateur, de l'application.
- Des bibliothèques de classes contenant de nombreux objets réutilisables.
- L'inspecteur d'objets pour connaître ou modifier les caractéristiques des objets.
- L'éditeur de code qui permet d'accéder directement à la logique sous-jacente du programme.
- Le gestionnaire de projet qui permet de gérer les fichiers constituant un ou plusieurs projets.
- De nombreux outils, comme un éditeur d'images accessible dans la barre d'outils et un débogueur intégré accessible par menus, qui permettent de gérer le développement de l'application directement dans l'EDI.
- Des outils en ligne de commande, y compris des compilateurs, des éditeurs de liens.

Vous pouvez utiliser Delphi pour concevoir tout type d'application 32 bits, que ce soit un utilitaire de portée générale ou un programme complexe de gestion de données ou des applications distribuées. Les outils de base de données de Delphi et ses composants orientés données permettent de développer rapidement des applications de bases de données de bureau ou client/serveur. En utilisant les contrôles orientés données de Delphi, vous pouvez visualiser des données réelles

alors même que vous concevez votre application et voir immédiatement le résultat d'une requête de base de données ou d'une modification de l'interface de l'application.

Le chapitre 5, "Création d'applications, de composants et de bibliothèques", décrit comment Delphi gère les différents types d'applications.

De nombreux objets fournis dans la bibliothèque des classes sont accessibles dans la palette des composants de l'EDI. La palette des composants montre tous les contrôles, visuels ou non, que vous pouvez placer sur une fiche. Chaque onglet contient des composants regroupés par fonctionnalité. Par convention, les noms des objets de la bibliothèque des classes commencent par un T, comme *TStatusBar*.

Ce qui est révolutionnaire dans Delphi est que vous pouvez créer vos propres composants en utilisant le Pascal Objet. La plupart des composants fournis sont écrits en Pascal Objet. Vous pouvez ajouter à la palette les composants que vous avez écrits et la personnaliser à votre convenance en insérant de nouveaux onglets.

Vous pouvez également recourir à Delphi pour le développement multiplate-forme, sous Linux et Windows, en utilisant CLX. CLX contient un ensemble de classes qui, lorsque vous les utilisez à la place de la VCL, permettent à votre programme de passer indifféremment de Windows à Linux.

Développement d'applications

Alors même que vous concevez visuellement l'interface utilisateur d'une application, Delphi génère le code Object Pascal correspondant pour gérer l'application. Dès que vous sélectionnez et modifiez les propriétés des composants et des fiches, le résultat de ces modifications apparaît automatiquement dans le code source, et vice-versa. Vous pouvez modifier directement les fichiers source avec tout éditeur de texte, y compris l'éditeur de code intégré. Les modifications effectuées dans le code sont immédiatement reflétées dans l'environnement visuel.

Création des projets

Tout le développement d'applications avec Delphi s'effectue par le biais des projets. Quand vous créez une application dans Delphi vous créez un projet. Un projet est une collection de fichiers qui constituent une application. Certains de ces fichiers sont créés au cours de la conception. D'autres sont générés automatiquement lorsque vous compilez le code source du projet.

Vous pouvez voir le contenu d'un projet à l'aide d'un outil de gestion de projet nommé le Gestionnaire de projet. Le gestionnaire de projet présente la liste, sous forme d'une vue hiérarchisée, des noms d'unités et des fiches contenues éventuellement dans chaque unité, ainsi que les chemins d'accès aux fichiers du

projet. Vous pouvez modifier directement la plupart de ces fichiers, mais il est plus simple et plus sûr d'utiliser les outils visuels de Delphi.

En haut de la hiérarchie, se trouve un fichier groupe. Vous pouvez combiner plusieurs projets dans un groupe de projets. Cela vous permet d'ouvrir plusieurs projets à la fois dans le gestionnaire de projet. Les groupes de projets permettent de rassembler des projets liés et de travailler sur eux, par exemple des applications qui fonctionnent ensemble ou font partie d'une application multi-niveaux. Si vous ne travaillez que sur un seul projet, vous n'avez pas besoin de fichier groupe de projets pour créer une application.

Les fichiers projet, qui décrivent des projets individuels, des fichiers et des options associées, portent l'extension .dpr. Les fichiers projet contiennent des directives pour la construction d'une application ou d'un objet partagé. Quand vous ajoutez et supprimez des fichiers en utilisant le gestionnaire de projet, le fichier projet est mis à jour. Vous spécifiez les options du projet dans le dialogue Options de projet, qui contient des onglets pour les divers aspects de votre projet, comme les fiches, l'application, le compilateur. Ces options de projet sont stockées avec le projet dans le fichier projet.

Les unités et les fiches sont les blocs de base de la construction d'une application Delphi. Un projet peut partager n'importe quel fichier fiche et unité existant, y compris ceux qui se trouvent hors de l'arborescence des répertoires du projet. Cela inclut des procédures et des fonctions personnalisées, écrites sous forme de routines indépendantes.

Si vous ajoutez à un projet un fichier partagé, celui-ci n'est pas copié dans le répertoire du projet en cours ; il reste à sa place initiale. L'ajout du fichier partagé au projet en cours inscrit le nom et le chemin du fichier dans la clause **uses** du fichier projet. Delphi s'en charge automatiquement lorsque vous ajoutez des unités à un projet

Quand vous compilez un projet, l'emplacement des fichiers qui constituent le projet n' a aucune importance. Le compilateur traite les fichiers partagés comme ceux créés par le projet lui-même.

Edition du code

L'éditeur de code Delphi est un éditeur ASCII complet. Si vous utilisez l'environnement de programmation visuel, une fiche est automatiquement affichée dans un nouveau projet. Vous pouvez commencer la conception de l'interface de votre application en plaçant des objets sur la fiche et en modifiant leur fonctionnement dans l'inspecteur d'objets. Mais d'autres tâches de programmation, comme l'écriture des gestionnaires d'événements pour les objets, doivent se faire en tapant directement le code.

Le contenu d'une fiche et toutes ses propriétés ainsi que ses composants et leurs propriétés peuvent être modifiés sous forme de texte dans l'éditeur de code. Vous pouvez ajuster le code généré dans l'éditeur de code et ajouter d'autres composants en tapant du code dans l'éditeur. Au fur et à mesure que vous tapez du code dans l'éditeur, le compilateur l'analyse constamment afin de changer la

disposition de la fiche. Vous pouvez revenir à la fiche, voir et tester les changements apportés dans l'éditeur, puis continuer à modifier la fiche elle-même.

La génération de code Delphi et le système de flux des propriétés est entièrement ouvert à l'examen. Le code source de tout ce qui se trouve dans le fichier exécutable final (tous les objets VCL, les objets CLX, les sources RTL et tous les fichiers projet Delphi) peut être visualisé et modifié dans l'éditeur de code.

Compilation des applications

Quand vous avez fini de concevoir l'interface de votre application sur la fiche, après avoir écrit le code supplémentaire souhaité, vous pouvez compiler le projet depuis l'EDI ou depuis la ligne de commande.

Tous les projets ont comme cible un fichier exécutable distribuable unique. Vous pouvez voir ou tester votre application à divers stades du développement en la compilant, la construisant ou l'exécutant :

- Quand vous la compilez, seules les unités qui ont changé depuis la dernière compilation sont recompilées.
- Quand vous la construisez, toutes les unités du projet sont compilées, qu'elles aient ou non changé depuis la dernière compilation. Cette technique est utile quand vous n'êtes pas certain des fichiers qui ont été modifiés ou quand vous voulez simplement garantir que tous les fichiers en cours soient synchronisés. Il est également important de construire l'application quand vous avez changé les directives globales du compilateur, afin d'assurer que tout le code se compile de façon correcte. Vous pouvez tester ainsi la validité de votre code source, sans compiler le projet.
- Quand vous l'exécutez, vous compilez l'application, puis l'exécutez. Si vous avez modifié le code source depuis la dernière compilation, le compilateur recompile les modules qui ont été changés et lie à nouveau votre application.

Si vous avez regroupé ensemble plusieurs projets, vous pouvez compiler ou construire tous les projets du groupe en une seule fois. Choisissez Projet | Compiler tous les projets ou Projet | Construire tous les projets, le groupe de projets étant sélectionné dans le gestionnaire de projet.

Débogage des applications

Delphi dispose d'un débogueur intégré qui permet de localiser et de corriger les erreurs d'une application. Le débogueur intégré permet de contrôler l'exécution du programme, de surveiller la valeur de variables et d'éléments de structures de données ou de modifier la valeur de données lors de l'exécution.

Le débogueur intégré peut suivre à la fois les erreurs d'exécution et les erreurs de logique. En exécutant le programme jusqu'à un emplacement spécifique et en visualisant la valeur des variables, les fonctions de la pile des appels et les

sorties du programme, vous pouvez surveiller son comportement et trouver les endroits où il ne se comporte pas comme prévu. Le débogueur est décrit plus en détail dans l'aide en ligne.

Vous pouvez également utiliser la gestion des exceptions pour connaître, localiser et traiter les erreurs. Les exceptions dans Delphi sont des classes, comme les autres classes de Delphi, sauf que, par convention, leur nom commence par E au lieu de T.

Déploiement des applications

Delphi dispose d'outils facilitant le déploiement d'une application. Par exemple, InstallShield Express (non disponible dans toutes les versions) vous aide à créer un programme d'installation pour votre application qui contient tous les fichiers nécessaires à l'exécution de l'application distribuée. Pour davantage d'informations sur le déploiement, voir chapitre 13, "Déploiement des applications".

Remarque Les versions de Delphi n'ont pas toutes des capacités de déploiement.

Le logiciel TeamSource (non disponible dans toutes les versions) est également utilisable disponible pour suivre les mises à jour des applications.

Chapitre

3

Utilisation des bibliothèques de composants

Ce chapitre présente les bibliothèques de composants et décrit certains composants que vous pouvez utiliser au cours du développement de vos applications. Delphi comprend à la fois la bibliothèque de composants visuels (VCL) et la bibliothèque de composants Borland multiplate-forme (CLX). La VCL est réservée au développement Windows tandis que CLX permet le développement multiplate-forme Windows et Linux. Ces deux bibliothèques de classes sont différentes mais présentent de nombreuses similarités. Les objets, propriétés, méthodes et événements qui ne font pas partie de CLX sont marqués "VCL seulement".

Présentation des bibliothèques de composants

VCL et CLX sont des bibliothèques de classes constituées de nombreux objets, dont certains sont également des composants ou des contrôles, que vous utilisez pour développer des applications. Les deux bibliothèques se ressemble beaucoup et contiennent de nombreux objets identiques. Certains objets de la VCL implémentent des fonctionnalités disponibles uniquement sous Windows, comme les objets qui apparaissent dans les onglets ADO, BDE, QReport, COM+, Services Web et Serveurs de la palette de composants. Pratiquement tous les objets CLX sont disponibles à la fois pour Windows et pour Linux.

Les objets VCL et CLX sont des entités actives qui contiennent toutes les données nécessaires ainsi que les "méthodes" (code) qui modifient ces données. Les données sont stockées dans les champs et les propriétés des objets et le code est constitué de méthodes qui agissent sur les valeurs de ces champs et propriétés. Chaque objet est déclaré en tant que "classe". Tous les objets VCL et CLX descendent de l'objet ancêtre *TObject* y compris ceux que vous développez en Pascal Objet.

Les composants forment un sous-ensemble des objets. Les composants sont des objets que vous pouvez placer sur une fiche ou un module de données et manipuler pendant la conception. Les composants apparaissent sur la palette des composants. Vous pouvez spécifier leurs propriétés sans écrire de code. Tous les composants VCL ou CLX descendent de l'objet *TComponent*.

Les composants sont des objets au véritable sens du terme programmation orientée objets (POO) car

- ils encapsulent des ensembles de données et des fonctions d'accès aux données,
- ils héritent des données et du comportement des objets dont ils sont dérivés,
- ils opèrent de façon interchangeable avec d'autres objets dérivés d'un ancêtre commun, par le biais d'un concept appelé *polymorphisme.*

Contrairement à la plupart des composants, les objets ne figurent pas sur la palette des composants. A la place, une variable d'instance par défaut est déclarée dans l'unité de l'objet, sinon, vous devez en déclarer une vous-même.

Les contrôles constituent un type spécial de composants, visibles aux utilisateurs à l'exécution. C'est un sous-ensemble des composants. Les contrôles sont des composants visuels que vous pouvez voir lorsque votre application est exécutée. Les contrôles ont des propriétés en commun qui spécifient leurs attributs visuels, comme *Height* et *Width.* Les propriétés, méthodes et événements que partagent les contrôles sont tous dérivés de *TControl*.

Reportez-vous au chapitre 10, "Utilisation de CLX pour le développement multiplate-forme", pour avoir des détails sur la programmation multiplate-forme et connaître les différences entre les environnements Windows et Linux. Vous pouvez accéder à des informations détaillées sur tous les objets VCL ou CLX, en utilisant l'aide en ligne pendant que vous programmez. Dans l'éditeur de code, placez le curseur en un endroit quelconque de l'objet et appuyez sur F1 pour afficher de l'aide sur les composants VCL ou CLX.

Si vous utilisez Kylix pour développer des applications multiplates-formes, son *Guide du développeur* est fait pour l'environnement Linux. Vous pouvez consulter ce manuel dans l'aide en ligne de Kylix ou sur la version imprimée livrée avec Kylix.

Propriétés, méthodes et événements

VCL et CLX sont toutes deux une hiérarchie d'objets, intégrée à l'EDI de Delphi, qui vous permet de développer rapidement des applications. Les objets des deux bibliothèques sont basés sur des propriétés, des méthodes et des événements. Chaque objet contient des données membres (propriétés), des fonctions qui opèrent sur les données (méthodes) et un moyen d'interagir avec les utilisateurs des classes (événements). La VCL est écrite en Pascal Objet, alors que CLX est basée sur Qt, une bibliothèque de classes C++.

Propriétés

Les *Propriétés* sont les caractéristiques d'un objet, relatives à son comportement visible ou aux opérations qu'il effectue. Par exemple, la propriété *Visible* détermine si un objet doit être vu ou non dans l'interface d'une application. Des propriétés bien conçues simplifient l'utilisation de vos composants par d'autres programmeurs et en facilite la maintenance.

Voici quelques fonctionnalités utiles des propriétés :

- Alors que les méthodes ne sont disponibles qu'à l'exécution, vous pouvez accéder aux propriétés et les modifier au cours de la conception et obtenir une réponse immédiate des composants dans l'EDI.
- Les propriétés peuvent être accédées via l'inspecteur d'objets dans lequel vous pouvez changer les valeurs visuellement. Définir les propriétés au moment de la conception est plus simple qu'écrire directement le code, et ce dernier est plus facile à maintenir.
- Comme les données sont encapsulées, elles sont protégées et privées pour l'objet réel.
- Les appels effectués pour obtenir ou définir des valeurs sont des méthodes, et donc le traitement reste invisible pour l'utilisateur de l'objet. Par exemple, les données peuvent résider dans une table, mais apparaître au programmeur comme des données membres normales.
- Vous pouvez implémenter la logique qui déclenche des événements ou modifier d'autres données pendant l'accès à la propriété. Par exemple, changer la valeur d'une propriété peut nécessiter la modification d'une autre. Vous pouvez effectuer ce changement dans les méthodes créées pour la propriété
- Les propriétés peuvent être virtuelles.
- Une propriété n'est pas limitée à un seul objet. Changer une propriété d'un objet peut affecter plusieurs autres objets. Par exemple, définir la propriété *Checked* d'un bouton radio affecte tous les autres boutons radio du groupe.

Méthodes

Une *méthode* est une procédure qui est toujours associée à une classe. Les méthodes définissent le comportement d'un objet. Les méthodes de classe peuvent accéder à toutes les propriétés *publiques*, *protégées* et *privées* et aux données membres de la classe, on les désigne fréquemment par le terme fonctions membres.

Evénements

Un *événement* est une action ou une occurrence détectée par un programme. La plupart des applications modernes sont dites pilotées par événements, car elles sont conçues pour répondre à des événements. Dans un programme, le programmeur n'a aucun moyen de prévoir la séquence exacte des actions que va entreprendre l'utilisateur. Il peut choisir un élément de menu, cliquer sur un bouton ou sélectionner du texte. Vous allez donc écrire le code qui gère chacun

des événements qui vous intéressent au lieu d'écrire du code s'exécutant toujours selon le même ordre.

Quelle que soit la façon dont un événement a été appelé, Delphi recherche si du code a été écrit pour gérer cet événement. Si c'est le cas, ce code est exécuté, sinon, le comportement par défaut se produit.

Les types d'événements qui peuvent survenir se divisent en deux grandes catégories :

- Evénements utilisateur
- Evénements système

Quelle que soit la façon dont l'événement a été appelé, Delphi recherche si vous avez écrit du code pour gérer cet événement. Si c'est le cas, ce code est exécuté, sinon rien ne se passe.

Evénements utilisateur

Les événements utilisateur sont des actions initiées par l'utilisateur. Les événements utilisateur sont, par exemple, *OnClick* (l'utilisateur a cliqué avec la souris), *OnKeyPress* (l'utilisateur a appuyé sur une touche du clavier) et *OnDblClick* (l'utilisateur a double-cliqué sur un bouton de la souris). Ces événements sont toujours rattachés à une action de l'utilisateur.

Evénements système

Ce sont des événements que le système d'exploitation déclenche pour vous. Par exemple, l'événement *OnTimer* (le composant Timer déclenche l'un de ces événements lorsqu'un intervalle prédéfini s'est écoulé), l'événement *OnCreate* (le composant est en train d'être créé), l'événement *OnPaint* (un composant ou une fenêtre a besoin d'être redessiné), etc. En règle générale, ces événements ne sont pas directement déclenchés par des actions de l'utilisateur.

Pascal Objet et les bibliothèques de classes

Le Pascal Objet, qui est constitué d'un ensemble d'extensions orientées objet du Pascal standard, est le langage de Delphi. En utilisant la palette des composants Delphi et l'inspecteur d'objets, vous pouvez placer des composants VCL ou CLX dans des fiches et manipuler leurs propriétés sans écrire de code.

Tous les objets VCL descendent de *TObject* qui est une classe abstraite dont les méthodes encapsulent des comportements essentiels comme la construction, la destruction et la gestion des messages. *TObject* est l'ancêtre immédiat de nombreuses classes simples.

Les *composants* de la VCL descendent de la classe abstraite *TComponent*. Les composants sont des objets que vous pouvez manipuler dans les fiches à la conception. Les composants visuels, c'est-à-dire ceux qui comme *TForm* ou *TSpeedButton* apparaissent à l'écran lors de l'exécution, sont appelés des *contrôles* et descendent de *TControl*.

En plus des composants visuels, les bibliothèques de composants contiennent de nombreux objets non visuels. L'EDI vous permet d'ajouter de nombreux composants non-visuels à vos programmes en les déposant dans les fiches. Si, par exemple, vous écrivez une application qui se connecte à une base de données, vous pouvez placer un composant *TDataSource* dans une fiche. Même si *TDataSource* est non-visuel, il est
représenté dans la fiche par une icône qui n'apparaît pas à l'exécution. Vous pouvez manipuler les propriétés et événements de *TDataSource* à l'aide de l'inspecteur d'objets comme pour un contrôle visuel.

Quand vous écrivez vos propres classes en Pascal Objet, elles doivent descendre de *TObject* dans la bibliothèque de classes que vous prévoyez utiliser. Utilisez VCL si vous écrivez une application Windows ou CLX si vous écrivez une application multiplate-forme. En dérivant de nouvelles classes à partir de la classe de base appropriée (ou de l'un de ses descendants), vous fournissez à vos classes des fonctionnalités essentielles tout en garantissant qu'elles peuvent fonctionner avec les autres classes de la bibliothèque des classes.

Utilisation du modèle objet

La programmation orientée objet (POO) est une extension de la programmation structurée qui intensifie la réutilisation du code et l'encapsulation de données avec des fonctionnalités. Quand vous avez créé un objet (ou, plus précisément, une classe), vous et d'autres programmeurs pouvez l'utiliser dans d'autres applications ce qui réduit les temps de développement et accroît la productivité.

Pour créer de nouveaux composants et les placer dans la palette de composants, voir chapitre 40, "Présentation générale de la création d'un composant".

Qu'est-ce qu'un objet ?

Un objet, ou *classe*, est un type de données qui encapsule des *données* et des *opérations sur ces données*. Avant la programmation orientée objet, les données et les opérations (les fonctions) constituaient des éléments distincts.

Vous pouvez commencer à comprendre les objets si vous comprenez les enregistrements Pascal Objet ou les *structures* C. Les enregistrements sont constitués de champs qui contiennent des données, chaque champ ayant son propre type. Les enregistrements sont un moyen commode de désigner une collection éléments de données variés.

Les objets sont également des collections d'éléments de données. Mais les objets, à la différence des enregistrements, contiennent des procédures et fonctions portant sur leurs données. Ces procédures et fonctions sont appelées des *méthodes*.

Les éléments de données d'un objet sont accessibles via des *propriétés*. Les propriétés des objets VCL ou CLX ont une valeur qu'il est possible de modifier à la conception sans écrire de code. Si vous voulez modifier la valeur d'une propriété à l'exécution, il vous suffit d'écrire un minimum de code.

La combinaison des données et de fonctionnalités dans un seul élément est appelée *encapsulation*. Outre l'encapsulation, la programmation orientée objet est caractérisée par *l'héritage* et le *polymorphisme*. Héritage signifie que les objets dérivent leurs fonctionnalités d'autres objets (appelés *ancêtres*) ; les objets peuvent modifier leurs comportements hérités. Polymorphisme signifie que différents objets qui dérivent d'un même ancêtre gèrent la même interface de méthode et de propriété, on dit aussi qu'ils sont interchangeables.

Examen d'un objet Delphi

Quand vous créez un nouveau projet, Delphi affiche une nouvelle fiche que vous pouvez personnaliser. Dans l'éditeur de code, Delphi déclare un nouveau type de classe pour la fiche et génère le code qui crée la nouvelle instance de fiche. Le code généré pour une nouvelle application Windows ressemble à ceci :

```
unit Unit1;
interface

uses Windows, Classes, Graphics, Forms, Controls, Dialogs;

type
  TForm1 = class(TForm) { La déclaration de type de la fiche commence ici }
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end; { La déclaration de type de la fiche s'arrête ici }

var
  Form1: TForm1;

implementation { Début de la partie implémentation }
{$R *.DFM}
end.{ Fin de la partie implémentation et de l'unité }
```

Le nouveau type de classe est *TForm1*, il dérive du type *TForm* qui est également une classe.

Une classe ressemble à un enregistrement car tous les deux contiennent des champs de données, mais une classe contient également des méthodes (du code agissant sur les données de l'objet). Pour le moment, *TForm1* semble ne contenir ni champs, ni méthodes car vous n'avez ajouté à la fiche aucun composant (les champs du nouvel objet) et vous n'avez pas créé de gestionnaires d'événements (les méthodes du nouvel objet). *TForm1* contient les champs et méthodes hérités même s'ils n'apparaissent pas dans la déclaration de type.

La déclaration de variable suivante déclare une variable nommée *Form1* ayant le nouveau type *TForm1*.

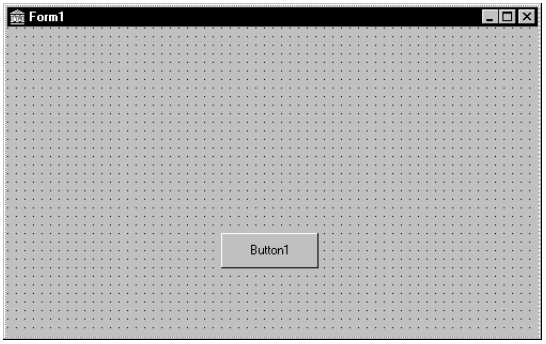
```
var
  Form1: TForm1;
```

Form1 représente une instance, ou objet, du type de classe *TForm1*. Vous pouvez déclarer plusieurs instances d'un type de classe ; par exemple, pour créer plusieurs fenêtres enfant dans une application utilisant l'interface d'une application à documents multiples (MDI). Chaque instance a ses données propres, mais elle utilise le même code pour exécuter les méthodes.

Même si vous n'avez ajouté aucun composant à la fiche ni écrit de code, vous disposez déjà d'une application Delphi que vous pouvez compiler et exécuter. Elle se contente d'afficher une fiche vide.

Vous pouvez alors ajouter un composant bouton à la fiche et écrire un gestionnaire d'événement *OnClick* qui modifie la couleur de la fiche quand l'utilisateur clique sur le bouton. Le résultat a l'aspect suivant :

Figure 3.1 Une fiche simple



Quand l'utilisateur clique sur le bouton, la couleur de la fiche passe au vert. Voici le code du gestionnaire d'événement pour l'événement *OnClick* du bouton :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Color := clGreen;
end;
```

Un objet peut contenir d'autres objets dans ses champs de données. A chaque fois que vous placez un composant dans une fiche, un nouveau champ apparaît dans la déclaration de type de la fiche. Si vous créez l'application qui vient d'être décrite, et examinez le code dans l'éditeur de code, vous trouvez :

```
unit Unit1;

interface

uses Windows, Classes, Graphics, Forms, Controls;

type
  TForm1 = class(TForm)
    Button1: TButton;{ Nouveau champ de données }
    procedure Button1Click(Sender: TObject);{ Nouvelle déclaration de méthode }
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
```

```
procedure TForm1.Button1Click(Sender: TObject);{ Le code de la nouvelle méthode }
begin
  Form1.Color := clGreen;
end;

end.
```

TForm1 possède un champ *Button1* qui correspond au bouton que vous avez ajouté à la fiche. *TButton* est du type classe, de sorte que *Button1* fait référence à un objet.

Tous les gestionnaires d'événements que vous écrivez dans Delphi sont des méthodes de l'objet fiche. Chaque fois que vous créez un gestionnaire d'événement, une méthode est déclarée dans le type de l'objet fiche. Le type *TForm1* contient maintenant une nouvelle méthode, la procédure *Button1Click*, déclarée à l'intérieur de la déclaration de type de *TForm1*. Le code qui implémente la méthode *Button1Click* apparaît dans la partie **implementation** de l'unité.

Modification du nom d'un composant

Vous devez toujours utiliser l'inspecteur d'objet pour modifier le nom d'un composant. Par exemple, vous pouvez changer le nom par défaut de la fiche *Form1* pour lui donner un nom plus parlant, comme *ColorBox*. Quand vous modifiez la valeur de la propriété *Name* de la fiche dans l'inspecteur d'objets, le nouveau nom est automatiquement reflété dans le fichier .dfm ou .xfm de la fiche (que vous ne devez généralement pas modifier directement) et dans le code source Pascal Objet généré par Delphi :

```
unit Unit1;

interface

uses Windows, Classes, Graphics, Forms, Controls;

type
  TColorBox = class(TForm){ remplacement de TForm1 par TColorBox }
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  ColorBox: TColorBox;{ Remplacement de Form1 par ColorBox }

implementation

{$R *.DFM}

procedure TColorBox.Button1Click(Sender: TObject);
begin
  Form1.Color := clGreen;{ La référence à Form1 n'a pas été actualisée ! }
end;

end.
```

Remarquez que le code du gestionnaire d'événement *OnClick* du bouton n'a pas été modifié. Comme vous avez écrit le code, vous devez le mettre à jour et corriger les références à la fiche vous-même :

```
procedure TColorBox.Button1Click(Sender: TObject);
begin
  ColorBox.Color := clGreen;
end;
```

Héritage des données et du code d'un objet

L'objet *TForm1* décrit paraît simple. *TForm1* semble ne contenir qu'un seul champ (*Button1*), une méthode (*Button1Click*) et aucune propriété. Vous pouvez néanmoins afficher, masquer et redimensionner la fiche, ajouter ou retirer les icônes standard en haut de la fiche ou la configurer pour qu'elle fasse partie d'une application à interface de document multiple (MDI). Vous pouvez faire tout cela car la fiche a *hérité* de toutes les propriétés et méthodes du composant VCL *TForm*. Quand vous ajoutez une nouvelle fiche à votre projet, vous partez de *TForm* que vous personnalisez en lui ajoutant des composants, en modifiant la valeur des propriétés et en écrivant des gestionnaires d'événements. Pour personnaliser un objet, il faut d'abord dériver un nouvel objet d'un objet existant ; quand vous ajoutez une nouvelle fiche à un projet, Delphi dérive automatiquement une nouvelle fiche du type *TForm* :

```
TForm1 = class(TForm)
```

Un objet dérivé hérite de toutes les propriétés, événements et méthodes de l'objet dont il dérive. L'objet dérivé est appelé un *descendant* et l'objet dont il dérive est appelé son *ancêtre*. Si vous recherchez *TForm* dans l'aide en ligne, vous trouverez la liste de ses propriétés, événements et méthodes, y compris ceux que *TForm* a hérité de *ses* ancêtres. Un objet ne peut avoir qu'un seul ancêtre immédiat, mais il peut avoir plusieurs descendants directs.

Portée et qualificateurs

La *portée* détermine l'accessibilité des champs, propriétés et méthodes d'un objet. Tous les membres déclarés par un objet sont accessibles dans l'objet et dans ses descendants. Même si le code d'implémentation d'une méthode apparaît hors de la déclaration de l'objet, la méthode reste dans la portée de l'objet car elle est déclarée dans la déclaration de l'objet.

Quand vous écrivez du code pour implémenter une méthode qui désigne les propriétés, méthodes ou champs de l'objet dans lequel la méthode est déclarée, vous n'avez pas besoin de préfixer ces identificateurs avec le nom de l'objet. Si, par exemple, vous placez un bouton dans une nouvelle fiche, vous pouvez écrire le gestionnaire d'événement suivant pour l'événement *OnClick* du bouton :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Color := clFuchsia;
  Button1.Color := clLime;
end;
```

La première instruction est équivalente à :

```
Form1.Color := clFuchsia
```

Il n'est pas nécessaire de qualifier *Color* avec *Form1* car la méthode *Button1Click* fait partie de *TForm1* ; les identificateurs placés dans la méthode sont alors dans la portée de l'instance de *TForm1* où la méthode a été déclarée. Par contre, la seconde instruction désigne la couleur de l'objet bouton, pas celle de la fiche dans laquelle le gestionnaire d'événement est déclaré, elle doit donc être qualifiée.

Delphi crée un fichier unité séparé (du code source) pour chaque fiche. Si vous souhaitez accéder aux composants d'une fiche depuis le fichier unité d'une autre fiche, vous devez qualifier les noms de composants de la manière suivante :

```
Form2.Edit1.Color := clLime;
```

De la même manière, vous pouvez accéder aux méthodes d'un composant d'une autre fiche. Par exemple,

```
Form2.Edit1.Clear;
```

Pour accéder aux composants de *Form2* depuis le ficher unité de *Form1*, vous devez également ajouter l'unité de *Form2* à la clause **uses** de l'unité de *Form1*.

La portée d'un objet s'étend aux descendants de l'objet. Vous pouvez néanmoins redéclarer un champ, une propriété ou une méthode dans un objet descendant. De telles redéclarations masquent ou surchargent le membre hérité.

Pour davantage d'informations sur la portée, l'héritage et la clause **uses**, voir le *Guide du langage Pascal Objet*.

Déclarations privées, protégées, publiques et publiées

Quand vous déclarez un champ, une propriété ou une méthode, la *visibilité* du nouveau membre est indiquée par l'un des mots-clés suivants : **private**, **protected**, **public** ou **published**. La visibilité d'un membre détermine son accessibilité par d'autres objets et unités.

- Un membre privé (**private**) est accessible uniquement dans l'unité dans laquelle il a été déclaré. Les membres privés sont souvent utilisés dans une classe pour implémenter d'autres méthodes et propriétés (publiques ou publiées).
- Un membre protégé (**protected**) est accessible dans l'unité dans laquelle la classe est déclarée et dans toute classe descendante, quelle que soit l'unité de la classe du descendant.
- Un membre public (**public**) est accessible partout où l'objet auquel il appartient est accessible, c'est-à-dire dans l'unité dans laquelle la classe est déclarée et dans toute unité utilisant cette unité.
- Un membre publié (**published**) a la même visibilité qu'un membre public mais le compilateur génère des informations de type à l'exécution pour les membres publiés. Les propriétés publiées apparaissent dans l'inspecteur d'objets à la conception.

Pour davantage d'informations sur la visibilité, voir le *Guide du langage Pascal Objet*.

Utilisation de variables objet

Vous pouvez affecter une variable objet à une autre variable objet si ces variables sont de même type ou de types compatibles pour l'affectation. En particulier, vous pouvez affecter une variable objet à une autre variable objet si le type de la variable affectée est un ancêtre du type de la variable qui est affectée. Par exemple, voici la déclaration du type *TDataForm* (VCL seulement) et une section déclaration de variables dans laquelle deux variables sont déclarées, *AForm* et *DataForm* :

```
type
  TDataForm = class(TForm)
  Button1: TButton;
    Edit1: TEdit;
    DataGrid1: TDataGrid;
    Database1: TDatabase;
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  AForm: TForm;
  DataForm: TDataForm;
```

AForm est de type *TForm* et *DataForm* est de type *TDataForm*. Comme *TDataForm* est un descendant de *TForm*, l'instruction d'affectation suivante est légale :

```
AForm := DataForm;
```

Supposez que vous remplissiez le gestionnaire d'événement de l'événement *OnClick* d'un bouton. Quand le bouton est choisi, le gestionnaire d'événement de l'événement *OnClick* est appelé. Chaque gestionnaire d'événement a un paramètre *Sender* de type *TObject* :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
⋮
end;
```

Comme *Sender* est de type *TObject*, tout objet peut être affecté à *Sender*. La valeur de *Sender* est toujours le contrôle ou le composant qui répond à l'événement. Vous pouvez tester *Sender* pour déterminer le type du composant ou du contrôle qui a appelé le gestionnaire d'événement en utilisant le mot réservé **is**. Par exemple,

```
if Sender is TEdit then
  FaireQuelqueChose
else
  FaireAutreChose;
```

Création, instanciation et destruction d'objets

La plupart des objets que vous utilisez dans Delphi sont des composants qui, comme les boutons ou les zones de saisie, apparaissent à la conception et à l'exécution. Certains, comme les boîtes de dialogue standard, apparaissent uniquement à l'exécution. D'autres, comme les composants timer ou source de données, n'ont pas de représentation visuelle dans l'application à l'exécution.

Vous pouvez être amené à créer vos propres objets. Vous pouvez, par exemple, créer un objet *TEmployee* contenant les champs *Name*, *Title* et *HourlyPayRate*. Vous pouvez ensuite lui ajouter la méthode *CalculatePay* qui utilise les données du champ *HourlyPayRate* pour calculer le montant de la paye. La déclaration du *type TEmployee* peut prendre la forme suivante :

```
type
  TEmployee = class(TObject)
  private
    FName: string;
    FTitle: string;
    FHourlyPayRate: Double;
  public
    property Name: string read FName write FName;
    property Title: string read FTitle write FTitle;
    property HourlyPayRate: Double read FHourlyPayRate write FHourlyPayRate;
    function CalculatePay: Double;
  end;
```

Outre les champs, propriétés et méthodes que vous avez défini, *TEmployee* hérite de toutes les méthodes de *TObject*. Vous pouvez placer une déclaration de type comme celle-ci dans la partie **interface** ou dans la partie **implementation** d'une unité et créer des instances de la nouvelle classe en appelant la méthode *Create* que *TEmployee* hérite de *TObject* :

```
var
  Employee: TEmployee;
begin
  Employee := TEmployee.Create;
end;
```

La méthode *Create* est appelée un *constructeur*. Elle alloue la mémoire pour un objet nouvellement instancié et renvoie une référence sur l'objet.

Les composants d'une fiche sont créés et détruits automatiquement par Delphi. Mais si vous écrivez votre propre code pour instancier des objets, vous êtes responsable de leur libération. Chaque objet hérite de *TObject* une méthode *Destroy* (appelée un *destructeur*). Néanmoins, pour détruire un objet, vous devez toujours appeler la méthode *Free* (également héritée de *TObject*) car *Free* cherche une référence **nil** avant d'appeler *Destroy*. Par exemple,

```
Employee.Free
```

détruit l'objet *Employee* et libère sa mémoire.

Composants et appartenance

Delphi dispose d'un mécanisme intégré de gestion de la mémoire qui permet à un composant d'être responsable de la libération d'un autre composant. On dit que le premier est *propriétaire* du second. La mémoire d'un composant appartenant à un autre est automatiquement libérée quand la mémoire du propriétaire est libérée. Le propriétaire d'un composant (valeur de sa propriété *Owner*) est déterminé par un paramètre transmis au constructeur lors de la création du composant. Par défaut, une fiche possède tous les composants placés dedans et elle-même appartient à l'application. Ainsi, quand l'application est arrêtée, la mémoire de toutes les fiches et de leurs composants est libérée.

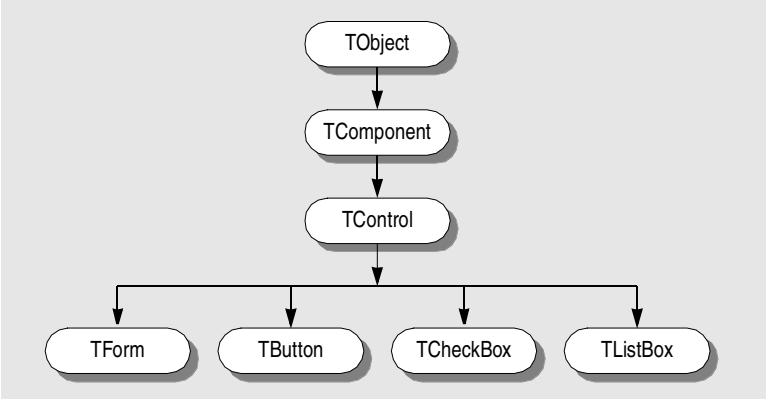
La propriété s'applique uniquement à *TComponent* et à ses descendants. Si vous créez un objet *TStringList* ou *TCollection* (même s'il est associé à une fiche), c'est à vous de libérer l'objet.

Remarque Ne confondez pas le *propriétaire* d'un composant avec son *parent*. Voir "Propriétés du parent" à la page 3-21.

Objets, composants et contrôles

Le diagramme suivant est une vue très simplifiée de la hiérarchie des héritages qui illustre les relations entre objets, composants et contrôles.

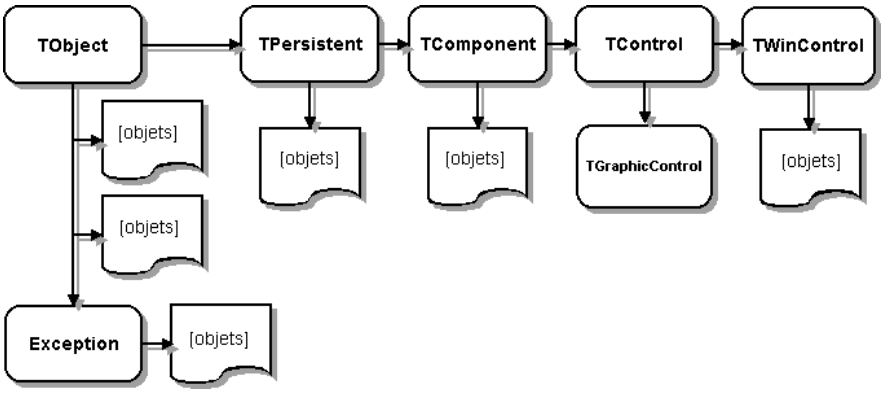
Figure 3.2 Objets, composants et contrôles



Chaque objet hérite de *TObject* et beaucoup d'objets héritent de *TComponent*. Les contrôles, qui héritent de *TControl*, ont la capacité à s'afficher eux-mêmes à l'exécution. Un contrôle comme *TCheckBox* hérite de toutes les fonctionnalités de *TObject*, *TComponent* et de *TControl*, et ajoute ses spécificités propres.

Le diagramme suivant est un résumé de la bibliothèque de composants visuels (VCL) qui montre les principales branches de l'arbre d'héritage. La bibliothèque de composants Borland multiplate-forme (CLX) lui ressemble beaucoup à ce niveau, mais *TWinControl* est remplacé par *TWidgetControl*.

Figure 3.3 Diagramme simplifié de la hiérarchie



Plusieurs importantes classes de base montrées dans la figure sont décrites dans le tableau suivant :

Tableau 3.1 Importantes classes de base

Classe	Description
TObject	C'est la classe de base et l'ancêtre ultime de tout ce qui se trouve dans la VCL ou dans CLX. *TObject* encapsule le comportement fondamental commun à tous les objets VCL/ CLX, en introduisant les méthodes qui effectuent les fonctions de base, comme la création, la maintenance et la destruction d'une instance d'objet.
Exception	Spécifie la classe de base de toutes les classes relatives aux exceptions. *Exception* fournit une interface cohérente pour les conditions d'erreur et permet aux applications de les gérer harmonieusement.
TPersistent	Spécifie la classe de base de tous les objets qui implémentent des propriétés. Les classes issues de *TPersistent* prennent en charge l'envoi de données aux flux et permettent l'affectation des classes.
TComponent	Spécifie la classe de base de tous les composants non visuels, comme *TApplication*. *TComponent* est l'ancêtre commun de tous les composants. Cette classe permet à un composant de figurer sur la palette des composants, de posséder d'autres composants et d'être manipulé directement sur une fiche.
TControl	Représente la classe de base de tous les contrôles visibles à l'exécution. *TControl* est l'ancêtre commun de tous les composants visuels et fournit les contrôles visuels standard comme la position et le curseur. Cette classe fournit également des événements qui répondent aux actions de la souris.
TWinControl	Spécifie la classe de base de tous les objets d'interface utilisateur, appelés également widgets. Les contrôles issus de *TWinControl* sont des contrôles fenêtrés qui peuvent capturer la saisie au clavier. (Dans CLX, *TWidgetControl* remplace *TWinControl*.)

Les quelques sections suivantes présentent une description générale des types de classes que contient chaque branche. Pour avoir une présentation complète de la hiérarchie des objets VCL, reportez-vous au poster VCL fourni avec ce produit. Pour avoir des détails sur CLX, reportez-vous au poster sur la hiérarchie des objets CLX, fourni avec ce produit, et à la documentation Kylix.

Branche TObject

La branche *TObject* comprend tous les objets qui descendent de *TObject* mais non de *TPersistent*. Tous les objets VCL ou CLX descendent de *TObject*, une classe abstraite dont les méthodes définissent des comportements fondamentaux comme la construction, la destruction et la gestion des messages ou des événements système. L'essentiel des capacités des objets VCL et CLX est basé sur les méthodes définies dans *TObject*. *TObject*encapsule le comportement fondamental commun à tous les objets VCL ou CLX, en introduisant des méthodes qui permettent :

- De répondre à la création ou la destruction d'instances d'objets.
- De donner des informations sur le type de classe et d'instance d'un objet et des informations de type à l'exécution (RTTI) sur ses propriétés publiées.
- La gestion des messages (VCL seulement).

TObject est l'ancêtre immédiat de nombreuses classes simples. Les classes contenues dans cette branche ont une caractéristique commune importante : elles sont transitoires. Cela signifie que ces classes ne disposent pas d'une méthode pour enregistrer leur état avant leur destruction ; elles ne sont pas persistantes.

Le groupe de classes le plus important de cette branche est constitué par la classe *Exception*. Cette classe propose un grand nombre de classes d'exceptions prédéfinies pour gérer automatiquement de nombreuses conditions d'exception comme les erreurs de division par zéro, les erreurs d'entrées/sorties ou les transtypages incorrects.

La branche *TObject* contient également un autre groupe de classes qui encapsulent des structures de données, comme :

- *TBits,* une classe qui stocke un "tableau" de valeur booléennes.
- *TList,* une classe liste liée.
- *TStack,* une classe qui gère un tableau de pointeurs du type dernier entré, premier sorti.
- *TQueue,* une classe qui gère un tableau de pointeurs du type premier entré, premier sorti.

Dans la VCL, vous trouverez aussi des enveloppes pour les objets externes comme *TPrinter*, qui encapsule l'interface imprimante Windows, et *TRegistry,* une enveloppe de bas niveau pour le registre du système et les fonctions qui opèrent sur le registre. Elles sont spécifiques à l'environnement Windows.

TStream est un bon exemple du type de classes contenues dans cette branche. *TStream* est la classe de base des objets flux qui permettent de lire ou d'écrire sur divers types de support de données, comme les fichiers disque ou la mémoire vive.

Comme vous pouvez le voir, cette branche contient un grand nombre de différents types de classes particulièrement utiles pour le développeur.

Branche TPersistent

Les objets de cette branche de VCL et de CLX descendent de *TPersistent* mais pas de *TComponent*. *TPersistent* ajoute aux objets la persistance. La persistance détermine ce qui est enregistré dans un fichier fiche ou un module de données et ce qui est chargé dans la fiche ou le module de données lorsqu'il est extrait de la mémoire.

Les objets de cette branche implémentent des propriétés pour les composants. Les propriétés sont uniquement chargés et enregistrés avec une fiche si elles ont un propriétaire. Le propriétaire doit être un composant. Cette branche introduit la fonction *GetOwner* qui vous permet de déterminer le propriétaire de la propriété.

Les objets de cette branche sont également les premiers à inclure une section publiée dans laquelle les propriétés peuvent être automatiquement chargées et enregistrées. Un méthode *DefineProperties* vous permet aussi d'indiquer la façon de charger et d'enregistrer les propriétés.

Voici quelques autres classes de la branche TPersistent de la hiérarchie :

- *TGraphicsObject*, une classe de base abstraite pour les objets graphiques, par exemple : *TBrush*, *TFont* et *TPen*.
- *TGraphic*, une classe de base abstraite pour les objets comme les icônes et les bitmaps qui peuvent stocker et afficher des images visuelles, par exemple : *TBitmap* et *TIcon* (et pour le développement Windows seulement : *TMetafile)*.
- *TStrings*, une classe de base pour les objets qui représentent une liste de chaînes.
- *TClipboard*, une classe contenant du texte ou des graphiques qui ont été coupés ou copiés d'une application.
- *TCollection*, *TOwnedCollection* et *TCollectionItem*, classes qui maintiennent les collections indexés d'éléments spécialement définis.

Branche TComponent

La branche *TComponent* contient des objets qui descendent de *TComponent* mais pas de *TControl*. Les objets de cette branche sont des composants que vous pouvez manipuler sur des fiches au cours de la conception. Ce sont des objets persistants aux capacités suivantes :

- Ils apparaissent dans la palette des composants et peuvent être modifiés dans le concepteur de fiche.
- Ils peuvent posséder et gérer d'autres composants.
- Ils se chargent et s'enregistrent eux-mêmes.

Plusieurs méthodes de *TComponent* dictent la façon dont agissent les composants durant la conception et les informations qui sont enregistrées avec le composant. La gestion des flux est introduite dans cette branche de la VCL et de CLX. Delphi gère automatiquement la plupart des opérations principales relatives aux

flux. Les propriétés sont persistantes si elles sont publiées et les propriétés publiées sont automatiquement mises en flux.

La classe *TComponent* introduit également le concept de possession (propriété) qui se propage dans la VCL et dans CLX. Deux propriétés supportent la possession (propriété) : *Owner* et *Components*. Chaque composant dispose d'une propriété *Owner* qui référence un autre composant comme sont propriétaire. Un composant peut posséder d'autres composants. Dans ce cas, tous les composants possédés sont référencés dans la propriété *Array* du composant.

Le constructeur d'un composant prend un seul paramètre qui est utilisé pour spécifier le propriétaire du nouveau composant. Si le propriétaire transmis existe, le nouveau composant est ajouté à la liste des composants du propriétaire. Outre la liste des composants, pour référencer les composants possédés, cette propriété fournit la destruction automatique des composants possédés. Si le composant a un propriétaire, il est détruit lorsque le propriétaire est détruit. Par exemple, comme *TForm* est un descendant de *TComponent*, tous les composants possédés par la fiche sont détruits, et leur emplacement en mémoire libéré, lorsque la fiche est détruite. Cela garantit que tous les composants de la fiche se nettoient eux-mêmes correctement lorsque leurs destructeurs sont appelés.

Si un type de propriété est *TComponent* ou un de ses descendants, le système de flux crée une instance de ce type lorsqu'il le lit. Si un type de propriété est *TPersistent* mais pas *TComponent*, le système de flux utilise l'instance existante, accessible via la propriété, et lit les valeurs des propriétés de cette instance.

Quand il crée un fichier fiche (fichier utilisé pour stocker les informations relatives aux composants de la fiche), le concepteur de fiche parcourt le tableau de ses composants et enregistre tous les composants dans la fiche. Chaque composant "sait" comment écrire dans un flux (dans ce cas, un fichier texte) ses propriétés modifiées. Parallèlement, lorsqu'il charge les propriétés des composants du fichier fiche, le concepteur de fiche parcourt le tableau des composants et charge chacun d'eux.

Cette branche contient, entre autres, les types de classes suivants :

- *TMainMenu*, une classe qui définit une barre de menus et les menus déroulants associés dans une fiche.
- *TTimer*, une classe qui inclut les fonctions de timer.
- *TOpenDialog*, *TSaveDialog*, *TFontDialog*, *TFindDialog*, *TColorDialog*, etc., fournissent les boîtes de dialogues communes.
- *TActionList*, une classe qui gère une liste d'actions utilisées par des composants et des contrôles, comme les éléments de menu et les boutons.
- *TScreen*, une classe qui mémorise les fiches et les modules de données instanciés par l'application, la fiche active et le contrôle actif dans cette fiche, la taille et la résolution de l'écran, ainsi que les curseurs et les fontes utilisables par l'application.

Les composants qui n'ont pas besoin d'interface visuelle peuvent être directement dérivés de *TComponent*. Pour faire un outil tel qu'un périphérique *TTimer*, vous devez le dériver de *TComponent*. Ce type de composant se trouve sur la palette des composants, mais exécute des fonctions internes accessibles par le code et qui n'apparaissent pas, à l'exécution, dans l'interface utilisateur.

Dans CLX, la branche *TComponent* contient aussi *THandleComponent*. C'est la classe de base des composants non visuels qui nécessitent un handle sur un objet Qt sous-jacent, comme les dialogues et les menus.

Branche TControl

La branche *TControl* est constituée de composants qui descendent de *TControl* mais pas de *TWinControl (TWidgetControl dans CLX)*. Les objets de cette branche sont des contrôles, c'est-à-dire des objets visuels que l'utilisateur de l'application peut voir et manipuler à l'exécution. Tous les contrôles ont des propriétés, méthodes et événements communs qui sont propres à l'aspect visuel des contrôles, comme la position du contrôle, le curseur associé à la fenêtre (au widget dans CLX) du contrôle, des méthodes pour dessiner ou déplacer le contrôle et des événements permettant de répondre aux actions de la souris. Ils ne peuvent jamais recevoir la saisie du clavier.

Si *TComponent* définit des comportements communs à tous les composants, *TControl* définit ceux communs à tous les contrôles visuels. Il s'agit de routines de dessin, des événements standard et de la notion de conteneur.

Les contrôles sont de deux types :

- Ceux qui ont leur propre fenêtre (ou widget)
- Ceux qui utilisent la fenêtre (ou le widget) de leur "parent"

Les contrôles qui ont leur propre fenêtre sont appelés contrôles "fenêtrés" (VCL) ou "basés sur des widgets" (CLX) et descendent de *TWinControl (TWidgetControl dans CLX)*. Les boutons et les cases à cocher en font partie.

Les contrôles qui utilisent une fenêtre (ou un widget) parent sont appelés contrôles "graphiques" et descendent de *TGraphicControl*. Les contrôles image et libellé en font partie. Dans la VCL, la principale différence entre ces types de composants est que les contrôles graphiques n'ont pas de handle de fenêtre et ne peuvent donc pas recevoir la focalisation ni contenir d'autres contrôles. Dans CLX, la principale différence entre ces types de composants est que les contrôles graphiques ne possèdent pas de widget associé et ne peuvent donc pas recevoir la focalisation ni contenir d'autres contrôles. Comme un contrôle graphique n'a pas besoin de handle, ses exigences en ressources système sont moindre et le dessin d'un graphique est plus rapide que celui d'un contrôle basé sur un widget.

Les contrôles *TGraphicControl* doivent se dessiner eux-mêmes et comprennent :

Tableau 3.2 Contrôles graphiques

Contrôle	Description
TImage	Affiche des images.
TLabel	Affiche du texte dans une fiche.
TBevel	Représente un contour biseauté.
TPaintBox	Propose un canevas dans lequel les applications peuvent dessiner ou restituer une image.

Ces classes contiennent les routines de dessin courantes (*Repaint*, *Invalidate*, etc.) qui ne peuvent pas recevoir la focalisation.

Branche TWinControl/TWidgetControl

La branche *TWinControl (TWidgetControl remplace TWinControl dans CLX)* contient tous les contrôles qui descendent de *TWinControl*. *TWinControl* est la classe de base de tous les contrôles fenêtrés, qui représentent un grand nombre des éléments que vous utilisez dans l'interface utilisateur d'une application.

TWidgetControl est la classe de base de tous les contrôles basés sur des widgets ou *widgets*. Le terme widget est une combinaison de "window" (fenêtre) et de "gadget". Pratiquement tout ce que vous utilisez dans l'interface utilisateur d'une application est un widget. Par exemple, les boutons, les libellés et les barres de défilement.

Voici les caractéristiques des contrôles fenêtrés et des widgets :

- Les deux peuvent recevoir la focalisation pendant l'exécution de l'application.
- Les autres contrôles peuvent afficher des données mais l'utilisateur peut utiliser le clavier pour interagir avec un contrôle fenêtré ou avec un contrôle basé sur un widget.
- Les contrôles fenêtrés ou basés sur des widgets peuvent contenir d'autres contrôles.
- Un contrôle qui contient d'autres contrôles est un parent. Seul un contrôle fenêtré ou basé sur un widget peut être le parent d'un ou de plusieurs contrôles enfants.
- Les contrôles fenêtrés disposent d'un handle de fenêtre. Les contrôles basés sur des widgets sont associés à un widget.

Les descendants de *TWinControl (TWidgetControl dans CLX)* sont des contrôles qui peuvent recevoir la focalisation, c'est-à-dire qu'ils peuvent recevoir la saisie au clavier de l'utilisateur de l'application. Cela signifie que beaucoup plus d'événements standard s'appliquent à eux.

Cette branche contient aussi bien les contrôles dessinés automatiquement (*TEdit*, *TListBox*, *TComboBox*, *TPageControl*, etc.) et des contrôles personnalisés que Delphi doit dessiner (par exemple *TDBNavigator*, *TMediaPlayer* (VCL seulement), *TGauge* (VCL seulement), etc.). Les descendants directs de *TWinControl (TWidgetControl dans CLX)* implémentent typiquement des contrôles standard, comme un champ d'édition, une boîte à option ou un contrôle page, et savent donc déjà comment se dessiner eux-mêmes.

La classe *TCustomControl* est fournie pour les composants qui nécessitent un handle mais n'encapsulent pas de contrôle standard ayant la capacité de se redessiner lui-même. Vous n'avez jamais à vous soucier de la façon dont les contrôles s'affichent ou répondent aux événements — Delphi encapsule complètement ce comportement pour vous.

Les sections suivantes présentent les contrôles. Reportez-vous au chapitre 7, "Manipulation des contrôles", pour plus d'informations sur l'utilisation des contrôles.

Propriétés communes à TControl

Tous les contrôles visuels (descendants de *TControl*) partagent certaines propriétés, entre autres :

- Propriétés d'action
- Propriétés de position, de taille et d'alignement
- propriétés d'affichage
- Propriétés du parent
- Une propriété de navigation
- Propriétés de glisser-déplacer
- Propriétés de glisser-ancrer (VCL seulement)

Ces propriétés sont héritées de *TControl* mais elles ne sont publiées (elles apparaissent dans l'inspecteur d'objets) que pour les composants où elles sont applicables. Ainsi, *TImage* ne publie pas la propriété *Color* car sa couleur est déterminée par l'image qu'il affiche.

Propriétés d'action

Les actions vous permettent de partager du code qui effectue certaines actions (par exemple, un bouton de barre d'outils et un élément de menu qui font la même chose) et fournissent un moyen simple et centralisé d'activer et de désactiver des actions en fonction de l'état de votre application.

- *Action* désigne l'action associée au contrôle.
- *ActionLink* contient l'objet liaison d'action associé au contrôle.

Propriétés de position, de taille et d'alignement

Cet ensemble de propriétés définissent la position et la taille d'un contrôle dans le contrôle parent :

- *Height* spécifie la taille verticale.
- *Width* spécifie la taille horizontale.
- *Top* indique la position du bord supérieur.
- *Left* indique la position du bord gauche.
- *AutoSize* spécifie si le contrôle se redimensionne automatiquement pour s'adapter à son contenu.
- *Align* détermine comment le contrôle s'aligne dans son conteneur (ou contrôle parent).
- *Anchor* spécifie comment le contrôle est ancré à son parent (VCL seulement).

Cet ensemble de propriétés détermine la hauteur, la largeur et la taille globale de la zone client d'un contrôle.

- *ClientHeight* indique la hauteur, exprimée en pixels, de la zone client du contrôle.
- *ClientWidth* indique la largeur, exprimée en pixels, de la zone client du contrôle.

Ces propriétés ne sont pas accessibles dans les composants non visuels, mais Delphi se souvient de l'endroit où vous avez placé les icônes de ces composants dans une fiche. La plupart du temps, vous définirez ou modifierez ces propriétés en manipulant l'image du contrôle dans la fiche ou en utilisant la palette Alignement. Vous pouvez également les modifier à l'exécution.

Propriétés d'affichage

Les propriétés suivantes définissent l'aspect général d'un contrôle :

- *Color* spécifie la couleur de fond d'un contrôle.
- *Font* modifie la couleur, le nom, le style ou la taille du texte.
- *Cursor* spécifie l'image utilisée pour représenter le pointeur de la souris lorsqu'il passe au-dessus de la région couverte par le contrôle.
- *DesktopFont* indique si le contrôle utilise la fonte des icônes Windows pour l'écriture de texte (VCL seulement).

Propriétés du parent

Pour conserver une présentation homogène dans une application, vous pouvez donner à un contrôle le même aspect que son conteneur, appelé son *parent*, en initialisant les propriétés du *parent* à *True*.

- *ParentColor* détermine où un contrôle cherche ses attributs de couleurs.
- *ParentFont* détermine où un contrôle cherche ses attributs de police.
- *ParentShowHint* détermine comment un contrôle décide s'il faut ou non afficher le texte de son conseil spécifié par la propriété Hint

Une propriété de navigation

La propriété suivante détermine comment les utilisateurs peuvent se déplacer entre les divers contrôles d'une fiche :

- *Caption* contient la chaîne de texte qui intitule un composant. Pour souligner un caractère d'une chaîne, faites-le précéder d'un caractère &. Ce type de caractère est appelé un accélérateur. L'utilisateur peut alors accéder au contrôle ou à l'élément de menu en appuyant sur *Alt* et sur le caractère souligné.

Propriétés de glisser-déplacer

Deux propriétés des composants affectent le comportement du glisser-déplacer :

- *DragMode* détermine la manière dont un glisser commence. Par défaut, *DragMode* a la valeur *dmManual* et vous devez appeler la méthode *BeginDrag* pour commencer un glisser. Si *DragMode* a la valeur *dmAutomatic*, le glisser commence dès que le bouton de la souris est enfoncé.
- *DragCursor* détermine la forme du pointeur de la souris quand il se trouve au-dessus d'un composant qui accepte un objet en train de glisser (VCL seulement).

Propriétés de glisser-ancrer (VCL seulement)

Les propriétés suivantes contrôlent le comportement de glisser-ancrer :

- *Floating* spécifie si le contrôle est flottant.
- *DragKind* spécifie si le contrôle est déplacé normalement ou pour un ancrage.
- *DragMode* détermine comment le contrôle commence des opérations de glisser-déplacer ou glisser-ancrer.
- *FloatingDockSiteClass* spécifie la classe utilisée pour le contrôle temporaire qui héberge le contrôle quand il est flottant.
- *DragCursor* est le curseur qui s'affiche pendant le glissement.
- *DockOrientation* spécifie comment le contrôle est ancré par rapport aux autres contrôles ancrés dans le même parent.
- *HostDockSite* spécifie le contrôle auquel ce contrôle est ancré.

Pour plus d'informations, voir "Implémentation du glisser-ancrer dans les contrôles" à la page 7-4.

Evénements standard communs à TControl

La VCL définit un ensemble d'événements standard pour ses contrôles. Les événements suivants sont déclarés comme faisant partie de la classe *TControl*, et sont donc disponibles à toutes les classes dérivées de *TControl* :

- *OnClick* se produit quand l'utilisateur clique sur le contrôle.
- *OnContextPopup* se produit lors de clics de l'utilisateur avec le bouton droit ou sinon lors d'appel du menu déroulant (comme en utilisant le clavier).
- *OnCanResize* se produit quand le contrôle est redimensionné.
- *OnResize* se produit immédiatement après que le contrôle ait été redimensionné.
- *OnConstrainedResize* se produit immédiatement après OnCanResize.
- *OnStartDock* se produit quand l'utilisateur commence à faire glisser un contrôle avec DragKind de dkDock (VCL seulement).
- *OnEndDock* se produit lorsque le glissement d'un objet se termine, soit par l'ancrage de l'objet soit par l'abandon du glissement (VCL seulement).
- *OnStartDrag* se produit lorsque l'utilisateur commence à faire glisser le contrôle ou un objet qu'il contient en cliquant dessus avec le bouton gauche de la souris et en maintenant enfoncé le bouton gauche de la souris.
- *OnEndDrag* se produit lorsque l'utilisateur arrête le glissement d'un objet, soit en déposant l'objet, soit en annulant le glissement.
- *OnDragDrop* se produit quand l'utilisateur dépose un objet qu'il faisait glisser.

- *OnMouseMove* se produit quand l'utilisateur déplace le pointeur de la souris au-dessus d'un contrôle.
- *OnDblClick* se produit quand l'utilisateur double-clique avec le bouton principal de la souris alors que le pointeur de la souris est au-dessus du contrôle.
- *OnDragOver* se produit quand l'utilisateur fait glisser un objet au-dessus d'un contrôle (VCL seulement).
- *OnMouseDown* se produit quand un utilisateur appuie sur un bouton de la souris alors que le pointeur de la souris est au-dessus d'un contrôle.
- *OnMouseUp* se produit lorsque l'utilisateur relâche un bouton de la souris qui a été enfoncé alors que le pointeur de la souris se trouvait au-dessus d'un composant.

Propriétés communes à TWinControl et TWidgetControl

Tous les contrôles visuels (descendants de*TWinControl* dans la VCL et de *TWidgetControl* dans CLX) partagent certaines propriétés, entre autres :

- Informations sur le contrôle
- Propriétés d'affichage du style de bordure
- Propriétés de navigation
- Propriétés de glisser-ancrer (VCL seulement)

Ces propriétés sont héritées de *TWinControl et de TWidgetControl*, mais elles ne sont publiées (elles apparaissent dans l'inspecteur d'objets) que pour les composants où elles sont applicables.

Propriétés d'informations générales

Les propriétés d'informations générales contiennent des informations sur l'apparence du *TWinControl et du TWidgetControl*, la taille et l'origine de la zone client, les fenêtres affectées et le contexte d'aide.

- *ClientOrigin* représente les coordonnées écran, exprimées en pixels, du coin supérieur gauche de la zone client d'un contrôle. Les coordonnées écran d'un contrôle dérivé de *TControl* mais non de *TWinControl* sont en fait les coordonnées écran du parent du contrôle ajoutées respectivement à ses propriétés *Left* et *Top*.
- *ClientRect* renvoie un rectangle dont les propriétés *Top* et *Left* sont définies par zéro et les propriétés *Bottom* et *Right* sont définies respectivement par les propriétés *Height* et *Width* du contrôle. *ClientRect* est équivalent à Rect(0, 0, *ClientWidth*, *ClientHeight*).
- *Brush* détermine la couleur et le motif utilisé pour dessiner l'arrière-plan du contrôle.
- *HelpContext* indique le numéro de contexte à utiliser pour appeler l'aide en ligne contextuelle.

- *Handle* donne accès au handle de fenêtre ou de widget du contrôle.

Propriétés d'affichage du style de bordure

Les propriétés de biseau contrôlent l'apparence des lignes, boîtes ou cadres biseautés sur les fiches et les contrôles fenêtrés de votre application.

Un plus grand nombre d'objets de la VCL publient ces propriétés ; ils ne sont pas disponibles dans CLX et les propriétés de style de bordure sont publiées sur peu d'objets.

- *InnerBevel* spécifie si le biseau interne est en relief, en creux ou plat (VCL seulement).
- *BevelKind* spécifie le type de biseau si le contrôle a des bords biseautés (VCL seulement).
- *BevelOuter* spécifie si le biseau externe à un aspect en relief, en creux ou plat.
- *BevelWidth* spécifie la largeur, exprimée en pixels, des biseaux internes et externes.
- *BorderWidth* est utilisée pour connaître ou spécifier la bordure du contrôle.
- *BevelEdges* est utilisée pour obtenir ou définir les bords biseautés du contrôle.

Propriétés de navigation

Deux propriétés supplémentaires déterminent comment les utilisateurs peuvent se déplacer entre les divers contrôles d'une fiche :

- *TabOrder* indique la position du contrôle dans l'ordre de tabulation du parent : l'ordre dans lequel les contrôles reçoivent la focalisation quand l'utilisateur appuie sur la touche *Tab*. Initialement, l'ordre de tabulation correspond à l'ordre d'ajout des composants dans la fiche. Vous pouvez le modifier en changeant *TabOrder*. *TabOrder* n'a de sens que si *TabStop* a la valeur *True*.
- *TabStop* détermine si l'utilisateur peut tabuler sur un contrôle. Si *TabStop* a la valeur *True*, le contrôle est dans l'ordre de tabulation.

Propriétés de glisser-ancrer (VCL seulement)

Les propriétés suivantes contrôlent le comportement de glisser-ancrer des objets VCL :

- *UseDockManager* spécifie si le gestionnaire d'ancrage est utilisé pour les opérations de glisser-ancrer.
- *VisibleDockClientCount* indique le nombre de contrôles visibles qui sont ancrés au contrôle fenêtré.
- *DockManager* spécifie l'interface du gestionnaire d'ancrage du contrôle.
- *DockClients* énumère les contrôles ancrés au contrôle fenêtré.
- *DockSite* spécifie si le contrôle peut être la cible d'opérations glisser-ancrer.

Pour plus d'informations, voir "Implémentation du glisser-ancrer dans les contrôles" à la page 7-4.

Evénements communs à TWinControl et TWidgetControl

Les événements suivants existent pour tous les contrôles dérivés de *TWinControl* dans la VCL (ainsi que tous les contrôles définis par Windows) et de *TWidgetControl* dans CLX. Ces événements sont en plus de ceux qui existent dans tous les contrôles.

- *OnEnter* se produit lorsque le contrôle est sur le point de recevoir la focalisation.
- *OnKeyDown* se produit quand l'utilisateur appuie sur une touche.
- *OnKeyPress* se produit quand un utilisateur appuie sur une touche alphanumérique.
- *OnKeyUp* se produit quand l'utilisateur relâche une touche enfoncée
- *OnExit* se produit quand la focalisation passe du contrôle à un autre contrôle.
- *OnMouseWheel* se produit quand l'utilisateur fait tourner la molette de la souris.
- *OnMouseWheelDown* se produit quand l'utilisateur fait tourner la molette de la souris vers le bas.
- *OnMouseWheelUp* se produit quand l'utilisateur fait tourner la molette de la souris vers le haut.

Les événements suivants concernent l'ancrage et sont disponibles uniquement dans la VCL :

- *OnUnDock* se produit quand l'application essaie de dépiler un contrôle ancré à un contrôle fenêtré (VCL seulement).
- *OnDockDrop* se produit quand un autre contrôle est ancré au contrôle (VCL seulement).
- *OnDockOver* se produit quand on fait glisser un autre contrôle par dessus le contrôle (VCL seulement).
- *OnGetSiteInfo* renvoie les informations d'ancrage du contrôle (VCL seulement).

Création de l'interface utilisateur de l'application

Tout le travail de conception visuelle de Delphi se passe sur des *fiches*. Lorsque vous ouvrez Delphi ou créez un nouveau projet, une fiche vierge s'affiche à l'écran. Vous pouvez l'utiliser pour commencer à construire l'interface de votre application, c'est-à-dire les fenêtres, les menus et les dialogues communs.

Vous concevez l'apparence de l'interface utilisateur graphique d'une application en disposant des composants visuels, boutons ou listes, sur la fiche. Delphi prend en charge les détails sous-jacents de la programmation. Vous pouvez aussi placer des composants invisibles sur les fiches pour capturer les informations issues des bases de données, effectuer des calculs et gérer d'autres interactions.

Le chapitre 6, "Conception de l'interface utilisateur des applications", donne des détails sur l'utilisation des fiches, comme la création dynamique de fiches modales, la transmission de paramètres aux fiches et la récupération de données à partir des fiches.

Utilisation de composants Delphi

De nombreux composants visuels sont fournis par l'environnement de développement lui-même, par le biais de la palette des composants. Tout le travail de conception visuelle de Delphi se passe sur des *fiches*. Lorsque vous ouvrez Kylix ou créez un nouveau projet, une fiche vierge s'affiche à l'écran. Vous sélectionnez des composants dans la palette et les déposez sur la fiche. Vous concevez l'apparence de l'interface utilisateur de l'application en disposant des composants visuels, boutons ou listes, sur la fiche. Une fois qu'un composant se trouve sur la fiche, vous pouvez ajuster sa position, sa taille ou d'autres propriétés de conception. Delphi prend en charge les détails sous-jacents de la programmation.

Les composants Delphi sont regroupés par fonctions sur différentes pages de la palette des composants. Par exemple, les composants les plus utilisés, comme ceux qui créent des menus, des boîtes d'édition ou des boutons, se trouvent dans la page Standard de la palette. Des contrôles VCL pratiques, comme les timer, boîte à peindre, lecteur de multimédia et conteneur OLE, se trouvent sur la page Système.

A première vue, les composants Delphi ressemblent aux autres classes. Mais, il existe des différences entre les composants de Delphi et les hiérarchies de classes standard avec lesquelles travaillent la plupart des programmeurs. Voici certaines de ces différences :

- Tous les composants Delphi descendent de *TComponent*.
- Les composants sont la plupart du temps utilisés tels quels et modifiés par le biais de leurs propriétés, au lieu de servir de "classes de base" à sous-classer pour ajouter des fonctionnalités ou modifier celles qui existent. Quand un composant est dérivé, on ajoute généralement du code spécifique aux fonctions membres de gestion des événements existants.
- Les composants sont alloués uniquement sur le tas et non sur la pile.
- Les propriétés des composants contiennent de façon intrinsèque des informations de type à l'exécution.
- Des composants peuvent être ajoutés à la palette de l'interface utilisateur de Delphi et manipulés sur une fiche.

Les composants offrent souvent un meilleur degré d'encapsulation que les classes standard. Considérez, par exemple, l'utilisation d'une boîte de dialogue contenant un bouton poussoir. Dans un programme Windows développé en utilisant les composants de la VCL, lorsqu'un utilisateur clique sur le bouton, le système génère un message WM_LBUTTONDOWN. Le programme doit intercepter ce message (généralement dans une instruction **switch**, une correspondance de message ou une table de réponse) et le diriger vers une routine qui s'exécutera en réponse au message.

La plupart des messages Windows (VCL) ou événements système (CLX) sont gérés par les composants de Delphi. Quand vous voulez répondre à un message, il vous suffit de fournir un gestionnaire d'événement.

Initialisation des propriétés d'un composant

Les propriétés publiées peuvent être initialisées à la conception avec l'inspecteur d'objets ou, dans certains cas, avec des éditeurs de propriétés spécifiques.

Pour spécifier des propriétés à l'exécution, il suffit de leur affecter de nouvelles valeurs dans le code de l'application.

Pour des informations sur les propriétés de chaque composant, consultez l'aide en ligne.

Utilisation de l'inspecteur d'objets

Quand vous sélectionnez un composant d'une fiche, l'inspecteur d'objets affiche ses propriétés publiées et vous permet (si c'est approprié) de les modifier. Utilisez la touche *Tab* pour vous déplacer entre la colonne des valeurs et la colonne des propriétés. Si le curseur est dans la colonne des propriétés, vous pouvez vous positionner sur une propriété en tapant les premières lettres de son nom. Pour les propriétés de type booléen ou énuméré, vous pouvez choisir une valeur dans une liste déroulante ou parcourir les valeurs en double-cliquant dans la colonne des valeurs.

Si un symbole plus (+) apparaît à côté du nom de la propriété, vous pouvez faire apparaître une liste de sous-valeurs pour la propriété en cliquant sur le symbole plus ou en tapant '+'. Si un symbole moins (-) apparaît à côté du nom de la propriété, vous pouvez faire apparaître une liste de sous-valeurs pour la propriété en cliquant sur le symbole moins ou en tapant '-'.

Par défaut, les propriétés de la catégorie Héritage ne sont pas affichées. Pour modifier les filtres d'affichage, cliquez avec le bouton droit de la souris dans l'inspecteur d'objets et choisissez Voir. Pour davantage d'informations, voir "catégories de propriétés" dans l'aide en ligne.

Si plusieurs composants sont sélectionnés, l'inspecteur d'objets affiche toutes les propriétés, sauf *Name*, communes aux composants sélectionnés. Si la valeur d'une propriété partagée n'est pas la même pour tous les composants sélectionnés, l'inspecteur d'objets affiche soit la valeur par défaut, soit la valeur de la propriété pour le premier composant sélectionné. Quand vous modifiez une propriété partagée, la modification s'applique à tous les composants sélectionnés.

Utilisation des éditeurs de propriété

Certaines propriétés, comme *Font* utilisent des éditeurs de propriétés spécifiques. Quand une telle propriété est sélectionnée dans l'inspecteur d'objets, un bouton points de suspension (...) apparaît à côté de sa valeur. Pour ouvrir l'éditeur de propriété, double-cliquez dans la colonne des valeurs, cliquez sur le bouton points de suspension ou tapez *Ctrl+Entrée* quand la focalisation se trouve sur la propriété ou sur sa valeur. Pour certains composants, il suffit de double-cliquer sur le composant dans la fiche pour ouvrir un éditeur de propriété.

Les éditeurs de propriété permettent de définir des propriétés complexes à partir d'une seule boîte de dialogue. Elles valident les saisies et permettent souvent de prévisualiser les effets d'une affectation.

Initialisation des propriétés à l'exécution

Vous pouvez à l'exécution utiliser votre code source pour affecter une valeur à toute propriété accessible en écriture. Vous pouvez ainsi, définir de manière dynamique le libellé d'une fiche :

```
Form1.Caption := MaChaine;
```

Appel de méthodes

Une méthode s'appelle comme une procédure ou une fonction ordinaire. Par exemple, les contrôles visuels disposent de la méthode *Repaint* qui rafraîchit l'image du contrôle à l'écran. Vous pouvez appeler la méthode *Repaint* d'un objet grille de dessin de la manière suivante :

```
DrawGrid1.Repaint;
```

Comme pour les propriétés, c'est la portée d'une méthode qui impose ou pas l'utilisation de qualificateurs. Par exemple, pour redessiner une fiche depuis le gestionnaire d'événement de l'un des contrôles enfant de la fiche, il n'est pas nécessaire de préfixer l'appel de méthode avec le nom de la fiche :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Repaint;
end;
```

Pour davantage d'informations sur la portée, voir "Portée et qualificateurs" à la page 3-9.

Utilisation des événements et des gestionnaires d'événements

Dans Delphi, l'essentiel du code que vous écrivez est exécuté, directement ou indirectement, en réponse à des *événements*. Un événement est un type particulier de propriété qui représente une situation à l'exécution, généralement une action de l'utilisateur. Le code qui répond directement à un événement, ce qu'on appelle un *gestionnaire d'événement*, est une procédure Pascal Objet. Les sections suivantes expliquent comment :

- Générer un nouveau gestionnaire d'événement
- Générer le gestionnaire de l'événement par défaut d'un composant
- Rechercher un gestionnaire d'événement
- Associer un événement à un gestionnaire d'événement existant
- Associer des événements de menu à des gestionnaires d'événements
- Supprimer un gestionnaire d'événement

Génération d'un nouveau gestionnaire d'événement

Delphi peut créer le squelette de gestionnaires d'événements pour les fiches et les autres composants. Pour créer un gestionnaire d'événement,

1 Sélectionnez un composant.

2 Cliquez dans la page Evénements de l'inspecteur d'objets. La page événement de l'inspecteur d'objets affiche tous les événements définis pour le composant sélectionné.

3 Sélectionnez l'événement de votre choix puis double-cliquez dans la colonne valeur ou appuyez sur *Ctrl+Entrée*. Delphi génère le gestionnaire d'événement dans l'éditeur de code et place le curseur dans le bloc **begin...end**.

4 A l'intérieur du bloc **begin...end**, entrez le code que vous voulez exécuter lorsque l'événement se produit.

Génération du gestionnaire de l'événement par défaut d'un composant

Certains composants ont un événement *par défaut*, celui que le composant a le plus souvent besoin de gérer. Par exemple, l'événement par défaut d'un bouton est *OnClick*. Pour créer un gestionnaire de l'événement par défaut, double-cliquez sur le composant dans le concepteur de fiche, cela génère le squelette de la procédure de gestion de l'événement et ouvre l'éditeur de code en plaçant le curseur à l'intérieur du corps de la procédure où il ne vous reste plus qu'à ajouter du code.

Certains composants n'ont pas d'événement par défaut. D'autres, comme le biseau (*TBevel*), n'ont pas du tout d'événement. D'autres composants encore peuvent réagir différemment si vous double-cliquez dessus dans le concepteur de fiche. Par exemple, plusieurs composants ouvrent un éditeur de propriété par défaut ou une autre boîte de dialogue quand on double-clique dessus à la conception.

Recherche de gestionnaires d'événements

Si vous avez généré le gestionnaire de l'événement par défaut d'un composant en double-cliquant dessus dans le concepteur de fiche, vous pouvez revenir dessus en recommençant. Double-cliquez sur le composant ; l'éditeur de code s'ouvre, le curseur positionné sur le début du corps du gestionnaire d'événement.

Pour rechercher le gestionnaire d'un événement qui n'est pas l'événement par défaut,

1 Dans la fiche, sélectionnez le composant dont vous recherchez le gestionnaire d'événement.

2 Dans l'inspecteur d'objets, cliquez sur l'onglet Evénements.

3 Sélectionnez l'événement dont vous recherchez le gestionnaire et double-cliquez dans la colonne des valeurs. L'éditeur de code s'ouvre, le curseur positionné sur le début du corps du gestionnaire d'événement.

Association d'un événement à un gestionnaire d'événement existant

Vous pouvez réutiliser le code en écrivant des gestionnaires d'événements qui gèrent plusieurs événements de composants. Par exemple, de nombreuses applications proposent des turboboutons qui sont l'équivalent de commandes de la barre des menus. Quand un bouton initie la même action qu'une commande

de menu, vous pouvez écrire un seul gestionnaire d'événement et l'affecter à l'événement *OnClick* du bouton et de l'élément de menu.

Pour associer un événement à un gestionnaire d'événement existant,

1. Dans la fiche, sélectionnez le composant dont vous voulez gérer un événement.
2. Dans la page Evénements de l'inspecteur d'objets, sélectionnez l'événement auquel vous voulez attacher un gestionnaire.
3. Cliquez sur le bouton flèche vers le bas à côté de l'événement afin d'ouvrir une liste des gestionnaires d'événements existants. La liste ne propose que les gestionnaires d'événements écrits pour des événements portant le même nom dans la même fiche. Sélectionnez dans la liste en cliquant sur un nom de gestionnaire d'événement.

Cette manière de procéder est un moyen simple de réutiliser des gestionnaires d'événements. Cependant, les *listes d'actions* et dans la VCL, les *bandes d'actions*constituent un outil plus puissant permettant de centraliser l'organisation du code répondant à des commandes de l'utilisateur. Les listes d'actions peuvent être utilisées dans les applications multiplates-formes, alors que les bandes d'actions ne le peuvent pas. Pour plus d'informations sur les listes d'actions et sur les bandes d'actions, voir "Organisation des actions pour les barres d'outils et les menus" à la page 6-18.

Utilisation du paramètre Sender

Dans un gestionnaire d'événement, le paramètre *Sender* indique le composant qui a reçu l'événement et qui a donc appelé le gestionnaire. Il est parfois pratique de partager entre plusieurs composants un même gestionnaire d'événement qui doit se comporter différemment selon le composant qui l'a appelé. Vous pouvez y arriver en utilisant le paramètre *Sender* dans une instruction **if...then...else**. Par exemple, le code suivant affiche le nom de l'application dans le titre d'une boîte de dialogue uniquement si l'événement *OnClick* a été reçu par *Button1*.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
if Sender = Button1 then
  AboutBox.Caption := 'About ' + Application.Title
else
  AboutBox.Caption := '';
AboutBox.ShowModal;
end;
```

Affichage et codage d'événements partagés

Si des composants partagent des événements, vous pouvez afficher leurs événements partagés dans l'inspecteur d'objets. Commencez par sélectionner les composants en maintenant enfoncée la touche *Maj* et en cliquant dessus dans le concepteur de fiche ; puis, choisissez l'onglet Evénements de l'inspecteur d'objets. Dans la colonne des valeurs de l'inspecteur d'objets, vous pouvez alors créer un nouveau gestionnaire d'événement ou affecter un gestionnaire d'événement existant aux événements partagés.

Association d'événements de menu à des gestionnaires d'événements

Le concepteur de menus, utilisé pour les composants *MainMenu* et *PopupMenu*, vous permet de spécifier simplement des menus déroulants ou surgissants dans une application. Pour que les menus fonctionnent, il faut néanmoins que chaque élément de menu réponde à l'événement *OnClick* qui se produit à chaque fois que l'utilisateur choisit l'élément de menu ou appuie sur sa touche de raccourci. Cette section explique comment associer des gestionnaires d'événements aux éléments de menu. Pour plus d'informations sur le concepteur de menus et les composants associés, voir "Création et gestion de menus" à la page 6-33.

Pour créer un gestionnaire d'événement pour un élément de menu,

1. Ouvrez le concepteur de menus en double-cliquant sur un objet *MainMenu* ou *PopupMenu*.
2. Sélectionnez un élément de menu dans le concepteur de menus. Dans l'inspecteur d'objets, vérifiez qu'une valeur est attribuée à la propriété *Name* de l'élément.
3. Dans le concepteur de menus, double-cliquez sur l'élément de menu. Delphi génère un gestionnaire d'événement dans l'éditeur de code et place le curseur dans le bloc **begin...end**.
4. A l'intérieur du bloc **begin...end**, entrez le code que vous voulez exécuter lorsque l'utilisateur sélectionne la commande de menu.

Pour associer un élément de menu à un gestionnaire d'événement *OnClick* existant *:*

1. Ouvrez le concepteur de menus en double-cliquant sur un objet *MainMenu* ou *PopupMenu*.
2. Sélectionnez un élément de menu dans le concepteur de menus. Dans l'inspecteur d'objets, vérifiez qu'une valeur est attribuée à la propriété *Name* de l'élément.
3. Dans la page Evénements de l'inspecteur d'objets, cliquez sur le bouton flèche vers le bas à côté de *OnClick* afin d'ouvrir une liste des gestionnaires d'événements existants. La liste ne propose que les gestionnaires d'événements écrits pour des événements *OnClick* dans la fiche. Sélectionnez un gestionnaire d'événement dans la liste en cliquant sur son nom.

Suppression de gestionnaires d'événements

Quand vous supprimez un composant d'une fiche en utilisant le concepteur de fiche, Delphi retire le composant de la déclaration de type de la fiche. Mais, il ne supprime pas les méthodes associées car elles peuvent être appelées par d'autres composants de la fiche. Vous pouvez supprimer manuellement une méthode (comme un gestionnaire d'événement) mais si vous le faites, vous devez supprimer la déclaration avancée de la méthode (dans la section **interface** de l'unité) et son implémentation (dans la section **implementation**) ; sinon, vous obtiendrez une erreur de compilation lors de la génération du projet.

Composants VCL et CLX

La palette de composants contient une sélection de composants qui gèrent une grande variété d'opérations de programmation. Vous pouvez ajouter, retirer ou réorganiser les composants de la palette et vous pouvez créer des *modèles* de composants et des *cadres* qui regroupent plusieurs composants.

Les composants ayant des fonctions similaires sont regroupés en pages dans la palette de composants. Les pages apparaissant dans la configuration par défaut dépendent de la version de Delphi que vous utilisez. Le tableau suivant énumère les pages définies par défaut et les composants disponibles pour la création des applications. Certaines des pages et des composants ne sont pas multiplates-formes, comme le tableau le souligne. Vous pouvez utiliser certains composants non visuels spécifiques à la VCL dans les applications CLX uniquement Windows, cependant, les applications ne seront pas multiplates-formes sauf si vous isolez ces parties du code.

Tableau 3.3 Pages de la palette de composants

Nom de la page	Description	multiplate-forme ?
Standard	Contrôles standard, menus	Oui
Supplément	Contrôles spécialisés	Oui sauf ApplicationEvents et CustomizeDlg
Win32	contrôles Windows courants	De nombreux composants identiques figurent sur la page Contrôles communs apparaissant à la place lors de la création des applications CLX ; RichEdit, UpDown, HotKey, Animate, DataTimePicker, MonthCalendar, Coolbar, PageScroller et ComboBoxEx ne sont pas multiplates-formes.
Système	Composants et contrôles permettant un accès au niveau du système, y compris les timers, le multimédia et le DDE.	Timer l'est mais pas PaintBox, MediaPlayer, OleContainer ni les composants Dde
AccèsBD	Composants pour le travail avec les données des bases de données qui ne sont pas liées à un mécanisme d'accès aux données particulier	Oui
ContrôleBD	Contrôles visuels orientés données	Oui sauf pour DBRichEdit, DBCtrlGrid et DBChart
dbExpress	Contrôles de bases de données qui utilisent dbExpress, une couche multiplate-forme indépendante des bases de données qui fournit des méthodes pour le traitement SQL dynamique. Elle définit une interface commune permettant d'accéder aux serveurs SQL.	Oui

Tableau 3.3 Pages de la palette de composants (suite)

Nom de la page	Description	multiplate-forme ?
DataSnap	Composants utilisés pour créer des applications de bases de données multiniveaux.	Non mais peuvent être utilisés dans des applications CLX Windows
BDE	Composants qui fournissent l'accès aux données via le Borland Database Engine	Non mais peuvent être utilisés dans des applications CLX Windows
ADO	Composants permettant d'accéder aux données par le biais du modèle ADO.	Non mais peuvent être utilisés dans des applications CLX Windows
InterBase	Composants fournissant un accès direct à InterBase	Oui
InternetExpress	Composants qui sont simultanément une application serveur web et le client d'une application de base de données multiniveaux	Non mais peuvent être utilisés dans des applications CLX Windows
Internet	Composants pour les protocoles de communication Internet et les applications web	Oui sauf pour ClientSocket, ServerSocket, QueryTableProducer, XMLDoc et WebBrowser
WebSnap	Composants pour la construction d'applications serveur Web	Non mais peuvent être utilisés dans des applications CLX Windows
FastNet	Contrôles Internet NetMasters	Non mais peuvent être utilisés dans des applications CLX Windows
QReport	Composants QuickReport utilisés pour créer des états incorporés.	Non mais peuvent être utilisés dans des applications CLX Windows
Dialogues	Boîte de dialogue les plus utilisées	Oui sauf pour OpenPictureDialog, SavePictureDialog, PrinterSetup-Dialog et PageSetupDialog
Win 3.1	ancien style des composants Win 3.1	Non
Exemples	Composants personnalisés exemple	Non
ActiveX	Exemples de contrôles ActiveX ; voir la documentation Microsoft (msdn.microsoft.com)	Non
COM+	Composant pour la gestion des événements COM+	Non mais peuvent être utilisés dans des applications CLX Windows
WebServices	Composants pour écrire des applications qui implémentent ou utilisent des services web basés sur SOAP	Non mais peuvent être utilisés dans des applications CLX Windows
Serveurs	Exemples serveur COM pour Microsoft Excel, Word, etc. (voir la documentation MSDN Microsoft)	Non mais peuvent être utilisés dans des applications CLX Windows
Indy - Clients	Composants Internet multiplates-formes pour le client (composants Internet Winshoes à code source libre)	Oui

Tableau 3.3 Pages de la palette de composants (suite)

Nom de la page	Description	multiplate-forme ?
Indy - Serveurs	Composants Internet multiplates-formes pour le serveur (composants Internet Winshoes à code source libre)	Oui
Indy - Divers	Composants Internet multiplates-formes supplémentaires (composants Internet Winshoes à code source libre)	Oui

L'aide en ligne propose des informations sur les composants de la palette. Cependant, certains des composants des pages ActiveX, Serveurs et Exemples sont proposés uniquement à titre d'exemple et ne sont pas documentés.

Ajout de composants personnalisés à la palette de composants

Vous pouvez installer des composants personnalisés — conçus par vous ou acquis séparément — dans la palette de composants et les utiliser dans vos applications. Pour écrire un composant, voir la partie V, "Création de composants personnalisés". Pour installer un composant existant, voir "Installation de paquets de composants" à la page 11-6.

Contrôles texte

De nombreuses applications présentent du texte à l'utilisateur ou lui permet d'en saisir. Le type de contrôle à employer pour contenir les informations dépend de la taille et du format des informations.

Utilisez ce composant :	Quand l'utilisateur doit :
TEdit	Modifier une seule ligne de texte.
TMemo	Modifier plusieurs lignes de texte.
TMaskEdit	Utiliser un format particulier, par exemple celui d'un code postal ou d'un numéro de téléphone.
TRichEdit	Modifier plusieurs lignes de texte en utilisant du texte mis en forme (VCL seulement).

TEdit et *TMaskEdit* sont de simples contrôles texte comprenant une boîte texte d'une ligne dans laquelle vous pouvez entrer des informations. Quand la boîte texte détient la focalisation, un point d'insertion clignotant apparaît.

Vous pouvez inclure du texte dans la boîte en donnant une valeur chaîne à sa propriété *Text*. Vous contrôlez l'apparence du texte dans la boîte en donnant des valeurs à sa propriété *Font*. Vous pouvez spécifier la police, la taille, la couleur et des attributs de fonte. Ces attributs affectent tout le texte de la boîte et ne peuvent s'appliquer individuellement à chacun des caractères.

Une boîte texte peut être conçue pour changer de taille en fonction de la taille de la police qu'elle contient. Vous faites cela en définissant la propriété *AutoSize* par True. Vous pouvez limiter le nombre de caractères que peut contenir une boîte de texte en attribuant une valeur à la propriété *MaxLength*.

TMaskEdit est un contrôle d'édition spécial qui valide le texte entré par le biais d'un masque indiquant les formats corrects du texte. Le masque peut également formater le texte affiché à l'utilisateur.

TMemo permet d'ajouter plusieurs lignes de texte.

Propriétés des contrôles texte

Voici quelques propriétés importantes des contrôles texte :

Tableau 3.4 Propriétés des contrôles texte

Propriété	Description
Text	Détermine le texte qui apparaît dans la boîte de saisie ou le contrôle mémo.
Font	Contrôle les attributs du texte écrit dans le contrôle boîte texte ou mémo.
AutoSize	Permet à la hauteur de la boîte texte de changer de façon dynamique selon la font sélectionnée.
ReadOnly	Spécifie si l'utilisateur est autorisé à modifier le texte.
MaxLength	Limite le nombre de caractères d'un contrôle texte.

Propriétés communes aux contrôles mémo et de texte formaté

Les contrôles mémo et de texte formaté, qui gèrent plusieurs lignes de texte, ont plusieurs propriétés en commun. Veuillez noter que les contrôles de texte formaté ne sont pas multiplates-formes.

TMemo est un autre type de boîte texte, contenant plusieurs lignes de texte. Les lignes d'un contrôle mémo peuvent s'étendre au-delà de la marge droite de la boîte texte ou aller à la ligne automatiquement. Vous décidez du retour à la ligne à l'aide de la propriété *WordWrap*.

Les contrôles mémo et de texte formaté possèdent d'autres propriétés, dont :

- *Alignment* spécifie comment le texte est aligné (gauche, droite ou centré) à l'intérieur du composant.
- La propriété *Text* contient le texte du contrôle. Votre application peut déterminer si le texte a été modifié en examinant la propriété *Modified*.
- *Lines* contient le texte sous la forme d'une liste de chaînes.
- *OEMConvert* détermine si le texte du contrôle est converti en caractères OEM. Cela s'avère utile pour valider les noms de fichiers (VCL seulement).
- *WordWrap* détermine si le texte revient à la ligne après la marge droite.
- *WantReturns* détermine si l'utilisateur peut insérer des passages à la ligne dans le texte.
- *WantTabs* détermine si l'utilisateur peut insérer des tabulations dans le texte.
- *AutoSelect* détermine si le texte est automatiquement sélectionné (mis en évidence) quand le contrôle devient actif.

- *SelText* contient la partie du texte actuellement sélectionnée (mise en évidence).
- *SelStart* et *SelLength* indiquent la position des premier et dernier caractères de la partie sélectionnée du texte.

A l'exécution, vous pouvez sélectionner tout le texte d'un mémo en utilisant la méthode *SelectAll*.

Contrôles de texte formaté (VCL seulement)

Le composant éditeur de texte formaté (*TRichEdit*) est un composant mémo qui gère le texte mis en forme, l'impression, la recherche et le glisser-déplacer du texte. Il vous permet de spécifier les propriétés de police, d'alignement, de tabulation, d'indentation et de numérotation.

Contrôles de saisies spécialisées

Les composants suivants proposent d'autres méthodes pour recevoir des saisies.

Utilisez ce composant :	Quand l'utilisateur doit :
TScrollBar	Sélectionner des valeurs dans un intervalle continu.
TTrackBar	Sélectionner des valeurs dans un intervalle continu (visuellement plus parlant qu'une barre de défilement).
TUpDown	Sélectionner une valeur à l'aide d'un incrémenteur associé à un composant de saisie (VCL seulement)
THotKey	Entrer des séquences clavier *Ctrl*/*Maj*/*Alt* (VCL seulement).
TSpinEdit	Sélectionner une valeur à l'aide d'un widget incrémenteur (CLX seulement).

Barres de défilement

Le composant barre de défilement crée une barre de défilement utilisée pour faire défiler le contenu d'une fenêtre, d'une fiche ou d'un autre contrôle. Le code écrit dans le gestionnaire d'événement *OnScroll* détermine comment le contrôle se comporte quand l'utilisateur fait défiler la barre de défilement.

Le composant barre de défilement est rarement utilisé car la plupart des composants visuels disposent de leurs propres barres de défilement sans nécessiter de programmation. Par exemple, *TForm* propose les propriétés *VertScrollBar* et *HorzScrollBar* qui configurent automatiquement des barres de défilement pour la fiche. Pour créer une région défilante dans une fiche, utilisez *TScrollBox*.

Barres graduées

Une barre graduée peut définir des valeurs entières dans un intervalle continu. Elle sert à ajuster des propriétés telle qu'une coleur, un volume ou une luminosité. L'utilisateur déplace la glissière en la faisant glisser à une position donnée ou en cliquant dans la barre.

- Utilisez les propriétés *Max* et *Min* pour définir les bornes supérieure et inférieure de l'intervalle de la barre graduée.
- Utilisez *SelEnd* et *SelStart* pour mettre en évidence un intervalle sélectionné. Voir la figure 3.4.
- La propriété *Orientation* détermine si la barre graduée est verticale ou horizontale.
- Par défaut, une barre graduée dispose d'une ligne de graduations en bas. Utilisez la propriété *TickMarks* pour modifier leur emplacement. Pour contrôler l'espacement des graduations, utilisez la propriété *TickStyle* et la méthode *SetTick*.

Figure 3.4 Trois vues du composant barre graduée

- *Position* définit une position par défaut dans la barre graduée et indique à l'exécution la valeur sélectionnée par l'utilisateur.
- Par défaut, l'utilisateur peut se déplacer d'une graduation vers le haut ou vers le bas en utilisant les touches de déplacement correspondantes. Affectez *LineSize* pour changer cet incrément.
- Affectez *PageSize* pour déterminer le nombre de graduations du déplacement quand l'utilisateur appuie sur les touches *Pg. Haut* et *Pg. Bas*.

Contrôles flèches haut-bas (VCL seulement)

Un contrôle flèches haut-bas (*TUpDown*) est constitué d'une paire de boutons fléchés qui permettent à l'utilisateur de modifier une valeur entière d'un incrément fixe. La valeur en cours est donnée par la propriété *Position* ; l'incrément, qui vaut 1 par défaut, est spécifié par la propriété *Increment*. Utilisez la propriété *Associate* pour associer un autre composant (comme un contrôle d'édition) au contrôle haut-bas.

Contrôles incrémenteur (CLX seulement)

Un contrôle incrémenteur (*TSpinEdit*) est également appelé widget haut-bas, comme le widget flèches ou le bouton incrémenteur. Ce contrôle permet à l'utilisateur de l'application de changer une valeur entière par incréments fixes, soit en cliquant sur les boutons fléchés haut ou bas pour augmenter ou diminuer la valeur affichée, soit en tapant directement la valeur dans la boîte de l'incrémenteur.

La valeur en cours est donnée par la propriété *Value* ; l'incrément, qui vaut 1 par défaut, est spécifié par la propriété *Increment*.

Contrôles touche d'accès rapide (VCL seulement)

Utilisez le composant touche d'accès rapide (*THotKey*) pour affecter une séquence de touches qui transfère la focalisation à un composant. La propriété *HotKey* contient la combinaison de touches en cours et la propriété *Modifiers* détermine les touches disponibles pour *HotKey*.

Le composant raccourci clavier peut être affecté à la propriété *ShortCut* d'un élément de menu. Ensuite, lorsqu'un utilisateur saisit la combinaison de touches spécifiée par les propriétés *HotKey* et *Modifiers*, Windows active l'élément de menu.

Contrôles séparateur

Un séparateur (*TSplitter*) placé entre deux contrôles alignés permet aux utilisateurs de redimensionner les contrôles. Utilisés avec des composants comme les volets ou les boîtes de groupe, les séparateurs vous permettent de décomposer une fiche en plusieurs volets contenant chacun plusieurs contrôles.

Après avoir placé un volet ou un autre contrôle dans une fiche, ajoutez un séparateur ayant le même alignement que le contrôle. Le dernier contrôle doit être aligné sur le client afin qu'il remplisse tout l'espace restant quand les autres sont redimensionnés. Vous pouvez, par exemple, placer un volet sur le bord gauche d'une fiche, initialiser sa propriété *Alignment* par *alLeft*, puis placer un séparateur (ayant également l'alignement *alLeft*) à droite du volet, et enfin placer un autre volet (avec l'alignement *alLeft* ou *alClient*) à droite du séparateur.

Initialisez *MinSize* afin de spécifier la taille minimum que le séparateur doit laisser quand il redimensionne le contrôle adjacent. Initialisez *Beveled* à *True* pour donner au séparateur un aspect 3D.

Boutons et contrôles similaires

En dehors des menus, les boutons constituent le moyen le plus simple de déclencher une commande dans une application. Delphi propose plusieurs contrôles de type bouton :

Utilisez ce composant :	Pour :
TButton	Présenter des choix de commandes avec du texte dans des boutons
TBitBtn	Présenter des choix de commandes dans des boutons contenant du texte et des glyphes
TSpeedButton	Créer des groupes de boutons dans les barres d'outils
TCheckBox	Présenter des options de type Oui/Non
TRadioButton	Présenter un ensemble de choix mutuellement exclusifs
TToolBar	Disposer des boutons et d'autres contrôles en ligne et ajuster automatiquement leur taille et leur position
TCoolBar	Afficher une collection de contrôles fenêtrés dans des bandes déplaçables et redimensionnables *(VCL seulement)*

Contrôles bouton

Les utilisateurs cliquent sur les contrôles bouton pour initier des actions. Les boutons sont libellés par du texte qui représente l'action. Vous spécifiez le texte en attribuant une valeur chaîne à la propriété *Caption*. Vous pouvez aussi sélectionner la plupart des boutons en appuyant sur une touche du clavier,

appelée raccourci clavier. Le raccourci est indiqué sur le bouton par une lettre soulignée.

Les utilisateurs cliquent sur les contrôles bouton pour initier des actions. Vous pouvez associer une action à un composant *TButton* en créant un gestionnaire d'événement *OnClick* correspondant. En double-cliquant sur un bouton à la conception, vous affichez le gestionnaire d'événement *OnClick* du bouton dans l'éditeur de code.

- Set Affectez la valeur *True* à la propriété *Cancel* pour que le bouton déclenche son événement *OnClick* quand l'utilisateur appuie sur ***Echap***.
- Affectez la valeur *True* à la propriété *Default* pour que la touche ***Entrée*** déclenche l'événement *OnClick* du bouton.

Boutons bitmap

Un bouton bitmap (*BitBtn*) est un contrôle bouton qui contient une image bitmap.

- Pour attribuer un bitmap personnalisé à votre bouton, affectez la propriété *Glyph*.
- Utilisez la propriété *Kind* pour configurer automatiquement un bouton avec un glyphe et un comportement par défaut.
- Par défaut, le glyphe est à gauche du texte. Pour le déplacer, utilisez la propriété *Layout*.
- Le glyphe et le texte sont automatiquement centrés dans le bouton. Pour changer leur position, utilisez la propriété *Margin*. *Margin* détermine le nombre de pixels entre le bord de l'image et le bord du bouton.
- Par défaut, l'image et le texte sont séparés par 4 pixels. Utilisez *Spacing* pour augmenter ou réduire cette distance.
- Les boutons bitmap peuvent avoir 3 états : haut, bas et enfoncé. Affectez la valeur 3 à la propriété *NumGlyphs* pour attribuer un bitmap différent à chaque état.

Turboboutons

Les turboboutons, qui affichent généralement une image, peuvent fonctionner en groupe. Ils sont souvent utilisés avec des volets pour créer des barres d'outils.

- Pour faire fonctionner des turboboutons en groupe, affectez à la propriété *GroupIndex* de tous les boutons la même valeur non-nulle.
- Par défaut, des turboboutons apparaissent à l'état haut (non sélectionné). Pour afficher un turbobouton à l'état sélectionné, affectez la valeur *True* à sa propriété *Down*.
- Si *AllowAllUp* a la valeur *True*, tous les turboboutons d'un groupe peuvent être non sélectionnés. Affectez la valeur *False* à *AllowAllUp* pour qu'un groupe de boutons se comporte comme un groupe de boutons radio.

Pour plus d'informations sur les turboboutons, reportez-vous aux sous-rubriques de la section "Ajout d'une barre d'outils en utilisant un composant volet" à la page 6-49.

Cases à cocher

Une case à cocher est une bascule qui permet à l'utilisateur de sélectionner un état activé ou désactivé. Quand l'option est activée, la case est cochée. Sinon, la case à cocher est vide. Vous créez des cases à cocher à l'aide de *TCheckBox*.

- Affectez *True* à *Checked* pour que la case soit cochée par défaut.
- Affectez *True* à *AllowGrayed* pour que la case à cocher puisse prendre trois états : cochée, non-cochée et grisée.
- La propriété *State* indique si la case est cochée (*cbChecked*), non cochée (*cbUnchecked*) ou grisée (*cbGrayed*).

Remarque Les contrôles case à cocher affichent un des deux états binaires. L'état indéterminé est utilisé quand les autres choix rendent impossible de déterminer la valeur en cours de la case à cocher.

Boutons radio

Les boutons radio proposent un ensemble de choix mutuellement exclusifs. Vous pouvez créer des boutons radio individuels à l'aide de *TRadioButton* ou utiliser le composant *groupe de boutons radio* (*TRadioGroup*) qui regroupe automatiquement des boutons radio. Cela permet à l'utilisateur de sélectionner une option dans un ensemble de choix limité. Voir "Regroupement de composants" à la page 3-44, pour plus d'informations.

Un bouton radio sélectionné s'affiche sous forme d'un cercle dont le centre est rempli. S'il n'est pas sélectionné, le bouton radio affiche un cercle vide. Donnez la valeur True ou False à la propriété Checked pour changer l'état visuel du bouton radio.

Barres d'outils

Les barres d'outils permettent aisément d'organiser et de gérer des contrôles visuels. Vous pouvez créer une barre d'outils à partir d'un composant volet et de turboboutons, ou utiliser le composant *ToolBar* puis choisir Nouveau bouton dans son menu contextuel pour chaque bouton à ajouter.

Le composant *TToolBar* présente plusieurs avantages : les boutons d'une barre d'outils ont automatiquement des dimensions et un espacement homogènes, les autres contrôles conservent leur position et hauteur relatives ; les contrôles peuvent automatiquement passer à la ligne s'il n'y a pas assez de place horizontalement. Le composant *TToolBar* propose également des options comme la transparence, les bordures en relief et les espaces et des séparations pour regrouper des contrôles.

Vous pouvez utiliser un ensemble d'actions regroupées sur des barres d'outils et des menus, en utilisant des *liste d'actions* ou *bandes d'actions*. Reportez-vous à "Utilisation des listes d'actions" à la page 6-26, pour savoir comment utiliser les listes d'actions avec boutons et barres d'outils.

Les barres d'outils peuvent aussi être parents d'autres contrôles, comme les boîtes de saisie, les boîtes à options, etc.

Barres multiples (VCL seulement)

Une barre multiple contient des contrôles enfant pouvant être déplacés et redimensionnés de manière indépendante. Chaque contrôle se trouve dans une bande indépendante. L'utilisateur positionne les contrôles en utilisant la poignée de redimensionnement à gauche de chaque bande.

A la conception et à l'exécution, la barre multiple exige une version 4.70, ou ultérieure, de COMCTL32.DLL (qui se trouve généralement dans le répertoire Windows\System ou Windows\System32). Les barres multiples ne peuvent pas être utilisées dans les applications multiplates-formes.

- La propriété *Bands* contient une collection d'objets *TCoolBand*. A la conception, vous pouvez ajouter, retirer ou modifier les bandes à l'aide de l'éditeur de bandes. Pour l'ouvrir, sélectionnez la propriété *Bands* dans l'inspecteur d'objets puis double-cliquez dans la colonne des valeurs à droite ou cliquez sur le bouton Points de suspension (...). Vous pouvez également créer des bandes en ajoutant de nouveaux contrôles fenêtrés de la palette.
- La propriété *FixedOrder* détermine si les utilisateurs peuvent réorganiser les bandes.
- La propriété *FixedSize* détermine si les bandes ont une hauteur uniforme.

Gestion de listes

Les listes proposent à l'utilisateur une collection d'éléments dans laquelle il peut choisir. Plusieurs composants affichent des listes :

Utilisez ce composant :	Pour afficher :
TListBox	Une liste de chaînes de texte
TCheckListBox	Une liste avec une case à cocher devant chaque élément
TComboBox	Une boîte de saisie avec une liste surgissante déroulante
TTreeView	Une liste hiérarchique
TListView	Une liste d'éléments (déplaçables) avec éventuellement des icônes, des en-têtes et des colonnes
TDateTimePicker	Une boîte liste permettant de saisir des dates ou des heures (VCL seulement)
TMonthCalendar	Un calendrier permettant de sélectionner des dates (VCL seulement)

Utilisez les composants non-visuels *TStringList* et *TImageList* pour gérer des ensembles de chaînes ou d'images. Pour plus d'informations sur les listes de chaînes, voir "Utilisation des listes de chaînes" à la page 3-54.

Boîtes liste et boîtes liste de cases à cocher

Les boîtes liste (*TListBox*) et les boîtes liste de cases à cocher affichent une liste dans laquelle l'utilisateur peut sélectionner des éléments.

- *Items* utilise un objet *TStrings* pour remplir le contrôle avec des valeurs.
- *ItemIndex* indique l'élément sélectionné dans la liste.

- *MultiSelect* spécifie si l'utilisateur peut sélectionner plusieurs éléments à la fois.
- *Sorted* détermine si la liste est triée alphabétiquement.
- *Columns* spécifie le nombre de colonnes dans le contrôle liste.
- *IntegralHeight* spécifie si la boîte liste n'affiche que des entrées affichées en entier verticalement (VCL seulement).
- *ItemHeight* spécifie la hauteur, exprimée en pixels, de chaque élément de la liste. La propriété *Style* peut neutraliser l'effet de *ItemHeight*.
- La propriété *Style* détermine comment une boîte liste affiche ses éléments. Par défaut, les éléments sont affichés sous la forme d'une chaîne. En modifiant la valeur de *Style*, vous pouvez créer des boîtes liste *dessinées par le propriétaire*, dans ce cas les éléments peuvent être graphiques et de hauteur fixe ou de hauteur variable. Pour plus d'informations sur les contrôles dessinées par le propriétaire, voir "Ajout de graphiques à des contrôles" à la page 7-13.

Pour créer une boîte liste simple,

1. Dans le projet, faites glisser un composant boîte liste sur une fiche depuis la palette de composants.
2. Redimensionnez la boîte liste et définissez son alignement, si nécessaire.
3. Double-cliquez sur la partie droite de la propriété *Items* ou choisissez le bouton Points de suspension pour afficher l'éditeur de liste de chaînes.
4. Utilisez l'éditeur pour entrer des lignes de texte libre comme contenu de la boîte liste.
5. Puis, choisissez OK.

Pour permettre aux utilisateurs de sélectionner plusieurs éléments de la liste, utilisez les propriétés *ExtendedSelect* et *MultiSelect*.

Boîtes à options

Une boîte à options (*TComboBox*) combine une boîte de saisie et une liste déroulante. Quand les utilisateurs saisissent des données, en entrant du texte dans la boîte de saisie ou en sélectionnant un élément de la liste, la valeur de la propriété *Text* change. Si *AutoComplete* est activée, l'application recherche et affiche la correspondance la plus proche dans la liste au fur et à mesure que l'utilisateur tape des données.

Les trois types de boîtes à options sont : standard, déroulante (par défaut) et liste déroulante.

- Utilisez la propriété *Style* pour spécifier le type de boîte à options que vous souhaitez.
- Utilisez *csDropDown* si vous voulez une boîte de saisie avec une liste déroulante. Utilisez *csDropDownList* pour que la boîte de saisie soit en lecture seule (ce qui oblige les utilisateurs à sélectionner dans la liste). Initialisez la propriété *DropDownCount* pour changer le nombre d'éléments affichés dans la liste.
- Utilisez *csSimple* pour créer une boîte à options avec une liste fixe qui reste toujours ouverte. Prenez soin de redimensionner la boîte à options pour que les éléments de la liste soient affichés.

- Utilisez *csOwnerDrawFixed* ou *csOwnerDrawVariable* pour créer des boîtes à options *dessinées par le propriétaire* qui affichent des éléments graphiques ou de hauteur variable. Pour plus d'informations sur les contrôles dessinées par le propriétaire, voir "Ajout de graphiques à des contrôles" à la page 7-13.

Pendant l'exécution, les boîtes à options CLX fonctionnent différemment des boîtes à options VCL. Dans CLX (mais pas dans la boîte à options VCL), vous pouvez ajouter un élément à une liste déroulante en entrant du texte et en appuyant sur Entrée dans le champ d'édition d'une boîte à options. Vous pouvez désactiver cette fonctionnalité en définissant *InsertMode* par ciNone. Il est également possible d'ajouter des éléments vides (sans chaîne) à la liste de la boîte à options. De plus, si vous maintenez la flèche bas enfoncée, vous ne vous arrêtez pas au dernier élément de la liste. Vous refaites un tour en recommençant au début.

Vues arborescentes

Une vue arborescente *(TTreeView)* affiche des éléments dans une table des matières indentée. Le contrôle propose des boutons qui permettent de développer ou de réduire les nœuds. Vous pouvez inclure des icônes en plus du libellé des éléments et afficher différentes icônes pour indiquer si un nœud est développé ou réduit. Vous pouvez également inclure des éléments graphiques, par exemple des cases à cocher, afin de refléter des informations sur l'état des éléments.

- *Indent* définit le nombre de pixels séparant horizontalement les éléments de leurs parents.
- *ShowButtons* active l'affichage des boutons '+' et '–' pour indiquer si un élément peut être développé.
- *ShowLines* active l'affichage de lignes de connexion qui montrent les relations hiérarchiques (VCL seulement).
- *ShowRoot* détermine si des lignes connectent les éléments racine (VCL seulement).

Pour lui ajouter des éléments au cours de la conception, double-cliquez sur le contrôle vue arborescente afin d'afficher l'éditeur d'éléments TreeView. Les éléments ajoutés deviennent la valeur de la propriété *Items*. Vous pouvez modifier les éléments pendant l'exécution en utilisant les méthodes de la propriété *Items*, qui est un objet de type *TTreeNodes*. *TTreeNodes* possède des méthodes pour ajouter des éléments, supprimer des éléments et naviguer entre les éléments dans la vue arborescente.

Les vues arborescentes peuvent afficher des colonnes et des sous-éléments semblables aux vues liste en mode vsReport.

Vues liste

Les vues liste, créées à l'aide de *TListView*, affichent des listes dans divers formats. Utilisez la propriété *ViewStyle* pour choisir le type de liste utilisé :

- *vsIcon* et *vsSmallIcon* affichent chaque élément sous la forme d'une icône avec un libellé. Les utilisateurs peuvent faire glisser les éléments dans la fenêtre de la vue liste (VCL seulement).

- *vsList* affiche les éléments comme icônes libellées qui ne peuvent pas être déplacées.
- *vsReport* affichent les éléments à raison d'un par ligne avec des informations organisées en colonnes. La colonne de gauche contient une petite icône et un libellé et les autres colonnes contiennent des sous-éléments spécifiés par l'application. Utilisez la propriété *ShowColumnHeaders* afin d'afficher des en-têtes de colonne.

Sélecteurs Date/Heure et calendriers mensuels (VCL seulement)

Le composant sélecteur date/heure affiche une boîte liste permettant de saisir des dates ou des heures. Le composant calendrier mensuel propose un calendrier permettant de saisir des dates ou des plages de dates. Pour utiliser ces composants, que ce soit à la conception ou à l'exécution, vous devez avoir la version 4.70, ou une version ultérieure, de COMCTL32.DLL (normalement dans le répertoire Windows\System ou Windows\System32). Ils ne peuvent pas être utilisés dans les applications multiplates-formes.

Regroupement de composants

Une interface utilisateur graphique est plus facile à utiliser quand des contrôles et les contrôles associés sont présentés dans des groupes. Delphi propose plusieurs composants permettant de regrouper des composants :

Utilisez ce composant :	Pour :
TGroupBox	Une boîte groupe standard avec un titre
TRadioGroup	Un groupe simple de boutons radio
TPanel	Un groupe de contrôles plus flexible visuellement
TScrollBox	Une zone défilante contenant des contrôles
TTabControl	Un ensemble d'onglets (du type classeur) mutuellement exclusifs
TPageControl	Un ensemble d'onglets (du type classeur) mutuellement exclusifs avec les pages correspondantes, chacune pouvant contenir d'autres contrôles
THeaderControl	Des en-têtes de colonne redimensionnables

Boîtes groupe et groupes de boutons radio

Une boîte groupe (*TGroupBox*) associe des contrôles d'une fiche. Les contrôles les plus fréquemment regroupés sont les boutons radio. Après avoir placé une boîte groupe dans une fiche, sélectionnez les composants dans la palette de composants et placez-les dans la boîte groupe. La propriété *Caption* contient le texte qui sert à libeller la boîte groupe à l'exécution.

Le composant groupe de boutons radio (*TRadioGroup*) simplifie le regroupement de boutons radio et gère leur fonctionnement en commun. Pour ajouter des boutons radio à un groupe, modifiez la propriété *Items* dans l'inspecteur d'objets ; chaque chaîne de *Items* constitue un bouton radio qui apparaît dans le groupe en utilisant la chaîne spécifiée comme libellé. La valeur de la propriété

ItemIndex détermine le bouton radio sélectionné. Affichez les boutons radio sur une ou plusieurs colonnes en définissant la valeur de la propriété *Columns*. Pour espacer les boutons, redimensionnez le composant groupe de boutons radio.

Volets

Le composant *TPanel* constitue un conteneur générique pour d'autres contrôles. Les volets sont généralement utilisés pour regrouper visuellement des composants sur une fiche. Il est possible d'aligner des volets dans la fiche pour conserver la même position relative quand la fiche est redimensionnée. La propriété *BorderWidth* détermine la largeur, en pixels, de la bordure entourant un volet.

Vous pouvez aussi placer d'autres contrôles sur un volet et utiliser la propriété *Align* pour positionner correctement tous les contrôles du groupe ou de la fiche. Vous pouvez choisir pour un volet un alignement alTop, pour que sa position soit maintenue même si la fiche est redimensionnée.

S vous voulez que le volet paraisse élevé ou enfoncé, utilisez les propriétés *BevelOuter* et *BevelInner*. Vous pouvez varier les valeurs de ces propriétés pour créer différents effets visuels 3D. Remarquez que si vous voulez seulement un biseau élevé ou enfoncé, il vaut mieux utiliser le contrôle *TBevel*, moins gourmand en ressources.

Vous pouvez aussi utiliser les volets pour construire des barres d'état ou des zones d'affichage d'information.

Boîtes de défilement

Les boîtes de défilement (*TScrollBox*) permettent de créer des zones défilantes à l'intérieur d'une fiche. Souvent, les applications ont besoin d'afficher plus d'informations qu'il ne peut apparaître dans une zone particulière. Certains contrôles, comme les boîtes liste, les mémos ou les fiches mêmes, peuvent automatiquement faire défiler leur contenu.

Les boîtes de défilement s'utilisent aussi pour créer des zones de défilement (vues) multiples dans une fenêtre. Les vues sont fréquentes dans les traitements de texte, les tableurs et les applications de gestion. Les boîtes de défilement vous offrent davantage de souplesse en vous permettant de définir arbitrairement une zone défilante dans une fiche.

Comme les volets et les boîtes groupe, les boîtes de défilement contiennent d'autres contrôles, comme les objets *TButton* et *TCheckBox*. Mais, normalement une boîte de défilement est invisible. Si les contrôles qu'elle contient ne peuvent rentrer dans sa partie visible, la boîte de défilement affiche automatiquement des barres de défilement.

A l'aide d'une boîte de défilement, vous pouvez aussi empêcher le défilement dans certaines zones d'une fenêtre, par exemple dans une barre d'outils ou dans une barre d'état (composants *TPanel*). Pour empêcher le défilement dans une barre d'outils ou dans une barre d'état, cachez les barres de défilement, puis placez une boîte de défilement dans la zone client de la fenêtre, entre la barre d'outils et la barre d'état. Les barres de défilement associées à la boîte de

défilement sembleront appartenir à la fenêtre, mais vous pourrez seulement faire défiler la zone se trouvant à l'intérieur de la boîte de défilement.

Contrôles onglets

Le composant contrôle onglets (*TTabControl*) crée un ensemble d'onglets semblables aux séparateurs d'un classeur. Vous pouvez créer des onglets en modifiant la propriété *Tabs* à l'aide de l'inspecteur d'objets ; chaque chaîne de *Tabs* représente un onglet. Le contrôle onglets est un simple volet avec un seul ensemble de composants dedans. Pour changer l'aspect du contrôle quand les onglets sont sélectionnés, écrivez un gestionnaire d'événement *OnChange*. Pour créer une boîte de dialogue multipage, utilisez plutôt un contrôle pages.

Contrôles pages

Le composant contrôle pages (*TPageControl*) est un ensemble de pages utilisé pour constituer une boîte de dialogue multipage. Un contrôle pages affiche plusieurs pages les unes sur les autres, et ce sont des objets *TTabSheet*. Vous sélectionnez une page dans l'interface utilisateur en cliquant sur son onglet, en haut du contrôle.

Pour créer une nouvelle page dans un contrôle pages lors de la conception, cliquez avec le bouton droit de la souris sur le contrôle pages et choisissez Nouvelle page. A l'exécution, vous ajoutez de nouvelles pages en créant l'objet correspondant à la page et en définissant sa propriété *PageControl* :

```
NewTabSheet = TTabSheet.Create(PageControl1);
NewTabSheet.PageControl := PageControl1;
```

Pour accéder à la page active, utilisez la propriété *ActivePage*. Pour changer de page active, définissez la propriété *ActivePage* ou la propriété *ActivePageIndex*.

Contrôles en-têtes

Un contrôle en-têtes (*THeaderControl*) est un ensemble d'en-têtes de colonnes que l'utilisateur peut sélectionner ou redimensionner à l'exécution. Modifiez la propriété *Sections* du contrôle pour ajouter ou modifier les en-têtes. Vous pouvez placer les sections d'en-tête au-dessus des colonnes ou des champs. Par exemple, les sections d'en-tête peuvent être placées sur une boîte liste (*TListBox*).

Rétroaction visuelle

Il existe plusieurs moyens de donner à l'utilisateur des informations sur l'état d'une application. Par exemple, certains composants, dont *TForm*, disposent de la propriété *Caption* qui peut être définie à l'exécution. Vous pouvez également créer des boîtes de dialogue pour afficher des messages. De plus, les composants

suivants sont particulièrement utiles pour fournir des indications visuelles à l'exécution.

Utilisez ce composant ou cette propriété :	Pour :
TLabel et *TStaticText*	Afficher du texte non modifiable
TStatusBar	Afficher une zone d'état (généralement en bas d'une fenêtre)
TProgressBar	Afficher le pourcentage effectué d'une tâche donnée
Hint et *ShowHint*	Activer les conseils d'aide (appelés aussi bulles d'aide)
HelpContext et *HelpFile*	Effectuer la liaison avec le système d'aide en ligne

Libellés et composants texte statique

Les libellés (*TLabel*) affichent du texte, ils sont généralement placés à côté d'autres composants. Vous placez un libellé sur une fiche lorsque vous avez besoin d'identifier ou d'annoter un autre composant, comme une boîte de saisie, ou lorsque vous voulez inclure du texte dans la fiche. Le composant libellé standard, *TLabel*, est un contrôle non-fenêtré (dans CLX, non basé sur un widget), qui ne peut donc pas recevoir la focalisation ; si vous avez besoin d'un libellé disposant d'un handle de fenêtre, utilisez à la place *TStaticText*.

Les propriétés des libellés sont les suivantes :

- *Caption* contient la chaîne de texte du libellé.
- *Font*, *Color* et d'autres propriétés déterminent l'apparence du libellé. Chaque libellé ne peut utiliser qu'une seule police, taille et couleur.
- *FocusControl* relie le contrôle libellé à un autre contrôle de la fiche. Si *Caption* comporte une touche accélératrice, le contrôle spécifié dans la propriété *FocusControl* obtient la focalisation quand l'utilisateur appuie sur la touche de raccourci.
- *ShowAccelChar* détermine si le libellé peut afficher un caractère de raccourci souligné. Si *ShowAccelChar* a la valeur *True*, tout caractère précédé d'un & apparaît souligné et active une touche de raccourci.
- *Transparent* détermine si les éléments sur lesquels le libellé est placé (par exemple des images) sont visibles.

Les libellés contiennent généralement du texte statique en lecture seule, que l'utilisateur de l'application ne peut pas modifier. Vous pouvez modifier le texte lorsque l'application est exécutée en attribuant une nouvelle valeur à la propriété *Caption*. Pour ajouter à une fiche un objet texte que l'utilisateur peut faire défiler ou modifier, utilisez *TEdit*.

Barres d'état

Même si vous pouvez utiliser un volet pour créer une barre d'état, il est plus simple d'utiliser le composant barre d'état. Par défaut, la propriété *Align* d'une barre d'état a la valeur *alBottom*, ce qui gère à la fois la position et la taille.

Si vous voulez afficher une seule chaîne de texte à la fois dans la barre d'état, définissez sa propriété *SimplePanel* par *True* et utilisez la propriété *SimpleText* pour contrôler le texte affiché dans la barre d'état.

Vous pouvez aussi diviser une barre d'état en plusieurs zones de texte, appelées volets. Pour créer des volets, modifiez la propriété *Panels* avec l'inspecteur d'objets et spécifiez les propriétés *Width*, *Alignment* et *Text* de chaque volet à l'aide de l'éditeur de volets. La propriété *Text* de chaque volet contient le texte affiché dans le volet.

Barres de progression

Quand une application effectue une opération longue, vous pouvez utiliser une barre de progression pour indiquer le pourcentage réalisé de l'opération. Une barre de progression affiche une ligne pointillée qui progresse de gauche à droite.

Figure 3.5 Une barre de progression

La propriété *Position* indique la longueur de la ligne pointillée. *Max* et *Min* déterminent l'étendue des valeurs prises par *Position*. Pour allonger la ligne, augmentez *Position* en appelant la méthode *StepBy* ou *StepIt*. La propriété *Step* détermine l'incrément utilisé par *StepIt*.

Propriétés d'aide ou de conseil d'aide

La plupart des contrôles visuels peuvent, à l'exécution, afficher de l'aide contextuelle ou des conseils d'aide. Les propriétés *HelpContext* et *HelpFile* spécifient un numéro de contexte d'aide le nom du fichier d'aide pour un contrôle.

La propriété *Hint* spécifie la chaîne de texte qui apparaît quand l'utilisateur déplace le pointeur de la souris au-dessus d'un contrôle ou d'un élément de menu. Pour activer les conseils, définissez *ShowHint* par *True* ; l'initialisation de *ParentShowHint* à *True* force la propriété *ShowHint* du contrôle a prendre la même valeur que celle de son parent.

Grilles

Les grilles affichent des informations disposées en lignes et en colonnes. Si vous concevez une application de base de données, utilisez les composants *TDBGrid* et *TDBCtrlGrid* décrits au chapitre 15, "Utilisation de contrôles de données". Sinon, utilisez une grille de dessin ou une grille de chaînes standard.

Grilles de dessin

Une grille de dessin (*TDrawGrid*) affiche des données quelconques dans un format tabulaire. Ecrivez un gestionnaire d'événement *OnDrawCell* pour remplir les cellules de la grille.

- La méthode *CellRect* renvoie les coordonnées écran de la cellule spécifiée alors que la méthode *MouseToCell* renvoie la colonne et la ligne de la cellule se trouvant aux coordonnées écran spécifiées. La propriété *Selection* indique les limites de la sélection de cellules en cours.
- La propriété *TopRow* détermine la ligne qui apparaît en haut de la grille. La propriété *LeftCol* détermine la première colonne visible sur la gauche de la grille. *VisibleColCount* et *VisibleRowCount* indiquent, respectivement, le nombre de colonnes et de lignes visibles dans la grille.
- Vous pouvez modifier la largeur et la hauteur d'une colonne ou d'une ligne en utilisant les propriétés *ColWidths* et *RowHeights*. Définissez l'épaisseur des lignes du quadrillage de la grille avec la propriété *GridLineWidth*. Ajoutez des barres de défilement à la grille en utilisant la propriété *ScrollBars*.
- Vous pouvez spécifier les colonnes ou les lignes fixes (qui ne défilent pas) à l'aide des propriétés *FixedCols* et *FixedRows*. Attribuez une couleur aux colonnes et aux lignes fixes en utilisant la propriété *FixedColor*.
- Les propriétés *Options*, *DefaultColWidth* et *DefaultRowHeight* affectent également l'aspect et le comportement de la grille.

Grilles de chaînes

Le composant grille de chaînes est un descendant de *TDrawGrid* spécialisé afin de simplifier l'affichage de chaînes. La propriété *Cells* énumère les chaînes pour chaque cellule de la grille ; la propriété *Objects* énumère les objets associés à chaque chaîne. Il est possible d'accéder à toutes les chaînes et objets associés d'une colonne ou d'une ligne donnée en utilisant les propriétés *Cols* et *Rows*.

Editeur de liste de valeurs (VCL seulement)

TValueListEditor est une grille spécialisée pour la modification des listes de chaînes contenant des paires nom/valeur sous la forme Nom=Valeur. Les noms et les valeurs sont stockés dans un descendant de *TStrings* qui est la valeur de la propriété *Strings*. Vous pouvez rechercher la valeur d'un nom à l'aide de la propriété *Values*. *TValueListEditor* ne peut pas être utilisé en programmation multiplate-forme.

La grille contient deux colonnes, une pour les noms et une pour les valeurs. Par défaut, la colonne des noms s'appelle "Key" et la colonne des valeurs "Value". Vous pouvez modifier ces titres en définissant la propriété *TitleCaptions*. Vous pouvez omettre ces titres en utilisant la propriété *DisplayOptions* (qui contrôle également la façon dont se redimensionne le contrôle.)

Vous pouvez autoriser ou empêcher l'utilisateur de modifier la colonne des noms en utilisant la propriété *KeyOptions*. *KeyOptions* contient des options séparées pour autoriser la modification des noms, l'ajout de nouveaux noms, la suppression de noms, ainsi que pour déterminer si les nouveaux noms doivent être uniques.

Vous pouvez autoriser ou empêcher l'utilisateur de modifier les entrées de la colonne des valeurs en utilisant la propriété *ItemProps*. Chaque élément a un objet *TItemProp* séparé qui vous permet de :

- Fournir un masque afin d'imposer la validité de la saisie.
- Spécifier une longueur maximale pour les valeurs.
- Marquer les valeurs comme valeurs en lecture seule.
- Demander que l'éditeur de liste de valeurs affiche une flèche déroulante ouvrant la liste des valeurs parmi lesquelles l'utilisateur pourra choisir, ou un bouton Points de suspension déclenchant un événement que vous utiliserez pour afficher un dialogue dans lequel l'utilisateur entrera des données.

  Si vous spécifiez une flèche déroulante, vous devez fournir la liste des valeurs parmi lesquelles l'utilisateur peut choisir. Il peut s'agir d'une liste statique (la propriété *PickList* de l'objet *TItemProp*) ou les valeurs peuvent être ajoutées de manière dynamique à l'exécution en utilisant l'événement *OnGetPickList* de l'éditeur de liste de valeurs. Vous pouvez aussi combiner ces approches et avoir une liste statique modifiée par le gestionnaire de l'événement *OnGetPickList*.

  Si vous spécifiez un bouton Points de suspension, vous devez fournir la réponse qui est faite lorsque l'utilisateur clique sur ce bouton (y compris la définition d'une valeur, si approprié). Vous fournirez cette réponse en écrivant un gestionnaire pour l'événement *OnEditButtonClick*.

Affichage des graphiques

Les composants suivants facilitent l'incorporation d'éléments graphiques dans une application.

Utilisez ce composant :	Pour afficher :
TImage	des fichiers graphiques
TShape	des formes géométriques
TBevel	des lignes et des cadres en 3D
TPaintBox	des graphiques dessinés par l'application à l'exécution
TAnimate	des fichiers AVI (VCL seulement)

Images

Le composant image affiche une image graphique : bitmap, icône ou métafichier. La propriété *Picture* spécifie l'image à afficher. Utilisez les propriétés *Center*, *AutoSize*, *Stretch* et *Transparent* pour spécifier les options d'affichage. Pour davantage d'informations, voir "Présentation de la programmation relative aux graphiques" à la page 8-1.

Formes

Le composant forme affiche une forme géométrique. C'est un contrôle non fenêtré (dans CLX, non basé sur un widget), il ne peut donc pas recevoir la saisie de l'utilisateur. La propriété *Shape* spécifie la forme du contrôle. Pour modifier la couleur de la forme ou lui ajouter un motif, utilisez la propriété *Brush* qui contient un objet *TBrush*. Les propriétés *Color* et *Style* de *TBrush* contrôlent la manière dont la forme est dessinée.

Biseaux

Le composant biseau (*TBevel*) est une ligne qui peut apparaître en relief ou en creux. Certains composants, comme *TPanel*, disposent de propriétés intégrées pour créer des contours biseautés. Quand ces propriétés ne sont pas disponibles, utilisez un composant *TBevel* pour créer des contours, des boîtes ou des cadres biseautés.

Boîtes à peindre

Le composant boîte à peindre (*TPaintBox*) permet à une application de dessiner dans une fiche. Ecrivez un gestionnaire d'événement *OnPaint* pour restituer directement l'image dans le canevas (*Canvas*) de la boîte à peindre. Il n'est pas possible de dessiner hors des limites d'une boîte à peindre. Pour davantage d'informations, voir "Présentation de la programmation relative aux graphiques" à la page 8-1.

Contrôles animation (VCL seulement)

Le composant animation est une fenêtre qui affiche silencieusement une séquence vidéo AVI (Audio Video Interleaved). Une séquence AVI est composée d'une série de plans bitmap, comme un film. Les séquences AVI peuvent être sonorisées, mais les contrôles animation ne fonctionnent qu'avec les séquences AVI silencieuses. Les fichiers utilisés doivent être des fichiers AVI non compressés ou des séquences AVI compressées en utilisant l'algorithme RLE. Les contrôles animation ne peuvent pas être utilisés en programmation multiplate-forme.

Le composant animation comporte, entre autres, les propriétés suivantes :

- *ResHandle* est le handle Windows du module contenant la séquence AVI sous la forme d'une ressource. Initialisez *ResHandle* à l'exécution avec le handle d'instance ou le handle du module contenant la ressource animation. Après avoir initialisé *ResHandle*, affectez la propriété *ResID* ou *ResName* pour spécifier la ressource du module spécifié qui contient la séquence AVI à afficher dans le contrôle animation.
- Initialisez *AutoSize* à *True* pour que le contrôle animation ajuste sa taille à la taille des plans de la séquence AVI.
- *StartFrame* et *StopFrame* spécifient les plans où la séquence doit commencer et s'arrêter.
- Définissez *CommonAVI* pour afficher l'une des séquences AVI standard de Windows contenues dans Shell32.DLL.

- Spécifiez quand commencer ou arrêter l'animation en initialisant la propriété *Active* à *True* et *False*, respectivement, et le nombre de répétitions à effectuer en initialisant la propriété *Repetitions*.
- La propriété *Timers* permet d'afficher les plans en utilisant un timer. Cela permet de synchroniser la séquence animation avec d'autres actions, par exemple la restitution d'une piste sonore.

Développement de boîtes de dialogue

Les composants boîte de dialogue de la page Dialogues de la palette de composants permettent d'utiliser dans vos applications diverses boîtes de dialogue. Ces boîtes de dialogue donnent à vos applications une interface familière et cohérente dans laquelle l'utilisateur effectue certaines opérations communes sur les fichiers, comme l'ouverture, l'enregistrement ou l'impression. Ces boîtes de dialogue affichent et/ou obtiennent des données.

Chaque boîte de dialogue s'ouvre lorsque sa méthode *Execute* est appelée. *Execute* renvoie une valeur booléenne : si l'utilisateur choisit OK pour accepter les modifications apportées dans la boîte de dialogue, *Execute* renvoie *True* ; s'il choisit Annuler pour quitter la boîte de dialogue sans rien modifier, *Execute* renvoie *False*.

Si vous développez des applications multiplates-formes, vous pouvez utiliser les dialogues fournis avec CLX dans l'unité QDialogs. Pour les systèmes d'exploitation qui proposent des boîtes de dialogue natives pour les tâches communes, comme pour l'ouverture ou l'enregistrement d'un fichier, ou le changement de police ou de couleur, vous pouvez utiliser la propriété *UseNativeDialog*. Définissez *UseNativeDialog* par *True* si votre application doit s'exécuter dans un tel environnement et si vous voulez utiliser les dialogues natifs à la place des dialogues Qt.

Utilisation des boîtes de dialogue d'ouverture

TOpenDialog est le composant boîte de dialogue le plus couramment utilisé. Il est généralement employé par une option de menu Nouveau ou Ouvrir dans le menu Fichier de la barre de menus principale d'une fiche. La boîte de dialogue contient des contrôles qui vous permettent de sélectionner des groupes de fichiers en utilisant un caractère joker et de naviguer dans les répertoires.

Le composant *TOpenDialog* permet d'utiliser une boîte de dialogue Ouvrir dans votre application. La fonction de cette boîte de dialogue est de permettre à l'utilisateur de spécifier un fichier à ouvrir. Utilisez la méthode *Execute* pour afficher la boîte de dialogue.

Quand l'utilisateur choisit OK dans la boîte de dialogue, le nom du fichier sélectionné par l'utilisateur est stocké dans la propriété *FileName* de *TOpenDialog*. Vous pouvez ensuite utiliser cette valeur à votre guise.

L'exemple de code suivant peut être placé dans une *Action* et lié à la propriété *Action* du sous-élément d'un menu *TMainMenu* ou placé dans l'événement *OnClick* du sous-élément :

```
if OpenDialog1.Execute then
   filename := OpenDialog1.FileName;
```

Ce code affiche la boîte de dialogue et si l'utilisateur choisit le bouton OK, le nom du fichier sélectionné est copié dans la variable `filename` de type *AnsiString* préalablement déclarée.

Emploi d'objets utilitaires

La VCL et CLX proposent divers objets non-visuels qui simplifient les tâches courantes de programmation. Cette section décrit certains objets utilitaires qui facilitent les tâches suivantes :

- Utilisation des listes
- Utilisation des listes de chaînes
- Modification du registre Windows et des fichiers .INI
- Utilisation des flux

Utilisation des listes

Les objets suivants permettent de créer et gérer des listes :

Tableau 3.5 Composants de création et de gestion des listes

Objet	Gère
TList	Une liste de pointeurs
TObjectList	Une liste, gérée en mémoire, d'instances d'objets
TComponentList	Une liste, gérée en mémoire, de composants (c'est-à-dire d'instances de classes dérivées de *TComponent*)
TQueue	Une liste de pointeurs premier entré, premier sorti
TStack	Une liste de pointeurs dernier entré, premier sorti
TObjectQueue	Une liste d'objets premier entré, premier sorti
TObjectStack	Une liste d'objets dernier entré, premier sorti
TClassList	Une liste de types de classe
TCollection, *TOwnedCollection* et *TCollectionItem*	Des collections indicées d'éléments définis spécialement
TStringList	Une liste de chaînes

Pour davantage d'informations sur ces objets, consultez la référence en ligne.

Utilisation des listes de chaînes

Souvent les applications ont besoin de gérer des listes de chaînes de caractères. Par exemple, pour les éléments d'une boîte à options, les lignes d'un mémo, les noms de fonte ou les noms des lignes et colonnes d'une grille de chaînes. La VCL et CLX proposent une interface pour toutes les listes de chaînes via un objet appelé *TStrings* et son descendant *TStringList*. *TStringList* implémente les propriétés et méthodes abstraites introduites par *TStrings*, et introduit les propriétés, événements et méthodes pour

- trier les chaînes de la liste,
- interdire les chaînes en doubles dans les listes triées,
- répondre aux changements du contenu de la liste.

Outre les fonctionnalités concernant la gestion de listes de chaînes, ces objets permettent une interopérabilité simple ; vous pouvez ainsi modifier les lignes d'un mémo (instance de *TStrings*), puis utiliser ces lignes comme éléments d'une boîte à options (également une instance de *TStrings*).

Une propriété liste de chaînes apparaît dans l'inspecteur d'objets avec *TStrings* dans la colonne des valeurs. Double-cliquez sur *TStrings* pour ouvrir l'éditeur de liste de chaînes qui vous permet de modifier, ajouter ou supprimer des chaînes.

Vous pouvez également manipuler les objets liste de chaînes à l'exécution pour effectuer les opérations suivantes :

- Lecture et enregistrement des listes de chaînes
- Création d'une nouvelle liste de chaînes
- Manipulation des chaînes d'une liste
- Association d'objets à une liste de chaînes

Lecture et enregistrement des listes de chaînes

Les objets liste de chaînes disposent des méthodes *SaveToFile* et *LoadFromFile* qui permettent de stocker une liste de chaînes dans un fichier texte ou de charger un fichier texte dans une liste de chaînes. Chaque ligne du fichier texte correspond à une chaîne de la liste. En utilisant ces méthodes, vous pouvez, par exemple, créer un éditeur de texte simple en chargeant un fichier dans un composant mémo ou enregistrer les listes d'éléments de boîtes à options.

L'exemple suivant charge une copie du fichier WIN.INI dans un champ mémo et en fait une copie de sauvegarde nommée WIN.BAK.

```
procedure EditWinIni;
var
  FileName: string;{ stocke le nom du fichier }
begin
  FileName := 'C:\WINDOWS\WIN.INI';{ définit le nom du fichier }
  with Form1.Memo1.Lines do
  begin
   LoadFromFile(FileName);{ lit le fichier }
   SaveToFile(ChangeFileExt(FileName, '.BAK'));{ enregistre dans un fichier de sauvegarde }
  end;
end;
```

Création d'une nouvelle liste de chaînes

Habituellement, les listes de chaînes font partie de composants. Néanmoins, il est parfois commode de créer des listes de chaînes autonomes qui n'ont pas de composant associé (par exemple, pour stocker les chaînes d'une table de référence). La manière de créer et de gérer une liste de chaînes varie selon que la liste est une liste à court terme (construite, utilisée et détruite dans une même routine) ou une liste à long terme (disponible jusqu'à l'arrêt de l'application). Quel que soit le type de liste de chaînes créé, n'oubliez pas que c'est à vous de libérer la liste quand vous n'en n'avez plus besoin

Listes de chaînes à court terme

Si vous utilisez une liste de chaînes uniquement pour la durée d'une seule routine, vous pouvez la créer, l'utiliser et la détruire au même emplacement. C'est la méthode la plus fiable pour utiliser des objets liste de chaînes. Comme l'objet liste de chaînes alloue la mémoire pour lui-même et pour ses chaînes, il est important de protéger l'allocation en utilisant un bloc **try...finally** afin de garantir que l'objet libère sa mémoire même si une exception a lieu.

1 Construire l'objet liste de chaînes.
2 Utiliser la liste de chaînes dans la partie **try** d'un bloc **try...finally**.
3 Libérer l'objet liste de chaînes dans la partie **finally**.

Le gestionnaire d'événement suivant répond au choix d'un bouton en construisant un objet liste de chaînes, en l'utilisant puis en le détruisant :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TempList: TStrings;{ déclare la liste }
begin
  TempList := TStringList.Create;{ construit l'objet liste }
  try
    { utilise la liste de chaînes }
  finally
    TempList.Free;{ détruit l'objet liste }
  end;
end;
```

Listes de chaînes à long terme

Si la liste de chaînes doit être disponible tout au long de l'exécution de votre application, vous devez construire la liste au démarrage de l'application et la détruire avant la fermeture de l'application.

1 Dans le fichier unité de la fiche principale de votre application, ajoutez un champ de type *TStrings* à la déclaration de la fiche.

2 Créez un gestionnaire d'événement pour le *constructeur* de la fiche principale qui s'exécute avant que la fiche n'apparaisse. Ce gestionnaire d'événement doit créer une liste de chaînes et l'affecter au champ déclaré dans la première étape.

3 Ecrivez un gestionnaire d'événement qui libère la liste de chaînes dans l'événement *OnClose* de la fiche.

L'exemple suivant utilise une liste de chaînes à long terme pour stocker les clics de la souris dans la fiche principale, puis enregistre la liste dans un fichier avant l'arrêt de l'application.

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
{Pour CLX : uses SysUtils, Classes, QGraphics, QControls, QForms, Qialogs;}

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
    ClickList: TStrings;{ déclare le champ }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  ClickList := TStringList.Create;{ construit la liste }
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  ClickList.SaveToFile(ChangeFileExt(Application.ExeName, '.LOG'));{ enregistre la liste }
  ClickList.Free;{ détruit l'objet liste }
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  ClickList.Add(Format('Cliquer à (%d, %d)', [X, Y]));{ ajoute une chaîne à la liste }
end;

end.
```

Manipulation des chaînes d'une liste

Les opérations couramment effectuées sur les listes de chaînes sont les suivantes :

- Comptage des chaînes d'une liste
- Accès à une chaîne spécifique
- Recherche de la position d'une chaîne dans la liste
- Parcours des chaînes d'une liste
- Ajout d'une chaîne à une liste
- Déplacement d'une chaîne dans une liste
- Suppression d'une chaîne d'une liste
- Copie de la totalité d'une liste de chaînes

Comptage des chaînes d'une liste

La propriété en lecture seule *Count* renvoie le nombre de chaînes dans la liste. Comme les listes de chaînes utilisent des indices de base zéro, *Count* correspond à l'indice de la dernière chaîne plus un.

Accès à une chaîne spécifique

La propriété tableau *Strings* contient les chaînes de la liste, référencées par un indice de base zéro. Comme *Strings* est la propriété par défaut des listes de chaînes, vous pouvez omettre l'identificateur *Strings* pour accéder à la liste ; donc

```
StringList1.Strings[0] := 'Première chaîne.';
```

est équivalent à

```
StringList1[0] := 'Première chaîne.';
```

Recherche d'éléments dans une liste de chaînes

Pour rechercher une chaîne dans une liste de chaînes, utilisez la méthode *IndexOf*. *IndexOf* renvoie l'indice de la première chaîne de la liste qui correspond au paramètre transmis et renvoie –1 si la chaîne transmise en paramètre n'est pas trouvée. *IndexOf* recherche uniquement une correspondance exacte ; si vous voulez obtenir des chaînes de correspondance partielle, vous devez parcourir la liste de chaînes.

Vous pouvez, par exemple, utiliser *IndexOf* pour déterminer si un nom de fichier donné se trouve dans les éléments (*Items)* d'une boîte liste :

```
if FileListBox1.Items.IndexOf('WIN.INI') > -1 ...
```

Parcours des chaînes d'une liste

Pour parcourir les chaînes d'une liste, utilisez une boucle **for** allant de zéro à *Count* – 1.

L'exemple suivant convertit en majuscules chaque chaîne d'une boîte liste.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Index: Integer;
begin
```

```
  for Index := 0 to ListBox1.Items.Count - 1 do
    ListBox1.Items[Index] := UpperCase(ListBox1.Items[Index]);
end;
```

Ajout d'une chaîne à une liste

Pour ajouter une chaîne à la fin d'une liste de chaînes, utilisez la méthode *Add* en lui transmettant en paramètre la nouvelle chaîne. Pour insérer une chaîne dans la liste, appelez la méthode *Insert* en lui transmettant deux paramètres : la chaîne et l'indice à laquelle elle doit être placée. Si, par exemple, vous voulez placer la chaîne "Trois" en troisième position dans la liste, utilisez :

```
Insert(2, 'Trois');
```

Pour ajouter à une liste les chaînes d'une liste, appelez *AddStrings* :

```
StringList1.AddStrings(StringList2);  { ajoute à StringList1 les chaînes de StringList2 }
```

Déplacement d'une chaîne dans une liste

Pour déplacer une chaîne dans une liste de chaînes, appelez la méthode *Move* en lui transmettant deux paramètres : l'indice en cours de la chaîne et son nouvel indice. Par exemple, pour déplacer la troisième chaîne de la liste en cinquième position, utilisez :

```
Move(2, 4)
```

Suppression d'une chaîne d'une liste

Pour supprimer une chaîne d'une liste de chaînes, appelez la méthode *Delete* de la liste en lui transmettant l'indice de la chaîne à supprimer. Si vous ne connaissez pas l'indice de la chaîne à supprimer, utilisez la méthode *IndexOf* pour le déterminer. Pour supprimer toutes les chaînes de la liste, utilisez la méthode *Clear*.

L'exemple suivant utilise *IndexOf* et *Delete* pour trouver et supprimer une chaîne :

```
with ListBox1.Items do
begin
  BIndex := IndexOf('bureaucratie');
  if BIndex > -1 then
    Delete(BIndex);
end;
```

Copie de la totalité d'une liste de chaînes

Vous pouvez utiliser la méthode *Assign* pour copier les chaînes d'une liste source vers une liste de destination en remplaçant le contenu de la liste de destination. Pour ajouter les chaînes sans remplacer la liste de destination, utilisez la méthode *AddStrings*. Par exemple,

```
Memo1.Lines.Assign(ComboBox1.Items);    { remplace les chaînes existantes }
```

copie les lignes d'une boîte à options dans un mémo (en écrasant le contenu du mémo), alors que :

```
Memo1.Lines.AddStrings(ComboBox1.Items);   { ajoute les chaînes à la fin }
```

ajoute au mémo les lignes de la boîte à options.

Quand vous effectuez une copie locale d'une liste de chaînes, utilisez la méthode *Assign*. Si vous affectez une variable liste de chaînes à une autre :

```
StringList1 := StringList2;
```

l'objet liste de chaîne initial est perdu, ce qui peut donner des résultats imprévisibles.

Association d'objets à une liste de chaînes

Outre les chaînes stockées dans sa propriété *Strings*, une liste de chaînes peut gérer des références à des *objets* dans sa propriété *Objects*. Comme *Strings*, *Objects* est un tableau d'indice zéro. Le plus souvent, *Objects* sert à associer des bitmaps aux chaînes dans des contrôles dessinés par le propriétaire.

Utilisez la méthode *AddObject* ou *InsertObject* pour ajouter en une seule étape la chaîne et son objet associé à la liste. *IndexOfObject* renvoie l'indice de la première chaîne de la liste associée à l'objet spécifié. Les méthodes comme *Delete*, *Clear*, et *Move* agissent à la fois sur les chaînes et les objets ; ainsi, la suppression d'une chaîne supprime également l'éventuel objet correspondant.

Pour associer un objet à une chaîne existante, affectez l'objet à la propriété *Objects* pour le même indice. Vous ne pouvez pas ajouter d'objet sans ajouter une chaîne correspondante.

Registre Windows et fichiers .INI (VCL seulement)

Le registre système Windows est une base de données hiérarchique dans laquelle les applications stockent des informations de configuration. La classe VCL *TRegistry* propose les méthodes permettant de lire et d'écrire dans le registre.

Avant Windows 95, la plupart des applications stockaient les informations de configuration dans des fichiers d'initialisation, utilisant généralement l'extension .INI. La VCL propose les classes suivantes pour faciliter la maintenance et la migration de programmes utilisant les fichiers INI.

- *TRegistry* pour utiliser le registre (VCL seulement).
- *TIniFile* (VCL seulement) ou *TMemIniFile* pour utiliser les fichiers INI.
- *TRegistryIniFile* pour utiliser à la fois le registre et les fichiers INI (VCL seulement). *TRegistryIniFile* dispose de propriétés et méthodes similaires à celles de *TIniFile* mais il lit et écrit dans le registre système. En utilisant une variable de type *TCustomIniFile* (l'ancêtre commun à *TIniFile*, *TMemIniFile* et *TRegistryIniFile*), vous pouvez écrire un code générique qui accède soit au registre, soit à un fichier INI, selon l'endroit où il est utilisé.

Seul *TMemIniFile* peut être utilisé en programmation multiplate-forme.

Utilisation de TIniFile (VCL seulement)

Le format des fichiers .INI est toujours utilisé ; la plupart des fichiers de configuration de Delphi (comme le fichier DSK de configuration du bureau) sont dans ce format. Comme ce format de fichier était et est toujours important, la VCL fournit une classe facilitant la lecture et l'écriture de ces fichiers. *TIniFile* ne peut pas être utilisé en programmation multiplate-forme.

Quand vous instanciez l'objet *TIniFile*, il faut transmettre au constructeur un paramètre spécifiant le nom du fichier INI. Si le fichier n'existe pas déjà, il est automatiquement créé. Vous pouvez alors utiliser la méthode *ReadString*, *ReadInteger* ou *ReadBool*. Par ailleurs, si vous souhaitez lire une section entière du fichier .INI, vous pouvez utiliser la méthode*ReadSection*. Vous pouvez, inversement, écrire des valeurs en utilisant *WriteBool*, *WriteInteger* ou *WriteString*.

Chacune des routines Read attend trois paramètres. Le premier identifie la section du fichier .INI. Le second paramètre identifie la valeur à lire et le troisième est une valeur par défaut à utiliser si la section ou la valeur n'existe pas dans le fichier INI. De même, les routines Write créent la section et/ou la valeur si elles n'existent pas. L'exemple précédent crée un fichier .INI la première fois qu'il est exécuté, et ce fichier ressemble à ceci :

```
[Form]
Top=185
Left=280
Caption=Default Caption
InitMax=0
```

Lors des exécutions ultérieures de cette application, les valeurs INI sont lues lors de la création de la fiche et réécrites dans l'événement *OnClose*.

Utilisation de TRegistry

La plupart des applications 32 bits stockent leurs informations dans cette base plutôt que dans les fichiers .INI, car la base de registres est hiérarchique, plus robuste, et ne souffre pas des limitations de taille inhérentes aux fichiers .INI. L'objet *TRegistry* contient des méthodes pour ouvrir, fermer, sauvegarder, déplacer, copier et supprimer des clés.

TRegistry ne peut pas être utilisé en programmation multiplate-forme.

Pour plus d'informations, voir la rubrique *TRegistry* dans l'aide en ligne.

Utilisation de TRegIniFile

Si vous êtes habitué à l'utilisation des fichiers .INI, et que vous souhaitiez déplacer vos informations de configuration dans la base de registres, vous pouvez utiliser la classe *TRegIniFile*. *TRegIniFile* a été conçue pour utiliser les entrées de la base de registres de façon similaire à celles des fichiers .INI. Toutes les méthodes de *TIniFile* (lecture et écriture) existent dans *TRegIniFile*.

Quand vous créez un objet *TRegIniFile*, le paramètre que vous passez (le nom du fichier pour un objet *IniFile*) devient le nom d'une clé dans la clé de l'utilisateur en cours dans la base de registres, et toutes les sections et valeurs commencent à partir de cette racine. En fait, cet objet simplifie considérablement l'interfaçage

avec la base de registres, de sorte que vous pourriez vouloir l'utiliser plutôt que *TRegistry*, même si vous ne portez pas un code existant.

TRegIniFile ne peut pas être utilisé en programmation multiplate-forme.

Pour davantage d'informations, voir dans la référence VCL en ligne la rubrique *TRegIniFile*.

Création d'espaces de dessin

Le *TCanvas* encapsule un contexte de périphérique Windows dans la VCL et un dispositif de peinture (peintre Qt) dans CLX, qui gère tous les affichages dans les fiches, les conteneurs visuels (comme les volets) et l'objet imprimante (traité dans "Impression" à la page 3-61).

En utilisant l'objet canevas, vous n'avez plus besoin d'allouer crayons, pinceaux ou palettes : ils sont alloués et libérés automatiquement.

TCanvas propose un grand nombre de routines de dessin pour dessiner des lignes, des formes, des polygones, du texte, etc. dans tout contrôle contenant un canevas. Par exemple, voici un gestionnaire d'événement de bouton qui dessine une ligne depuis le coin supérieur gauche au milieu de la fiche, et affiche du texte brut sur la fiche :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Canvas.Pen.Color := clBlue;
   Canvas.MoveTo( 10, 10 );
   Canvas.LineTo( 100, 100 );
   Canvas.Brush.Color := clBtnFace;
   Canvas.Font.Name := 'Arial';
   Canvas.TextOut( Canvas.PenPos.x, Canvas.PenPos.y,'Fin de la ligne');
end;
```

Dans les applications Windows, l'objet *TCanvas* vous protège également contre les erreurs graphiques courantes de Windows, comme la restauration des contextes de périphérique, des crayons, des pinceaux, etc. à la valeur qu'ils avaient avant l'opération de dessin. *TCanvas* est utilisé dans Delphi quand il faut dessiner et il permet de le faire de manière à la fois simple et fiable.

Pour une liste complète des propriétés et méthodes de *TCanvas,* voir la référence en ligne.

Impression

L'objet VCL *TPrinter* encapsule les détails de l'impression sous Windows. Pour obtenir une liste des imprimantes disponibles, utilisez la propriété *Printers*. L'objet CLX *TPrinter* est un dispositif de peinture qui opère sur une imprimante. Il génère du postscript et l'envoie à lpr, lp ou à toute autre commande d'impression.

Les deux objets imprimante utilisent un *TCanvas* (similaire au *TCanvas* de la fiche), ce qui signifie que tout ce qui peut être dessiné dans une fiche peut également être imprimé. Pour imprimer une image, appelez d'abord la méthode *BeginDoc*, puis les routines des dessins de canevas à imprimer (y compris du texte en utilisant la méthode *TextOut*), et envoyez la tâche à l'imprimante en appelant la méthode *EndDoc*.

Cet exemple utilise un bouton et un mémo sur une fiche. Quand l'utilisateur clique sur le bouton, le contenu du mémo s'imprime avec une marge de 200 pixels autour de la page.

Pour pouvoir exécuter cet exemple, vous devez ajouter Printers dans votre clause **uses**.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRect;
  i: Integer;
begin
  with Printer do
    begin
      r := Rect(200,200,(Pagewidth - 200),(PageHeight - 200));
      BeginDoc;
      for i := 0 to Memo1.Lines.Count do
       Canvas.TextOut(200,200 + (i *
Canvas.TextHeight(Memo1.Lines.Strings[i])),
                                    Memo1.Lines.Strings[i]);
      Canvas.Brush.Color := clBlack;
      Canvas.FrameRect(r);
      EndDoc;
    end;
end;
```

Pour davantage d'informations sur l'objet *TPrinter*, voir dans l'aide en ligne la rubrique du même nom.

Utilisation des flux

Les flux sont simplement des moyens de lire et d'écrire des données. Ils offrent une interface commune pour la lecture et l'écriture dans différents supports, tels la mémoire, les chaînes, les sockets et les flux de blobs.

Dans l'exemple suivant, un fichier est copié dans un autre à l'aide de flux. L'application comprend deux contrôles d'édition (From et To) et un bouton pour copier le fichier.

```
procedure TForm1.CopyFileClick(Sender: TObject);
var
  stream1, stream2:TStream;
begin
  stream1:=TFileStream.Create(From.Text,fmOpenRead or fmShareDenyWrite);
  try
    stream2 := TFileStream.Create(To.Text fmOpenCreate or fmShareDenyRead);
    try
```

```
      stream2.CopyFrom(Stream1,Stream1.Size);
    finally
      stream2.Free;
  finally
    stream1.Free
end;
```

Utilisez des objets flux spécialisés pour lire, écrire dans un support de stockage. Chaque descendant de *TStream* implémente des méthodes pour accéder à un support de stockage particulier : fichier disque, mémoire dynamique, etc. Les descendants de *TStream* sont *TFileStream*, *TStringStream* et *TMemoryStream*. Outre les méthodes de lecture et d'écriture, ces objets permettent aux applications de se positionner de manière arbitraire dans le flux. Les propriétés de *TStream* donnent des informations sur le flux, comme sa taille ou la position en cours.

Chapitre

4

Sujets de programmation généraux

Ce chapitre décrit comment effectuer dans Delphi les tâches de programmation les plus courantes :

- Compréhension des classes
- Définition des classes
- Gestion des exceptions
- Utilisation des interfaces
- Définition de variants personnalisés.
- Utilisation des chaînes
- Utilisation des fichiers
- Conversion de mesures

Compréhension des classes

Une *classe* est une définition abstraite de propriétés, de méthodes, d'événements et de membres (comme les variables locales à la classe). Lorsque vous créez une instance d'une classe, cette instance est appelé objet. Le terme objet est souvent utilisé de manière inexacte dans la documentation Delphi et, lorsque la distinction entre une classe et une instance de cette classe est sans importance, le terme "objet" peut également faire référence à une classe.

Bien que Delphi comprenne de nombreuses classes dans sa hiérarchie d'objets, il vous faudra probablement en créer de nouvelles si vous écrivez des programmes orientés objets. Les classes que vous écrivez doivent descendre de *TObject* ou de l'un de ses descendants. La déclaration du type d'une classe peut posséder trois sections pour contrôler l'accessibilité de ses champs et de ses méthodes :

```
Type
  TClassName = Class(TObject)
    public
      {champs publics}
      {méthodes publiques}
```

```
    protected
      {champs protégés}
      {méthodes protégées}
    private
      {champs privés}
      {méthodes privées}
end;
```

- La section public déclare les champs et les méthodes sans aucune contrainte d'accès ; les instances de la classe et les classes dérivées peuvent accéder à ces champs et à ces méthodes.
- La section protected inclut les champs et les méthodes comportant certaines contraintes d'accès ; les classes dérivées peuvent accéder à ces champs et à ces méthodes.
- La section private déclare les champs et les méthodes ayant de fortes contraintes d'accès ; ils ne peuvent être accédés ni par les instances de la classe ni par les classes dérivées.

L'intérêt de l'utilisation des classes est que vous pouvez créer les nouvelles classes en les faisant dériver des classes existantes. Chaque classe hérite des champs et des méthodes de son parent et de ses classes ancêtres. Vous pouvez aussi déclarer des méthodes de la nouvelle classe qui surchargent les méthodes héritées, introduisant ainsi un nouveau comportement plus spécialisé.

La syntaxe générale d'une classe dérivée est la suivante :

```
Type
  TClassName = Class (TParentClass)
    public
      {champs publics}
      {méthodes publiques}
    protected
      {champs protégés}
      {méthodes protégées}
    private
      {champs privés}
      {méthodes privées}
end;
```

Si aucun nom de classe parent n'est spécifié, la classe hérite directement de *TObject*. *TObject* ne définit que quelques méthodes, ainsi qu'un constructeur et un destructeur de base.

Pour davantage d'informations sur la syntaxe, les définitions du langage et les règles à respecter pour les classes, voir dans l'aide en ligne du *Guide du langage Pascal Objet* la rubrique Types de classes.

Définition des classes

Delphi vous permet de déclarer des classes implémentant les fonctionnalités de programmation que vous voulez utiliser dans votre application. Certaines versions de Delphi comprennent une fonctionnalité appelée achèvement de classe qui simplifie le travail de définition et d'implémentation des nouvelles classes en générant le squelette du code nécessaire aux membres de classe que vous déclarez.

Pour définir une classe,

1. Dans l'EDI, commencez par ouvrir un projet puis choisissez Fichier | Nouveau | Unité afin de créer la nouvelle unité où vous pourrez définir la nouvelle classe.
2. Ajoutez la clause **uses** et la section **type** à la section **interface**.
3. Dans la section **type**, écrivez la déclaration de la classe. Vous devez déclarer toutes les variables membres, toutes les propriétés, toutes les méthodes et tous les événements.

```
TMyClass = class; {Descend implicitement de TObject}
public
  .
  .
  .
  .
  .
  .
private
  .
  .
  .
published {Si dérivé de TPersistent ou d'un niveau inférieur}
  .
  .
  .
```

Remarque L'objet qui contient les données du variant personnalisé doit être compilé avec RTTI. Cela signifie qu'il doit être compilé en utilisant la directive {$M+} ou descendre de *TPersistent* ou d'un descendant.

Si vous voulez que la classe dérive d'une classe particulière, vous devez indiquer cette classe dans la définition :

```
TMyClass = class(TParentClass); {Descend de TParentClass}
```

Par exemple :

```
type TMyButton = class(TButton)
  property Size: Integer;
  procedure DoSomething;
end;
```

Si votre version de Delphi supporte l'achèvement de classe : placez le curseur à l'intérieur de la définition d'une méthode dans la section **interface** et

appuyez sur Ctrl+Maj+C (ou cliquez avec le bouton droit et sélectionnez Compléter la classe sous le curseur). Delphi complète toutes les déclarations de propriété inachevées et crée les méthodes vides nécessaires dans la section **implementation**. (Si vous ne bénéficiez pas de l'achèvement de classe, vous devrez écrire le code vous-même, en complétant les déclarations de propriété et en écrivant les méthodes.)

Dans le cas de l'exemple précédent, si vous bénéficiez de l'achèvement de classe, Delphi ajoute les spécificateurs **read** et **write** à la déclaration de votre interface, y compris les champs ou les méthodes de support :

```
type TMyButton = class(TButton)
  property Size: Integer read FSize write SetSize;
  procedure FaireQuelqueChose;
private
  FSize: Integer;
  procedure SetSize(const Value: Integer);
```

Il ajoute également le code suivant à la section **implementation** de l'unité.

```
{ TMyButton }
procedure TMyButton.FaireQuelqueChose;
begin

end;
procedure TMyButton.SetSize(const Value: Integer);
begin
   FSize := Value;
end;
```

4 Remplissez les méthodes. Par exemple, pour faire sonner le bouton lorsque vous appelez la méthode FaireQuelqueChose, ajoutez un Beep entre **begin** et **end**.

```
{ TMyButton }
procedure TMyButton.FaireQuelqueChose;
begin
  Beep;
end;

procedure TMyButton.SetSize(const Value: Integer);
begin
  if fsize < > value then
  begin
    FSize := Value;
    FaireQuelqueChose;
  end;
end;
```

Remarquez que le bouton émet également un bip lorsque vous appelez SetSize pour modifier la taille du bouton.

Pour davantage d'informations sur la syntaxe, les définitions du langage et les règles à respecter pour les classes et les méthodes, voir dans l'aide en ligne du *Guide du langage Pascal Objet* les rubriques Types de classes et méthodes.

Gestion des exceptions

Delphi propose un mécanisme permettant de gérer les erreurs d'une manière systématique. La gestion des exceptions permet à l'application de gérer les erreurs si c'est possible et, si c'est nécessaire, de se fermer sans perdre de données ou de ressources. Les conditions d'erreurs sont indiquées dans Delphi par des exceptions. Cette section décrit les aspects suivants de l'utilisation des exceptions pour créer des applications fiables :

- Protection des blocs de code
- Protection de l'allocation de ressources
- Gestion des exceptions RTL
- Gestion des exceptions des composants
- Gestion des exceptions et sources externes
- Exceptions silencieuses
- Définition d'exceptions personnalisées

Protection des blocs de code

Pour rendre vos applications plus fiables, votre code doit reconnaître les exceptions quand elles se produisent et y répondre. Si vous ne spécifiez pas de réponse, l'application affiche une boîte message décrivant l'erreur. Votre travail consiste donc à déterminer où les erreurs peuvent se produire et à définir des réponses à ces erreurs, en particulier dans les situations où une erreur peut entraîner une perte de données ou de ressources système.

Quand vous créez la réponse à une exception, vous le faites pour des blocs de code. Quand une suite d'instructions nécessitent toutes le même type de réponse aux erreurs, vous pouvez les regrouper dans un bloc et définir des réponses aux erreurs qui portent sur la totalité du bloc.

Les blocs disposant de réponses spécifiques à des exceptions sont appelés des blocs protégés car ils sont capables de se prémunir contre les erreurs qui, sinon, provoquent l'arrêt de l'application ou la perte de données.

Pour protéger des blocs de code, vous devez maîtriser :

- Réponse aux exceptions
- Exceptions et contrôle d'exécution
- Réponses à des exceptions imbriquées

Réponse aux exceptions

Quand une condition d'erreur se produit, l'application déclenche une exception : elle crée un objet exception. Une fois l'exception déclenchée, votre application peut exécuter du code de nettoyage, gérer l'exception ou faire les deux.

Exécution de code de nettoyage

La manière la plus simple de répondre à une exception est de garantir que du code de nettoyage est bien exécuté. Ce type de réponse ne corrige pas la

situation qui a provoqué l'erreur mais vous assure que l'application ne laisse pas l'environnement dans un état instable. Généralement, vous utiliserez ce type de réponse pour garantir que l'application libère bien les ressources allouées quelle que soit l'erreur qui a eu lieu.

Gestion d'une exception

C'est une réponse spécifique à un type particulier d'exception. La gestion d'une exception supprime la condition d'erreur et détruit l'objet exception, ce qui permet à l'application de poursuivre son exécution. Normalement, vous définissez des gestionnaires d'exceptions pour permettre aux applications de se rétablir après des erreurs et de poursuivre l'exécution. Vous pouvez gérer des types divers d'exceptions, par exemple des tentatives d'ouverture d'un fichier inexistant, l'écriture dans un disque plein ou des débordements dans des calculs. Certaines d'entre elles, comme "Fichier non trouvé" sont faciles à corriger et à reprendre tandis que d'autres, comme l'insuffisance de mémoire, sont plus difficiles à corriger pour l'application ou l'utilisateur.

La gestion efficace des exceptions nécessite la maîtrise des sujets suivants :

- Création d'un gestionnaire d'exception
- Instructions de gestion des exceptions
- Utilisation de l'instance d'exception
- Portée des gestionnaires d'exceptions
- Spécification du gestionnaire d'exception par défaut
- Gestion des classes d'exceptions
- Redéclenchement de l'exception

Exceptions et contrôle d'exécution

Le Pascal Objet permet d'inclure facilement la gestion des erreurs dans les applications car les exceptions ne rentrent pas dans le déroulement normal du code. En fait, en déplaçant la vérification et la gestion des erreurs hors du déroulement principal de vos algorithmes, les exceptions peuvent simplifier le code que vous écrivez.

Quand vous déclarez un bloc protégé, vous définissez des réponses spécifiques aux exceptions qui peuvent se produire à l'intérieur du bloc. Quand une exception se produit dans le bloc, l'exécution sort du bloc, passe immédiatement à la réponse que vous avez défini.

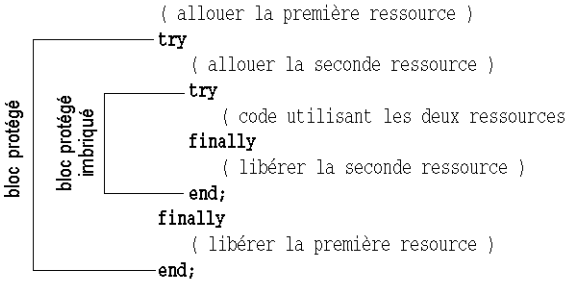
Exemple Le code suivant comporte un bloc protégé. Si une exception se produit dans le bloc protégé, l'exécution passe à la partie gestion d'exception qui génère un bip sonore. L'exécution se poursuit hors du bloc :

```
try
  AssignFile(F, FileName);
  Reset(F);
  ⋮
except
  on Exception do Beep;
end;
  ⋮ { l'exécution reprend ici, hors du bloc protégé }
```

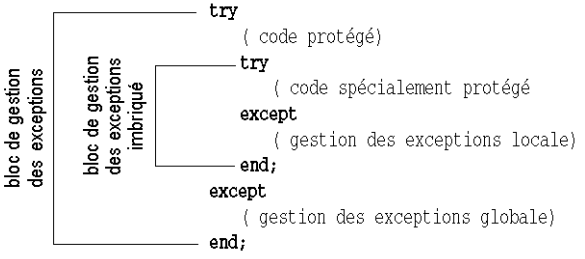
Réponses à des exceptions imbriquées

Votre code définit des réponses aux exceptions se produisant dans des blocs. Comme le Pascal permet d'imbriquer des blocs de code à l'intérieur d'autres blocs de code, vous pouvez même personnaliser les réponses à l'intérieur de blocs qui contiennent déjà des réponses personnalisées.

Dans le cas le plus simple vous pouvez, par exemple protéger l'allocation d'une ressource et, à l'intérieur de ce bloc protégé, définir des blocs qui allouent et protègent d'autres ressources. Conceptuellement, cela peut se représenter de la manière suivante :



Vous pouvez également utiliser des blocs imbriqués afin de définir une gestion locale, pour des exceptions spécifiques, qui redéfinit la gestion du bloc environnant. Conceptuellement, cela peut se représenter de la manière suivante :



Vous pouvez également mélanger différentes sortes de blocs de réponse aux exceptions, par exemple en imbriquant la protection de ressources dans des blocs de gestion d'exceptions ou l'inverse.

Protection de l'allocation de ressources

Pour garantir la fiabilité de vos applications, un élément essentiel est de s'assurer lors de l'allocation des ressources que vous les libérez même si une exception a lieu. Ainsi, quand votre application alloue de la mémoire, vous devez vous assurer qu'elle la libère bien. De même, si elle ouvre un fichier, vous devez vous assurer que le fichier est bien fermé ultérieurement.

N'oubliez pas que votre code n'est pas seul à générer des exceptions. L'appel d'une routine RTL ou d'un autre composant de votre application peut aussi

déclencher une exception. Votre code doit s'assurer que même dans ces situations les ressources allouées sont correctement libérées.

Pour protéger les ressources de manière fiable, vous devez maîtriser ceci :

- Quelles ressources doivent être protégées ?
- Création d'un bloc de protection de ressource

Quelles ressources doivent être protégées ?

Dans une situation normale, vous pouvez garantir qu'une application libère les ressources allouées en spécifiant simplement le code d'allocation et de libération des ressources. Quand des exceptions ont lieu, vous devez vous assurer que l'application exécute quand même le code de libération des ressources.

Certaines ressources courantes doivent toujours être libérées :

- Les fichiers
- La mémoire
- Les ressources Windows (VCL seulement)
- Les objets

Exemple Le gestionnaire d'événement suivant alloue de la mémoire puis génère une erreur ; il ne libère donc jamais la mémoire :

```
procedure TForm1.Button1Click(Sender: TComponent);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allouer 1Ko de mémoire }
  AnInteger := 10 div ADividend;{ cela provoque une erreur }
  FreeMem(APointer, 1024);{ Le code n'arrive jamais ici }
end;
```

Toutes les erreurs ne sont pas aussi évidentes, mais cet exemple illustre un principe important : quand l'erreur de division par zéro se produit, l'exécution sort du bloc, ainsi l'instruction *FreeMem* n'est donc jamais exécutée pour libérer la mémoire.

Pour être certain que *FreeMem* a la possibilité de libérer le bloc de mémoire alloué par *GetMem*, vous devez placer le code dans un bloc de protection de ressource.

Création d'un bloc de protection de ressource

Pour garantir que des ressources allouées sont effectivement libérées, même en cas d'exception, vous devez intégrer le code utilisant la ressource dans un bloc protégé, le code de libération de la ressource étant placé dans une partie spéciale du bloc. Voici l'organisation générale d'une allocation protégée de ressource :

```
{ allocation de la ressource }
try
  { instructions utilisant la ressource }
finally
```

```
  { libération de la ressource }
end;
```

Le secret de l'instruction **try..finally,** c'est que l'application exécute toujours les instructions placées dans la partie **finally** du bloc même quand des exceptions se produisent dans le bloc protégé. Si du code (même une routine appelée) de la partie **try** du bloc déclenche une exception, l'exécution s'interrompt là. Quand un gestionnaire d'exception est trouvé, l'exécution se poursuit dans la partie **finally**, qui est appelée le "code de nettoyage". Une fois la partie finally exécutée, le gestionnaire d'exception est appelé. Quand il n'y a pas d'exception, le code de nettoyage est exécuté dans l'ordre normal après toutes les instructions de la partie **try**.

Exemple Le gestionnaire d'événement défini par le code suivant alloue de la mémoire et génère une erreur mais libère quand même la mémoire allouée :

```
procedure TForm1.Button1Click(Sender: TComponent);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;

begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allouer 1Ko de mémoire }
  try
    AnInteger := 10 div ADividend;{ génère une erreur }
  finally
    FreeMem(APointer, 1024);{ malgré l'erreur, l'exécution reprend ici }
  end;
end;
```

Les instructions placées dans le bloc **finally** ne dépendent pas de l'apparition d'une exception. Si les instructions de la partie **try** ne déclenchent pas d'exception, l'exécution se poursuit quand même par le bloc **finally**.

Gestion des exceptions RTL

Quand vous écrivez du code qui appelle les routines de la bibliothèque d'exécution (RTL), comme les fonctions mathématiques ou les procédures de gestion de fichier, la RTL informe votre application des erreurs par le biais d'exceptions. Par défaut, les exceptions RTL génèrent un message affiché par l'application. Vous pouvez définir vos propres gestionnaires pour gérer différemment les exceptions RTL.

Il existe également des exceptions silencieuses qui, par défaut, n'affichent pas de message.

Les exceptions RTL sont gérées comme les autres exceptions. La gestion des exceptions RTL nécessite la maîtrise des sujets suivants :

- Qu'est-ce qu'une exception RTL ?
- Création d'un gestionnaire d'exception
- Instructions de gestion des exceptions
- Utilisation de l'instance d'exception

- Portée des gestionnaires d'exceptions
- Spécification du gestionnaire d'exception par défaut
- Gestion des classes d'exceptions
- Redéclenchement de l'exception

Qu'est-ce qu'une exception RTL ?

Les exceptions de la bibliothèque d'exécution sont définies dans l'unité *SysUtils*, elles dérivent toutes d'un type d'objet exception générique appelé Exception. Exception définit la chaîne du message, affiché par défaut, par les exceptions RTL.

Il existe plusieurs sortes d'exceptions déclenchées par la RTL, décrites dans le tableau suivant.

Tableau 4.1 Exceptions RTL

Type d'erreur	Cause	Signification
Entrées/Sorties	Erreur d'accès à un fichier ou à un périphérique d'E/S.	La plupart des exceptions d'E/S sont liées à des codes d'erreur renvoyés lors de l'accès à un fichier.
Tas	Erreur d'utilisation de la mémoire dynamique.	Les erreurs de tas se produisent quand il n'y a pas assez de mémoire disponible ou lorsqu'une application libère un pointeur qui pointe hors du tas.
Calcul entier	Opération illégale sur des expressions de type entier.	Ces erreurs sont la division par zéro, les nombres et les expressions hors étendue et les débordements.
Calcul à virgule flottante	Opération illégale sur des expressions de type réel.	Les erreurs dans les calculs à virgule flottante proviennent du coprocesseur ou de l'émulateur logiciel. Ces erreurs sont les instructions incorrectes, la division par zéro et les débordements.
Transtypage	Transtypage incorrect avec l'opérateur **as**.	Les objets ne peuvent être transtypés que dans des types compatibles.
Conversion	Conversion de type incorrect	Les fonctions de conversion de type comme IntToStr, StrToInt ou StrToFloat déclenchent des exceptions de conversion quand le paramètre ne peut être converti dans le type souhaité.
Matérielle	Condition du système	Les exceptions matérielles indiquent que le processeur ou l'utilisateur a généré une condition d'erreur ou une interruption, par exemple une violation d'accès, un débordement de pile ou une interruption clavier.
Variant	Coercition de type illégale	Des erreurs peuvent se produire dans des expressions faisant référence à des variants quand le variant ne peut être forcé dans un type compatible.

Pour avoir la liste des types d'exception RTL, voir le code de l'unité *SysUtils*.

Création d'un gestionnaire d'exception

Un gestionnaire d'exception est le code qui gère une exception spécifique ou toutes les exceptions se produisant dans un bloc de code protégé. Dans la programmation multiplate-forme, vous aurez très rarement besoin d'écrire un gestionnaire d'exception. La majorité des exceptions peuvent être gérées en utilisant les blocs **try..finally** comme décrit dans "Protection des blocs de code" à la page 4-5 et "Protection de l'allocation de ressources" à la page 4-7.

Pour définir un gestionnaire d'exception, incorporez le code à protéger dans un bloc de gestion des exceptions et spécifiez les instructions de gestion des exceptions dans la partie **except** du bloc. Le code suivant est le squelette d'un bloc de gestion des exceptions standard :

```
try
  { instructions à protéger }
except
  { instructions de gestion des exceptions }
end;
```

L'application exécute les instructions de la partie **except** uniquement si une exception se produit lors de l'exécution des instructions placées dans la partie **try**. L'exécution des instructions de la partie **try** inclut également les routines appelées par le code la partie **try**. Cela signifie que si la partie **try** appelle une routine ne définissant pas son propre gestionnaire d'exception, l'exécution revient sur le bloc de gestion des exceptions qui gère l'exception.

Quand une instruction de la partie **try** déclenche une exception, l'exécution passe immédiatement à la partie **except** où elle passe en revue les instructions de gestion d'exception spécifiées ou les gestionnaires d'exceptions, jusqu'à trouver un gestionnaire s'appliquant à l'exception en cours.

Quand l'application a trouvé un gestionnaire d'exception qui gère l'exception, elle exécute l'instruction puis détruit automatiquement l'objet exception. L'exécution reprend ensuite après la fin du bloc en cours.

Instructions de gestion des exceptions

Chaque instruction **on** dans la partie **except** d'un bloc **try..except** définit le code gérant un type particulier d'exception. Les instructions de gestion des exceptions ont la forme suivante :

```
on <type d'exception> do <instruction>;
```

Exemple Vous pouvez ainsi définir un gestionnaire d'exception pour la division par zéro qui définit un résultat par défaut :

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
  try
    Result := Sum div NumberOfItems;{ gère le cas normal }
  except
    on EDivByZero do Result := 0;{ gère l'exception si c'est nécessaire }
  end;
end;
```

Remarquez que cette organisation est plus claire que de placer un test de nullité à chaque appel de la fonction. Voici la même fonction écrite sans tirer profit des exceptions :

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
  if NumberOfItems <> 0 then{ tester systématiquement }
    Result := Sum div NumberOfItems{ utiliser le calcul normal }
  else Result := 0;{ gérer le cas exceptionnel }
end;
```

La différence entre ces deux fonctions résume très bien les différences entre une programmation utilisant les exceptions et une programmation qui ne les utilise pas. Cet exemple est relativement simple mais vous pouvez imaginer des calculs plus complexes faisant intervenir des centaines d'étapes, chacune pouvant échouer si un des paramètres parmi une douzaine est invalide.

En utilisant des exceptions, vous pouvez exprimer la "forme normale" de votre algorithme puis, après, définir les cas exceptionnels pour lesquels elle n'est pas applicable. Sans les exceptions, vous devez effectuer un test à chaque fois pour vous assurer que vous avez bien le droit d'effectuer l'étape suivante du calcul.

Utilisation de l'instance d'exception

La plupart du temps, un gestionnaire d'exception n'a pas besoin d'informations sur l'exception autre que son type, les instructions qui suivent on..do sont donc seulement spécifiques au type de l'exception. Néanmoins, dans certains cas vous avez besoin des informations contenues dans l'instance d'exception.

Pour lire dans un gestionnaire d'exception les informations spécifiques à une instance d'exception, vous devez utiliser une variante particulière de la construction on..do qui vous donne accès à l'instance d'exception. Cette forme spéciale nécessite la spécification d'une variable temporaire utilisée pour stocker l'instance.

Exemple Créez un nouveau projet qui contient une seule fiche, ajoutez-lui une barre de défilement et un bouton de commande. Double-cliquez sur le bouton et définissez le gestionnaire de son événement clic :

```
ScrollBar1.Max := ScrollBar1.Min - 1;
```

Cette ligne déclenche une exception car la valeur maximum de l'étendue d'une barre de défilement doit être supérieure à la valeur minimum. Le gestionnaire d'exception par défaut de l'application ouvre une boîte de dialogue contenant le message de l'objet exception. Vous pouvez redéfinir la gestion de l'exception dans ce gestionnaire d'événement afin de créer votre propre boîte message contenant la chaîne de message de l'exception :

```
try
  ScrollBar1.Max := ScrollBar1.Min - 1;
except
  on E: EInvalidOperation do
    MessageDlg('Ignorer l''exception: ' + E.Message, mtInformation, [mbOK], 0);
end;
```

La variable temporaire (ici E) est du type spécifié après le caractère deux points (*EInvalidOperation* dans cet exemple). Vous pouvez, si nécessaire, utiliser l'opérateur as pour transtyper l'exception dans un type plus spécifique.

Remarque Ne détruisez jamais l'objet exception temporaire. La gestion de l'exception détruit automatiquement l'objet exception. Si vous détruisez l'objet vous-même, l'application tente à nouveau de détruire l'objet, ce qui génère une violation d'accès.

Portée des gestionnaires d'exceptions

Il n'est pas nécessaire de spécifier dans chaque bloc des gestionnaires pour toutes les exceptions imaginables. En fait, vous n'avez besoin que des gestionnaires des exceptions que vous voulez gérer d'une manière particulière dans un bloc donné.

Si un bloc ne gère pas une exception spécifique, l'exécution sort de ce bloc et revient au bloc contenant le bloc (ou revient au code qui a appelé le bloc), et l'exception est toujours déclenchée. Ce processus se répète en augmentant la portée jusqu'à ce que l'exécution atteigne la portée de l'application ou un bloc qui, à un niveau quelconque, gère l'exception.

Spécification du gestionnaire d'exception par défaut

Vous pouvez définir un seul gestionnaire d'exception par défaut qui gère toutes les exceptions n'ayant pas de gestionnaire spécifiquement défini. Pour ce faire, vous devez ajouter une partie else dans la partie **except** du bloc de gestion des exceptions :

```
try
  { instructions }
except
  on ESomething do
    { code de gestion d'exception spécifique };
  else
    { code de gestion d'exception par défaut };
end;
```

L'ajout d'une gestion par défaut des exceptions dans un bloc garantit que ce bloc gère, d'une manière ou d'une autre, toutes les exceptions. Cela redéfinit donc toute gestion effectuée par un bloc conteneur.

Attention Il n'est pas conseillé d'utiliser le gestionnaire d'exception par défaut qui couvre un domaine trop vaste. La clause **else** gère toutes les exceptions, y compris celles dont vous ne connaissez rien. En général, votre code ne doit gérer que les exceptions que vous savez comment gérer. Si vous voulez à la fois "faire le ménage" et réserver la gestion des exceptions à du code disposant de davantage d'informations sur l'exception et la manière de la gérer, utilisez un bloc **try..finally** :

```
try
  try
  { instructions }
  except
```

```
    on ESomething do { code de gestion d'exception spécifique };
  end;
finally
  { code de nettoyage };
end;
```

Pour une autre approche de l'amplification de la gestion des exceptions, Redéclenchement de l'exception.

Gestion des classes d'exceptions

Comme les objets exception font partie d'une hiérarchie, vous pouvez spécifier des gestionnaires pour toute une partie de la hiérarchie en spécifiant un gestionnaire pour la classe d'exception dont dérive cette partie de la hiérarchie.

Exemple Le bloc suivant est le squelette d'un exemple gérant toutes les exceptions de calcul entier de manière spécifique :

```
try
  { instructions effectuant des opérations de calcul entier }
except
  on EIntError do { gestion spéciale des erreurs de calcul entier };
end;
```

Vous pouvez toujours définir des gestionnaires plus spécifiques pour des exceptions plus spécifiques. Vous devez juste placer les gestionnaires spécifiques avant le gestionnaire générique car l'application recherche les gestionnaires dans leur ordre d'apparition et exécute le premier gestionnaire applicable trouvé. Par exemple, le bloc suivant définit une gestion spécifique des erreurs d'étendue et un autre gestionnaire pour toutes les autres erreurs de calcul entier :

```
try
  { instructions effectuant des opérations de calcul entier }
except
  on ERangeError do { gestion des calculs hors étendue };
  on EIntError do { gestion des autres erreurs de calcul entier };
end;
```

Par contre, si le gestionnaire de *EIntError* est placé avant le gestionnaire de *ERangeError*, l'exécution n'atteint jamais le gestionnaire spécifique à *ERangeError*.

Redéclenchement de l'exception

Parfois, quand vous gérez localement une exception, vous voulez juste étendre la gestion définie par le bloc conteneur et pas la remplacer. Mais, bien entendu, quand votre gestionnaire local en a fini avec l'exception, il détruit automatiquement l'instance d'exception et le gestionnaire du bloc conteneur ne peut donc pas agir dessus. Vous pouvez néanmoins empêcher le gestionnaire de détruire l'exception, ce qui laisse au gestionnaire du conteneur l'opportunité d'y répondre.

Exemple Quand une exception se produit, vous voulez afficher un message à l'intention de l'utilisateur ou enregistrer l'erreur dans un fichier historique, puis laisser faire la gestion standard. Pour ce faire, déclarez un gestionnaire local de l'exception

qui affiche le message puis utilise le mot réservé raise. C'est ce que l'on appelle redéclencher l'exception, comme le montre le code suivant :

```
try
  { instructions }
  try
    { instructions spéciales }
  except
    on ESomething do
    begin
      { ne gère que les instructions spéciales }
      raise;{ redéclenche l'exception }
    end;
  end;
except
  on ESomething do ...;{ gestion à effectuer dans tous les cas }
end;
```

Si le code de la partie { instructions } déclenche une exception *ESomething*, seul le gestionnaire de la partie **except** extérieure s'exécute. Par contre, si c'est le code de la partie { instructions spéciales } qui déclenche une exception *ESomething*, la gestion définie dans la partie **except** intérieure est exécutée suivie par celle, plus générale, de la partie **except** extérieure.

En redéclenchant des exceptions, vous pouvez facilement définir une gestion spécifique d'exceptions pour des cas particuliers sans perdre (ou sans dupliquer) les gestionnaires existants.

Gestion des exceptions des composants

Les composants de Delphi déclenchent des exceptions pour indiquer des conditions d'erreur. La plupart des exceptions de composant signalent des erreurs de programmation qui sinon généreraient une erreur d'exécution. La technique pour gérer les exceptions de composants n'est pas différente de celle utilisée pour les exceptions RTL.

Exemple Les erreurs d'intervalles dans les propriétés indicées sont une source fréquente d'erreur dans les composants. Si, par exemple, pour une boîte liste dont la liste contient trois éléments (0..2), votre application tente d'accéder à l'élément numéro 3, la boîte liste déclenche une exception "Indice de liste hors limites".

Le gestionnaire d'événement suivant contient un gestionnaire d'exception qui informe l'utilisateur de l'accès à un indice invalide de la boîte liste :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add('une chaîne');{ ajoute une chaîne à la boîte liste }
  ListBox1.Items.Add('une autre chaîne');{ ajoute une autre chaîne... }
  ListBox1.Items.Add('encore une autre chaîne');{ ...et une troisième chaîne}
  try
    Caption := ListBox1.Items[3];{ Affecte la quatrième chaîne de la boîte liste à
l'intitulé de la fiche }
  except
    on EStringListError do
```

```
        MessageDlg('La boîte liste contient moins de quatre chaînes', mtWarning, [mbOK], 0);
    end;
  end;
```

Si vous cliquez sur le bouton, comme la boîte liste ne contient que trois chaînes, l'accès à la quatrième chaîne (Items[3]) déclenche une exception. Si vous cliquez une seconde fois sur le bouton, d'autres chaînes sont ajoutées à la liste et l'exception n'est donc plus déclenchée.

Gestion des exceptions et sources externes

HandleException propose une gestion par défaut des exceptions au niveau de l'application. Normalement, quand vous développez des applications multiplates-formes vous n'avez pas besoin d'appeler *TApplication.HandleException*. Cependant, vous risquez d'en avoir besoin lorsque vous écrivez des fichiers d'objets partagés ou des fonctions callback. Vous pouvez utiliser *TApplication.HandleException* pour empêcher une exception de sortir de votre code, en particulier lorsque le code est appelé à partir d'une source externe ne supportant pas les exceptions.

Par exemple, si une exception passe au travers de tous les blocs **try** du code de l'application, l'application appelle automatiquement la méthode *HandleException* qui affiche une boîte de dialogue indiquant qu'une erreur a eu lieu. Vous pouvez utiliser *HandleException* de la manière suivante :

```
try
  { instructions }
except
  Application.HandleException(Self);
end;
```

Pour toutes les exceptions sauf *EAbort*, *HandleException* appelle, s'il existe, le gestionnaire d'événement *OnException*. Si vous voulez à la fois gérer l'exception et proposer ce comportement par défaut, comme les composants intégrés, ajoutez un appel de *HandleException* à votre code :

```
try
  { instructions spéciales }
except
  on ESomething do
  begin
    { ne gère que les instructions spéciales }
    Application.HandleException(Self);{ appelle HandleException }
  end;
end;
```

Remarque Vous ne devez pas appeler *HandleException* depuis le code de gestion des exceptions d'un thread.

Pour plus d'informations, recherchez routines de gestion des exceptions dans l'index de l'aide.

Exceptions silencieuses

Les applications Delphi gèrent la plupart des exceptions qui ne sont pas gérées spécifiquement dans votre code en affichant une boîte de message qui affiche la chaîne de message de l'objet exception. Vous pouvez également définir des exceptions "silencieuses" pour lesquelles, par défaut l'application n'affiche pas le message d'erreur.

Les exceptions silencieuses sont utiles quand vous ne voulez pas signaler l'exception à l'utilisateur, mais simplement abandonner l'opération. L'abandon d'une opération est semblable à l'utilisation des procédures *Break* et *Exit* pour sortir d'un bloc, mais elle permet de sortir de plusieurs niveaux de blocs imbriqués.

Les exceptions silencieuses descendent toutes du type d'exception standard *EAbort*. Le gestionnaire d'exception par défaut des applications VCL et CLX Delphi affiche la boîte de dialogue de message d'erreur pour toutes les exceptions qu'il reçoit sauf pour celles qui dérivent de *EAbort*.

Remarque Dans les applications console, la boîte de dialogue d'erreur est affichée pour toutes les exceptions *EAbort* non gérées.

Il y a un moyen rapide de déclencher des exceptions silencieuses : au lieu de construire l'objet manuellement, vous pouvez appeler la procédure *Abort*. *Abort* déclenche automatiquement une exception *EAbort* qui sort de l'opération en cours sans afficher de message d'erreur.

Exemple L'exemple suivant propose un exemple simple d'abandon d'une opération. Dans une fiche contenant une boîte liste vide et un bouton, attachez le code suivant à l'événement *OnClick* du bouton :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 10 do{ boucler dix fois }
  begin
    ListBox1.Items.Add(IntToStr(I));{ ajouter un nombre à la liste }
    if I = 7 then Abort;{ arrêter au septième }
  end;
end;
```

Définition d'exceptions personnalisées

Non seulement les exceptions vous permettent de protéger votre code des erreurs générées par la bibliothèque d'exécution ou par les composants, mais vous pouvez également utiliser le même mécanisme pour gérer des conditions exceptionnelles dans votre propre code.

Pour utiliser des exceptions dans votre code, vous devez suivre ces étapes :

- Déclaration d'un type objet exception
- Déclenchement d'une exception

Déclaration d'un type objet exception

Comme les exceptions sont des objets, définir un nouveau type d'exception est aussi simple que déclarer un nouveau type d'objet. Bien que vous puissiez déclencher toute instance d'objet comme une exception, les gestionnaires d'exception standard ne gèrent que les exceptions qui descendent de Exception.

Par convention, les nouveaux types d'exception doivent être dérivés de *Exception* ou de l'une des autres exceptions standard. De cette manière, si vous déclenchez votre nouvelle exception dans un bloc de code qui n'est pas protégé par un gestionnaire spécifique à cette exception, l'un des gestionnaires standard la gérera.

Exemple Par exemple, examinez la déclaration suivante :

```
type
  EMyException = class(Exception);
```

Si vous déclenchez *EMyException* sans spécifier pour elle de gestionnaire spécifique, un gestionnaire de *Exception* (ou un gestionnaire d'exception par défaut) pourra la gérer. Comme la gestion standard pour *Exception* affiche le nom de l'exception déclenchée, vous pourrez voir que c'est votre nouvelle exception qui a été déclenchée.

Déclenchement d'une exception

Pour indiquer une condition d'erreur paralysante dans une application, vous pouvez déclencher une exception, ce qui implique la construction d'une instance de ce type et l'appel du mot réservé **raise**.

Pour déclencher une exception, appelez le mot réservé raise en le faisant suivre par une instance d'un objet exception. Cela vous permet d'établir une exception comme issue d'une adresse particulière. Quand un gestionnaire d'exception gère effectivement l'exception, il se termine en détruisant l'instance d'exception : vous n'avez donc jamais à le faire vous-même.

Le déclenchement d'une exception définit dans l'unité *System* la variable *ErrorAddr* par l'adresse à laquelle l'application a déclenché l'exception. Vous pouvez faire référence à *ErrorAddr* dans vos gestionnaires d'exceptions, par exemple pour informer l'utilisateur de l'emplacement de l'erreur. Vous pouvez aussi spécifier dans la clause raise la valeur qui apparaîtra dans *ErrorAddr* au moment où se produit l'exception.

Attention N'attribuez pas vous-même la valeur de *ErrorAddr*. Elle est prévue pour être en lecture seule.

Pour spécifier l'adresse de l'erreur d'une exception, ajoutez le mot réservé **at** après l'instance d'exception en la faisant suivre d'une expression adresse, par exemple un identificateur.

Par exemple, étant donné la déclaration suivante :

```
type
  EPasswordInvalid = class(Exception);
```

vous pouvez déclencher une exception “mot de passe incorrect” à tout moment en appelant raise avec une instance de *EPasswordInvalid*, comme suit :

```
if Password <> CorrectPassword then
  raise EPasswordInvalid.Create('Mot de passe saisi incorrect');
```

Utilisation des interfaces

Le mot réservé **interface** de Delphi vous permet de créer et d'utiliser des interfaces dans votre application. Les interfaces constituent un moyen d'étendre le modèle d'héritage simple de Pascal Objet en permettant à une même classe d'implémenter plusieurs interfaces et à plusieurs classes n'ayant pas le même ancêtre de partager la même interface. Les interfaces sont utiles quand les mêmes ensembles d'opérations, par exemple la manipulation de flux, portent sur une gamme variée d'objets. Les interfaces sont également un aspect fondamental des modèles d'objets distribués COM (Component Object Model) et CORBA (Common Object Request Broker Architecture).

Interfaces en tant que caractéristiques du langage

Une interface est semblable à une classe ne contenant que des méthodes abstraites et une définition claire de ses fonctionnalités. Strictement parlant, les définitions des méthodes d'interface spécifient le nombre et le type de leurs paramètres, le type renvoyé et le comportement prévu. Les méthodes d'une interface sont nommées de façon à indiquer le rôle de l'interface. Par convention, les interfaces sont nommées en fonction de leur comportement en préfixant leur nom par une lettre *I* en majuscule. Par exemple, une interface *IMalloc* doit allouer, libérer et gérer de la mémoire. De même, une interface *IPersist* peut être utilisée comme une interface de base générale pour des descendants, chacun d'eux définissant des prototypes de méthode spécifiques permettant de charger et d'enregistrer l'état d'un objet dans un stockage, un flux ou un fichier.

Une interface utilise la syntaxe suivante :

```
IMyObject = interface
  procedure MyProcedure;
end;
```

Voici un exemple simple de déclaration d'une interface :

```
type
IEdit = interface
  procedure Copy; stdcall;
  procedure Cut; stdcall;
  procedure Paste; stdcall;
  function Undo: Boolean; stdcall;
end;
```

Comme les classes abstraites, les interfaces ne sont jamais instanciées elles-mêmes. Pour utiliser une interface, vous devez l'obtenir en l'implémentant dans une classe.

Pour implémenter une interface, vous devez définir une classe qui déclare l'interface dans sa liste d'ancêtres, ce qui indique qu'elle implémente toutes les méthodes de l'interface :

```
TEditor = class(TInterfacedObject, IEdit)
  procedure Copy; stdcall;
  procedure Cut; stdcall;
  procedure Paste; stdcall;
  function Undo: Boolean; stdcall;
end;
```

Alors que les interfaces définissent le comportement et la signature de leurs méthodes, elles n'en définissent pas l'implémentation. Dès lors que l'implémentation faite dans la classe se conforme à la définition de l'interface, l'interface est totalement polymorphique : l'accès et l'utilisation de l'interface restent identiques dans toutes ses implémentations.

Implémentation des interfaces au travers de la hiérarchie

L'utilisation d'interfaces permet d'envisager une conception qui sépare la manière d'utiliser une classe de la manière dont elle est implémentée. Deux classes peuvent implémenter la même interface sans descendre nécessairement de la même classe de base. Cet appel polymorphique de la même méthode pour des objets sans rapport entre eux est possible dans la mesure où les objets implémentent la même interface. Par exemple, soit l'interface :

```
IPaint = interface
  procedure Paint;
end;
```

et les deux classes,

```
TSquare = class(TPolygonObject, IPaint)
  procedure Paint;
end;

TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Que ces deux classes aient ou non un ancêtre commun, elles sont toujours compatibles pour l'affectation avec une variable de type *IPaint* :

```
var
  Painter: IPaint;
begin
  Painter := TSquare.Create;
  Painter.Paint;
  Painter := TCircle.Create;
  Painter.Paint;
end;
```

Il est possible d'obtenir le même résultat en faisant dériver *TCircle* et *TSquare* d'une classe *TFigure* qui implémente la méthode virtuelle *Paint*. Dans ce cas *TCircle* et *TSquare* doivent surcharger la méthode *Paint*. *IPaint* est alors remplacée par *TFigure*. Cependant, considérez l'interface suivante :

```
IRotate = interface
  procedure Rotate(Degrees: Integer);
end;
```

qui a du sens pour un rectangle, mais pas pour le cercle. Les classes seraient alors définies de la manière suivante :

```
TSquare = class(TRectangularObject, IPaint, IRotate)
  procedure Paint;
  procedure Rotate(Degrees: Integer);
end;

TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Vous pouvez, ultérieurement créer une classe *TFilledCircle* qui implémente l'interface *IRotate* afin de permettre la rotation du motif utilisé pour remplir le cercle sans avoir à ajouter la rotation au cercle simple.

Remarque Dans ces exemples, on suppose que la classe de base immédiate ou une classe ancêtre a implémenté les méthodes de *IInterface* qui gèrent le comptage de références. Pour plus d'informations, voir "Implémentation de IInterface" à la page 4-22 et "Gestion mémoire des objets interface" à la page 4-26.

Utilisation d'interfaces avec des procédures

Les interfaces permettent également d'écrire des procédures génériques pouvant gérer des objets sans que ces objets descendent d'une classe de base particulière. En utilisant les interfaces *IPaint* et *IRotate* définies précédemment, vous pouvez écrire les procédures suivantes :

```
procedure PaintObjects(Painters: array of IPaint);
var
  I: Integer;
begin
  for I := Low(Painters) to High(Painters) do
    Painters[I].Paint;
end;

procedure RotateObjects(Degrees: Integer; Rotaters: array of IRotate);
var
  I: Integer;
begin
  for I := Low(Rotaters) to High(Rotaters) do
    Rotaters[I].Rotate(Degrees);
end;
```

RotateObjects n'a pas besoin que les objets sachent se dessiner par eux-mêmes et *PaintObjects* n'exige pas que les objets sachent comment pivoter. Cela permet aux procédures génériques précédentes d'être utilisées plus fréquemment que si elles avaient été écrites uniquement pour la classe *TFigure*.

Pour des détails sur la syntaxe, la définition du langage et les règles s'appliquant aux interfaces, voir dans le *Guide du langage Pascal Objet* en ligne, la section Interfaces d'objet.

Implémentation de IInterface

Toutes les interfaces dérivent, directement ou non, de l'interface *IInterface*. Cette interface définit les fonctionnalités essentielles d'une interface, c'est-à-dire l'interrogation dynamique et la gestion de la durée de vie. Ces fonctionnalités sont mises en place par les trois méthodes de *IInterface* :

- *QueryInterface* est une méthode d'interrogation dynamique d'un objet donné qui obtient les références des interfaces gérées par l'objet.
- *_AddRef* est une méthode de comptage de références qui incrémente le compteur à chaque appel réussi de *QueryInterface*. Tant que le compteur de références est non nul, l'objet doit rester en mémoire.
- *_Release* est utilisée avec *_AddRef* pour permettre à un objet de connaître sa durée de vie et de déterminer s'il peut se supprimer lui-même. Quand le compteur de références atteint zéro, l'objet est libéré de la mémoire.

Chaque classe qui implémente des interfaces doit implémenter les trois méthodes de *IInterface,* les autres méthodes déclarées dans toutes ses interfaces ancêtre, ainsi que toutes les méthodes déclarées dans l'interface même. Néanmoins, vous pouvez hériter de l'implémentation des méthodes d'interface déclarées dans votre classe.

En implémentant vous-même ces méthodes, vous bénéficiez d'un moyen supplémentaire de gestion de la durée de vie désactivant le mécanisme de comptage de références. C'est une technique puissante qui permet de découpler les interfaces du comptage de références.

TInterfacedObject

Delphi définit une classe simple, *TInterfacedObject* qui, pratiquement, sert de classe de base car elle implémente les méthodes de *IInterface*. *La classe TInterfacedObject* est déclarée de la manière suivante dans l'unité *System* :

```
type
  TInterfacedObject = class(TObject, IInterface)
  protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
  end;
```

Dériver directement de *TInterfacedObject* est trivial. Dans l'exemple de déclaration suivant, *TDerived* est un descendant direct de *TInterfacedObject* qui implémente une interface hypothétique, *IPaint* :

```
type
  TDerived = class(TInterfacedObject, IPaint)
    ...
  end;
```

Comme *TInterfacedObject implémente* les méthodes de *IInterface*, elle gère automatiquement le comptage de références et la gestion mémoire pour les objets interfacés. Pour davantage d'informations, voir "Gestion mémoire des objets interface" à la page 4-26, qui traite également de l'écriture de classes implémentant des interfaces sans utiliser le mécanisme de comptage de références inhérent à *TInterfacedObject*.

Utilisation de l'opérateur as

Les classes implémentant des interfaces peuvent utiliser l'opérateur **as** pour se lier dynamiquement à l'interface. Dans l'exemple suivant :

```
procedure PaintObjects(P: TInterfacedObject)
var
  X: IPaint;

begin
  X := P as IPaint;
{ instructions }
end;
```

la variable *P*, de type *TInterfacedObject*, peut être affectée à la variable *X* qui est une référence à l'interface *IPaint*. La liaison dynamique rend cette affectation possible. Pour cette affectation, le compilateur génère du code qui appelle la méthode *QueryInterface* de l'interface *IInterface* de *P*. C'est parce que le compilateur ne peut déduire du type déclaré de *P* si l'instance de *P* supporte *IPaint*. A l'exécution, *soit P* se résout en une référence à une interface *IPaint*, soit une exception est déclenchée. Dans l'un ou l'autre cas, l'affectation de *P* à *X* ne génère pas une erreur à la compilation comme se serait le cas si *P* avait pour type une classe n'implémentant pas *IInterface*.

Quand vous utilisez l'opérateur **as** pour une liaison dynamique avec une interface, vous devez respecter les règles suivantes :

- Déclaration explicite de *IInterface* : même si toutes les interfaces dérivent de *IInterface*, il ne suffit pas qu'une classe implémente les méthodes de *IInterface pour pouvoir* utiliser l'opérateur **as**. Cela reste vrai même si la classe implémente également les interfaces qu'elle déclare explicitement. La classe doit déclarer explicitement *IInterface* dans sa liste d'interfaces.
- Utilisation d'un IID : les interfaces peuvent utiliser un identificateur basé sur un identificateur global unique (GUID). Les GUID utilisés pour identifier des interfaces sont appelés des identificateurs d'interface (IID). Si vous utilisez l'opérateur **as** avec une interface, elle doit avoir un IID associé. Pour créer un

nouveau GUID dans votre code source, vous pouvez utiliser le raccourci *Ctrl+Maj+G* de l'éditeur de code.

Réutilisation de code et délégation

Une des manières pour réutiliser du code avec les interfaces consiste à utiliser un objet contenant un autre objet ou un objet contenu dans un autre objet. L'utilisation de propriétés de type objet est un moyen de contenir et de réutiliser du code. Pour exploiter cette caractéristique des interfaces, Pascal Objet emploie le mot clé **implements** qui rend facile l'écriture de code pour déléguer tout ou partie de l'implémentation d'une interface à un sous-objet. L'agrégation est un autre moyen de réutiliser le code par contenance et par délégation. Avec l'agrégation, un objet externe contient un objet interne qui implémente des interfaces qui ne sont exposées que par l'objet externe. La VCL et CLX possèdent des classes qui supportent l'agrégation.

Utilisation de implements pour la délégation

De nombreuses classes ont des propriétés qui sont des sous-objets. Vous pouvez également utiliser des interfaces comme type d'une propriété. Quand une propriété est de type interface (ou d'un type de classe qui implémente les méthodes d'une interface), vous pouvez utiliser le mot clé **implements** pour spécifier que les méthodes de cette interface sont déléguées à la référence d'objet ou d'interface qui est l'instance de la propriété. Le délégué doit seulement fournir l'implémentation des méthodes. il n'a pas besoin de déclarer qu'il gère l'interface. La classe contenant la propriété doit inclure l'interface dans sa liste d'ancêtres.

Par défaut, l'utilisation du mot clé **implements** délègue toutes les méthodes d'interface. Vous pouvez néanmoins utiliser les clauses de résolution des méthodes ou déclarer des méthodes dans votre classe qui implémentent certaines des méthodes de l'interface comme moyen de surcharger ce comportement par défaut.

L'exemple suivant utilise le mot clé implements dans la conception d'un objet adaptateur de couleurs qui convertit une valeur de couleur RVB sur 8 bits en une référence *Color* :

```
unit cadapt;

type
IRGB8bit = interface
    ['{1d76360a-f4f5-11d1-87d4-00c04fb17199}']
    function Red: Byte;
    function Green: Byte;
    function Blue: Byte;
  end;

IColorRef = interface
    ['{1d76360b-f4f5-11d1-87d4-00c04fb17199}']
    function Color: Integer;
  end;
```

```
{ TRGB8ColorRefAdapter associe un IRGB8bit à un IColorRef }
  TRGB8ColorRefAdapter = class(TInterfacedObject, IRGB8bit, IColorRef)
  private
    FRGB8bit: IRGB8bit;
    FPalRelative: Boolean;
  public
    constructor Create(rgb: IRGB8bit);
    property RGB8Intf: IRGB8bit read FRGB8bit implements IRGB8bit;
    property PalRelative: Boolean read FPalRelative write FPalRelative;
    function Color: Integer;
  end;

implementation

constructor TRGB8ColorRefAdapter.Create(rgb: IRGB8bit);
begin
  FRGB8bit := rgb;
end;

function TRGB8ColorRefAdapter.Color: Integer;
begin
  if FPalRelative then
    Result := PaletteRGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue)
  else
    Result := RGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue);
end;
end.
```

Pour davantage d'informations sur la syntaxe, les détails de l'implémentation et les règles concernant le mot clé **implements**, voir dans l'aide en ligne du *Guide du Langage Pascal Objet*, la rubrique Interfaces d'objet.

Agrégation

L'agrégation offre une approche modulaire pour réutiliser du code via des sous-objets qui définissent la fonctionnalité d'un objet conteneur, mais qui cachent les détails de l'implémentation de cet objet. Dans l'agrégation, un objet externe implémente une ou plusieurs interfaces. La seule exigence est qu'il implémente *IInterface*. L'objet ou les objets internes peuvent implémenter une ou plusieurs interfaces, mais seul l'objet externe expose les interfaces. Cela vaut à la fois pour les interfaces qu'il implémente et pour celles qui sont implémentées par les objets qu'il contient. Les clients ne savent rien des objets internes. Bien que l'objet externe fournisse l'accès aux interfaces de l'objet interne, leur implémentation est complètement transparente. Donc, la classe de l'objet externe peut échanger le type de classe de l'objet interne avec toute classe qui implémente la même interface. De même, le code des classes objet interne peut être partagé par d'autres classes qui veulent l'utiliser.

Le modèle d'implémentation de l'agrégation définit explicitement les règles pour implémenter *IInterface* en utilisant la délégation. L'objet interne doit implémenter une *IInterface* sur lui-même, qui contrôle le comptage de références de l'objet interne. Cette implémentation de *IInterface* surveille la relation entre l'objet externe et l'objet interne. Par exemple, quand un objet de ce type (l'objet interne) est créé, la création ne réussit que pour une interface demandée de type

IInterface. L'objet interne implémente également une deuxième *IInterface* pour toutes les interfaces qu'il implémente. Ce sont les interfaces exposées par l'objet externe. Cette deuxième *IInterface* délègue les appels de *QueryInterface*, *AddRef* et *Release* à l'objet externe. L'interface *IInterface* externe est appelée le "controlling Unknown".

Reportez-vous à l'aide en ligne de MicroSoft pour connaître les règles de création d'une agrégation. Quand vous écrivez vos propres classes d'agrégation, vous pouvez aussi vous référer aux détails de l'implémentation de *IInterface* dans *TComObject*. *TComObject* est une classe COM qui supporte l'agrégation. Si vous écrivez des applications COM, vous pouvez aussi utiliser directement *TComObject* comme une classe de base.

Gestion mémoire des objets interface

L'un des concepts clé de la conception d'interfaces est la gestion de la durée de vie des objets qui les implémentent. Les méthodes *_AddRef* et *_Release* de *IInterface* constituent un moyen d'implémenter la gestion de la durée de vie. *_AddRef* et *_Release* surveillent la durée de vie d'un objet en incrémentant le compteur de références à l'objet quand une référence d'interface est transmise à un client et qu'elles détruisent l'objet quand le compteur de références est nul.

Si vous créez des objets COM pour des applications distribuées (en environnement Windows seulement), alors vous devez adhérer strictement aux règles de comptage des références. Mais, si vous utilisez des interfaces dans votre application uniquement en interne, alors vous avez un choix qui dépend de la nature de votre objet et de la façon dont vous avez décidé de l'utiliser.

Utilisation du comptage de références

Delphi vous fournit l'essentiel de la gestion mémoire *IInterface* grâce à son implémentation de l'interrogation et du comptage de références de l'interface. Cependant, si vous utilisez un objet qui vit et meurt via ses interfaces, vous pouvez aisément utiliser le comptage de références en dérivant de ces classes. *TInterfacedObject* est la non coclasse qui définit ce comportement. Si vous choisissez d'utiliser le comptage de références, vous devez faire attention à ne manipuler l'objet que sous la forme d'une référence d'interface et à être cohérent dans votre comptage de références. Par exemple :

```
procedure beep(x: ITest);

function test_func()
var
  y: ITest;
begin
  y := TTest.Create; // comme y est de type ITest, le compteur de références vaut 1
  beep(y); // l'appel de la fonction beep incrémente le compteur de références
  // et le décrémente à son retour
  y.something; // l'objet est toujours là avec un compteur de références valant 1
end;
```

C'est la manière la plus claire et la plus prudente de gérer la mémoire et, si vous utilisez *TInterfacedObject*, elle est utilisée automatiquement. Si vous ne respectez pas ces règles, votre objet peut disparaître inopinément, comme l'illustre le code suivant :

```
function test_func()
var
  x: TTest;
begin
  x := TTest.Create; // pas encore de compteur de références pour l'objet
  beep(x as ITest); // le compteur est incrémenté par l'appel de beep
  // et décrémenté à son retour
  x.something; // surprise ! l'objet n'est plus là
end;
```

Remarque Dans les exemples précédents, la procédure *beep*, telle qu'elle est déclarée, incrémente le compteur de références (appel de *_AddRef*) pour le paramètre. Par contre, les déclarations suivantes ne le font pas :

```
procedure beep(const x: ITest);
```

ou

```
procedure beep(var x: ITest);
```

Ces déclarations génèrent un code plus concis et plus rapide.

Vous ne pouvez pas utiliser le comptage de références dans un cas : si votre objet est un composant ou un contrôle contenu dans un autre composant. Dans un tel cas, le comptage de références ne peut être appliqué de manière cohérente : vous pouvez toujours utiliser les interfaces, mais sans utiliser le comptage de référence car la durée de vie de l'objet n'est pas régie par ses interfaces.

Situations où il ne faut pas utiliser le comptage de références

Si votre objet est un composant ou un contrôle détenu par un autre composant, votre objet utilise alors un autre système de gestion mémoire ayant son origine dans *TComponent*. Vous ne devez pas mélanger l'approche de la durée de vie des objets utilisée par les composants VCL ou CLX avec le système de comptage de références. Si vous voulez créer un objet qui gère les interfaces, vous pouvez définir une implémentation vide des méthodes *_AddRef* et *_Release* de *IInterface* afin de court-circuiter le mécanisme de comptage de références. Par exemple :

```
function TMyObject._AddRef: Integer;
begin
  Result := -1;
end;

function TMyObject._Release: Integer;
begin
  Result := -1;
end;
```

Vous devez quand même implémenter normalement *QueryInterface* afin de permettre l'interrogation dynamique de votre objet.

Comme vous implémentez *QueryInterface*, vous pouvez toujours utiliser l'opérateur **as** pour des interfaces de composant dans la mesure où vous créez un identificateur d'interface (IID). Vous pouvez également utiliser l'agrégation. Si l'objet externe est un composant, l'objet interne implémente le comptage de références comme d'habitude, en déléguant au "controlling Unknown." C'est au niveau de l'objet composant externe que la décision est prise de circonvenir les méthodes *_AddRef* et *_Release* et de prendre en charge la gestion de la mémoire par une approche basée sur les composants. En fait, vous pouvez utiliser *TInterfacedObject* comme classe de base pour l'objet interne d'une agrégation qui a un composant comme objet externe contenant.

Remarque Le "controlling Unknown" est l'interface *IUnknown* implémentée par l'objet externe et pour laquelle le comptage de références de l'objet entier est tenu à jour. *IUnknown* est identique à *IInterface*, mais est utilisée à la place dans les applications basées sur COM (Windows seulement). Pour plus d'informations sur les diverses implémentations de *IUnknown* ou de *IInterface* par les objets internes ou externes, voir "Agrégation" à la page 25 et les rubriques d'aide en ligne de Microsoft sur le "controlling Unknown."

Utilisation des interfaces dans les applications distribuées (VCL seulement)

Les interfaces sont des éléments fondamentaux des modèles d'objets distribués COM, SOAP et CORBA. Delphi fournit des classes de base pour les technologies qui étendent la fonctionnalité d'interface de base de *TInterfacedObject*, qui implémente simplement les méthodes de l'interface *IInterface*.

Quand on utilise COM, les classes et les interfaces sont définies en termes de *IUnknown* plutôt que *IInterface*. Il n'y a pas de différence sémantique entre *IUnknown* et *IInterface* ; l'utilisation de *IUnknown* est simplement un moyen d'adapter les interfaces Delphi à la définition COM. Les classes COM ajoutent la fonction d'utilisation de fabriques et d'identificateurs de classes (CLSID). Les fabriques de classes sont responsables de la création des instances de classes via les CLSID. Les CLSID sont utilisés pour recenser et manipuler des classes COM. Les classes COM qui ont des fabriques et des identificateurs de classes sont appelées des CoClasses. Les CoClasses tirent profit des capacités de gestion des versions de *QueryInterface*, de sorte que lorsqu'un module logiciel est mis à jour, *QueryInterface* peut être appelée à l'exécution pour connaître les capacités actuelles d'un objet.

De nouvelles versions d'anciennes interfaces, ainsi que de nouvelles interfaces ou fonctionnalités d'un objet, peuvent devenir disponibles aux nouveaux clients de façon immédiate. En même temps, les objets gardent l'entière compatibilité avec le code client existant ; aucun recompilation n'est requise puisque les implémentations des interfaces sont cachées (les méthodes et les paramètres restent constants). Dans les applications COM, les développeurs peuvent changer l'implémentation pour améliorer les performances ou pour toute autre raison interne, sans perdre le code client basé sur cette interface. Pour plus d'informations sur les interfaces COM, voir chapitre 33, "Présentation des technologies COM".

Quand vous distribuez une application en utilisant SOAP, les interfaces sont requises pour transporter leurs propres informations de type à l'exécution (RTTI). Le compilateur ajoute les informations RTTI à une interface uniquement lorsqu'elle est compilée en utilisant le commutateur {$M+}. De telles interfaces sont appelées *interfaces invocables*. Le descendant de toute interface invocable est également invocable. Cependant, si une interface invocable descend d'une autre interface qui n'est pas invocable, les applications client peuvent appeler uniquement les méthodes définies dans l'interface invocable et dans ses descendants. Les méthodes héritées d'ancêtres non invocables ne sont pas compilées avec les informations de type et donc ne peuvent être appelées par les clients.

Le moyen le plus simple de définir des interfaces invocables est de définir votre interface de sorte qu'elle descende de *IInvokable*. *IInvokable* est identique à *IInterface*, sauf qu'elle est compilée en utilisant le commutateur {$M+}. Pour davantage d'informations sur les applications Web Service qui sont distribuées en utilisant SOAP, et sur les interfaces invocables, voir chapitre 31, "Utilisation de services Web".

CORBA est une autre technologie d'applications distribuées. L'utilisation d'interfaces dans les applications CORBA se fait par le biais de classes stubs sur le client et de classes squelettes sur le serveur. Ces classes stubs et squelettes gèrent les détails du marshaling des appels d'interfaces pour que les valeurs des paramètres et les valeurs de retour soient correctement transmises. Les applications doivent utiliser une classe stub ou squelette ou employer la DII (Dynamic Invocation Interface) qui convertit tous les paramètres en variants spéciaux (de sorte qu'elles transportent leurs propres informations de type).

Définition de variants personnalisés

Le type Variant est un type puissant intégré au langage Pascal Objet. Les variants représentent des valeurs dont le type n'est pas déterminé au moment de la compilation. Au contraire, le type de valeur peut changer à l'exécution. Les variants peuvent se combiner à d'autres variants ainsi qu'à des valeurs entières, réelles, chaînes et booléennes dans les expressions et les affectations ; le compilateur effectue automatiquement les conversions de type.

Par défaut, les variants peuvent contenir des valeurs qui sont des enregistrements, des ensembles, des tableaux statiques, des fichiers, des classes, des références de classes ou des pointeurs. Mais, vous pouvez étendre le type Variant afin de travailler avec tout exemple particulier de ces types. Il suffit de créer un descendant de la classe *TCustomVariantType* qui indique comment le type Variant effectue les opérations standard.

Pour créer un a type Variant,

1 Etablissez une correspondance entre le stockage des données du variant et l'enregistrement *TVarData*.

2. Déclarez une classe qui dérive de *TCustomVariantType*. Implémentez tous les comportements requis (y compris les règles de conversion de type) dans la nouvelle classe.
3. Ecrivez les méthodes utilitaires permettant de créer les instances de votre variant personnalisé et reconnaître son type.

Les étapes ci-dessus étendent le type Variant de sorte que les opérateurs standard puissent fonctionner avec le nouveau type et que ce dernier puisse être convertit en d'autres types de données. Vous pourrez ultérieurement améliorer votre nouveau type Variant pour qu'il supporte les propriétés et les méthodes que vous définissez. Quand vous créez un type Variant qui supporte les propriétés ou les méthodes, utilisez *TInvokeableVariantType* ou *TPublishableVariantType* comme classe de base plutôt que *TCustomVariantType*.

Stockage des données d'un type Variant personnalisé

Les variants stockent leurs données dans le type enregistrement *TVarData*. Ce type est un enregistrement contenant 16 octets. Le premier mot indique le type du variant, et les 14 octets restants sont disponibles pour le stockage des données. Bien que votre nouveau type Variant puisse fonctionner directement avec un enregistrement *TVarData*, il est en général plus simple de définir un type enregistrement dont les membres portent des noms significatifs du nouveau type puis de convertir ce nouveau type en type enregistrement *TVarData*.

Par exemple, l'unité VarConv définit un type variant personnalisé qui représente une mesure. Les données de ce type contiennent les unités (*TConvType*) de mesure, ainsi que la valeur (un double). L'unité VarConv définit son propre type pour représenter cette valeur :

```
TConvertVarData = packed record
  VType: TVarType;
  VConvType: TConvType;
  Reserved1, Reserved2: Word;
  VValue: Double;
end;
```

Ce type est exactement de la même taille que l'enregistrement *TVarData*. Quand vous travaillez avec un variant personnalisé du nouveau type, le variant (ou son enregistrement *TVarData*) peut être convertit en *TConvertVarData*, et le type Variant personnalisé fonctionnera simplement avec l'enregistrement *TVarData* comme s'il était du type *TConvertVarData*.

Remarque Quand vous définissez un enregistrement qui correspond à l'enregistrement *TVarData*, assurez-vous de le définir compressé.

Si votre nouveau type Variant personnalisé a besoin de plus de 14 octets pour stocker les données, vous pouvez définir un nouveau type enregistrement qui contienne un pointeur ou l'instance d'un objet. Par exemple, l'unité VarCmplx utilise une instance de la classe *TComplexData* pour représenter les données dans

un variant à valeur complexe. Il définit donc un type enregistrement de même taille que *TVarData* contenant une référence à un objet *TComplexData* :

```
TComplexVarData = packed record
  VType: TVarType;
  Reserved1, Reserved2, Reserved3: Word;
  VComplex: TComplexData;
  Reserved4: LongInt;
end;
```

Les références d'objet sont en réalité des pointeurs (sur deux mots), et ce type est de même taille que l'enregistrement *TVarData*. Comme précédemment, un variant personnalisé complexe (ou son enregistrement *TVarData*), peut être convertit en *TComplexVarData*, et le type variant personnalisé fonctionnera avec l'enregistrement *TVarData* comme il le ferait avec un type *TComplexVarData*.

Création d'une classe pour le type variant personnalisé

Les variants personnalisés fonctionnent en utilisant une classe utilitaire spéciale qui indique comment les variants du type personnalisé effectuent les opérations standard. Vous créez cette classe utilitaire en écrivant un descendant de *TCustomVariantType*. Cela implique de redéfinir les méthodes virtuelles appropriées de *TCustomVariantType*.

Transtypage

Le transtypage est une des fonctionnalités du type variant personnalisé les plus importantes à implémenter. La flexibilité des variants vient en partie de leur transtypage implicite.

Il y a deux méthodes à implémenter pour que le type Variant personnalisé effectue des transtypages : *Cast*, qui convertit un autre type Variant en votre variant personnalisé, et *CastTo*, qui convertit votre variant personnalisé en un autre type Variant.

Quand vous implémentez l'une de ces méthodes, il est relativement facile de faire les conversions logiques à partir des types variant intégrés. Cependant, vous devez envisager la possibilité que le variant vers ou depuis lequel vous transtypez peut être un autre type Variant personnalisé. Pour résoudre cette situation, vous pouvez essayer, dans un premier temps, une conversion en un des types Variant intégrés.

Par exemple, la méthode *Cast* suivante, de la classe *TComplexVariantType*, utilise le type Double comme type intermédiaire :

```
procedure TComplexVariantType.Cast(var Dest: TVarData; const Source: TVarData);
var
  LSource, LTemp: TVarData;
begin
  VarDataInit(LSource);
  try
    VarDataCopyNoInd(LSource, Source);
    if VarDataIsStr(LSource) then
```

```
      TComplexVarData(Dest).VComplex := TComplexData.Create(VarDataToStr(LSource))
    else
    begin
      VarDataInit(LTemp);
      try
        VarDataCastTo(LTemp, LSource, varDouble);
        TComplexVarData(Dest).VComplex := TComplexData.Create(LTemp.VDouble, 0);
      finally
        VarDataClear(LTemp);
      end;
    end;
    Dest.VType := VarType;
  finally
    VarDataClear(LSource);
  end;
end;
```

En plus de l'utilisation d'un Double comme type Variant intermédiaire, il faut remarquer deux ou trois choses dans cette implémentation :

- La dernière étape de cette méthode définit le membre *VType* de l'enregistrement *TVarData* renvoyé. Ce membre donne le code du type Variant. Il est défini par la propriété *VarType* de *TComplexVariantType*, qui est le code du type Variant affecté au variant personnalisé.
- Les données du variant personnalisé (*Dest*) sont transtypées depuis *TVarData* dans le type enregistrement utilisé pour stocker ses données (*TComplexVarData*). Cela facilite le travail sur ces données.
- La méthode fait une copie locale du variant source au lieu de travailler directement avec ses données. Cela évite les effets secondaires qui pourraient affecter les données source.

Lors du transtypage depuis un variant complexe vers un autre type, la méthode *CastTo* utilise également le type intermédiaire Double (pour tout type de destination autre que chaîne) :

```
procedure TComplexVariantType.CastTo(var Dest: TVarData; const Source: TVarData;
  const AVarType: TVarType);
var
  LTemp: TVarData;
begin
  if Source.VType = VarType then
    case AVarType of
      varOleStr:
        VarDataFromOleStr(Dest, TComplexVarData(Source).VComplex.AsString);
      varString:
        VarDataFromStr(Dest, TComplexVarData(Source).VComplex.AsString);
    else
      VarDataInit(LTemp);
      try
        LTemp.VType := varDouble;
        LTemp.VDouble := TComplexVarData(LTemp).VComplex.Real;
        VarDataCastTo(Dest, LTemp, AVarType);
      finally
```

```
        VarDataClear(LTemp);
      end;
    end
  else
    RaiseCastError;
end;
```

Remarquez que la méthode *CastTo* prévoit le cas où les données variant source n'ont pas un code de type qui corresponde à la propriété *VarType*. Ce cas se produit uniquement pour les variants source vides (non affectés).

Implémentation d'opérations binaires

Pour que le type variant personnalisé puisse fonctionner avec les opérateurs binaires standard (+, -, *, /, div, mod, shl, shr, and, or, xor de l'unité System), vous devez redéfinir la méthode *BinaryOp*. *BinaryOp* a trois paramètres : la valeur de l'opérande gauche, la valeur de l'opérande droit et l'opérateur. Implémentez cette méthode pour effectuer l'opération et renvoyer le résultat en utilisant la même variable que celle qui contenait l'opérande gauche.

Par exemple, la méthode *BinaryOp* suivante vient de *TComplexVariantType*, définie dans l'unité VarCmplx :

```
procedure TComplexVariantType.BinaryOp(var Left: TVarData; const Right: TVarData;
  const Operator: TVarOp);
begin
  if Right.VType = VarType then
    case Left.VType of
      varString:
        case Operator of
          opAdd: Variant(Left) := Variant(Left) + TComplexVarData(Right).VComplex.AsString;
        else
          RaiseInvalidOp;
        end;
      else
        if Left.VType = VarType then
          case Operator of
            opAdd:
              TComplexVarData(Left).VComplex.DoAdd(TComplexVarData(Right).VComplex);
            opSubtract:
              TComplexVarData(Left).VComplex.DoSubtract(TComplexVarData(Right).VComplex);
            opMultiply:
              TComplexVarData(Left).VComplex.DoMultiply(TComplexVarData(Right).VComplex);
            opDivide:
              TComplexVarData(Left).VComplex.DoDivide(TComplexVarData(Right).VComplex);
            else
              RaiseInvalidOp;
          end
        else
          RaiseInvalidOp;
    end
  else
    RaiseInvalidOp;
end;
```

Plusieurs remarques importantes sur cette implémentation s'imposent :

Cette méthode ne gère que le cas où le variant du côté droit de l'opérateur est un variant personnalisé représentant un nombre complexe. Si l'opérande gauche est un variant complexe et non l'opérande droit, le variant complexe force l'opérande droit à être d'abord transtypé en variant complexe. Il le fait en redéfinissant la méthode *RightPromotion* pour qu'elle exige toujours le type de la propriété *VarType* :

```
function TComplexVariantType.RightPromotion(const V: TVarData;
  const Operator: TVarOp; out RequiredVarType: TVarType): Boolean;
begin
  { Complex Op TypeX }
  RequiredVarType := VarType;
  Result := True;
end;
```

L'opérateur d'addition est implémenté pour une chaîne et un nombre complexe (par conversion de la valeur complexe en chaîne et concaténation) et les opérateurs d'addition, soustraction, multiplication et division sont implémentés pour deux nombres complexes en utilisant les méthodes de l'objet *TComplexData* qui est stocké dans les données du variant complexe. On y accède en transtypant l'enregistrement *TVarData* en enregistrement *TComplexVarData* et en utilisant son membre *VComplex*.

Essayer tout autre opérateur ou combinaison de types force la méthode à appeler la méthode *RaiseInvalidOp*, qui entraîne une erreur d'exécution. La classe *TCustomVariantType* contient de nombreuses méthodes utilitaires comme *RaiseInvalidOp* qui peuvent être utilisées dans l'implémentation des types variants personnalisés.

BinaryOp ne fonctionne qu'avec un nombre limité de types : les chaînes et d'autres variants complexes. Il est possible, cependant, d'effectuer des opérations entre des nombres complexes et d'autres types numériques. Pour que la méthode *BinaryOp* fonctionne, les opérandes doivent être convertis en variants complexes avant que les valeurs ne soient transmises à cette méthode. Nous avons déjà vu (plus haut) comment utiliser la méthode *RightPromotion* pour forcer l'opérande droit à être un variant complexe quand l'opérande gauche est un complexe. Une méthode similaire, *LeftPromotion*, force le transtypage de l'opérande gauche quand l'opérande droit est un complexe :

```
function TComplexVariantType.LeftPromotion(const V: TVarData;
  const Operator: TVarOp; out RequiredVarType: TVarType): Boolean;
begin
  { TypeX opérateur complexe }
  if (Operator = opAdd) and VarDataIsStr(V) then
    RequiredVarType := varString
  else
    RequiredVarType := VarType;
  Result := True;
end;
```

Cette méthode *LeftPromotion* force l'opérande gauche à être transtypé en un autre variant complexe, sauf si c'est une chaîne et que l'opérateur est l'addition, auquel cas *LeftPromotion* permet à l'opérande de rester une chaîne.

Implémentation d'opérations de comparaison

Il y a deux façons de permettre à un type variant personnalisé de supporter les opérateurs de comparaison (=, <>, <, <=, >, >=). Vous pouvez redéfinir la méthode *Compare* ou la méthode *CompareOp*.

La méthode *Compare* est la plus simple si votre type variant personnalisé supporte la totalité des opérateurs de comparaison. *Compare* prend trois paramètres : l'opérande gauche, l'opérande droit et un paramètre var qui renvoie la relation entre les deux. Par exemple, l'objet *TConvertVariantType* de l'unité VarConv implémente la méthode *Compare* suivante :

```
procedure TConvertVariantType.Compare(const Left, Right: TVarData;
  var Relationship: TVarCompareResult);
const
  CRelationshipToRelationship: array [TValueRelationship] of TVarCompareResult =
    (crLessThan, crEqual, crGreaterThan);
var
  LValue: Double;
  LType: TConvType;
  LRelationship: TValueRelationship;
begin
  // supporte...
  // la comparaison entre convvar et un nombre
  //     Compare la valeur de convvar et le nombre donné
  //   convvar1 cmp convvar2
  //     Compare après conversion de convvar2 dans le type de l'unité convvar1
  // La droite peut aussi être une chaîne.  Si la chaîne a des infos d'unité alors elle
  // est traitée comme un varConvert sinon elle est traitée comme un double
  LRelationship := EqualsValue;
  case Right.VType of
    varString:
      if TryStrToConvUnit(Variant(Right), LValue, LType) then
        if LType = CIllegalConvType then
          LRelationship := CompareValue(TConvertVarData(Left).VValue, LValue)
        else
          LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
                                      TConvertVarData(Left).VConvType, LValue, LType)
        else
          RaiseCastError;
    varDouble:
      LRelationship := CompareValue(TConvertVarData(Left).VValue, TVarData(Right).VDouble);
    else
      if Left.VType = VarType then
        LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
                          TConvertVarData(Left).VConvType, TConvertVarData(Right).VValue,
                          TConvertVarData(Right).VConvType)
      else
        RaiseInvalidOp;
  end;
```

```
    Relationship := CRelationshipToRelationship[LRelationship];
end;
```

Si le type personnalisé ne supporte pas le concept de “supérieur à” ou “inférieur à” et seulement “égal à” ou “différent de”, il est difficile d’implémenter la méthode *Compare*, car *Compare* doit renvoyer *crLessThan*, *crEqual* ou *crGreaterThan*. Quand la seule réponse correcte est “différent de”, il est impossible de savoir s’il faut renvoyer *crLessThan* ou *crGreaterThan*. Donc, pour les types qui ne supportent pas le concept d’ordre, vous pouvez à la place redéfinir la méthode *CompareOp*.

CompareOp a trois paramètres : la valeur de l’opérande gauche, la valeur de l’opérande droit et l’opérateur de comparaison. Implémentez cette méthode pour effectuer l’opération et renvoyer un booléen qui indique si la comparaison est *True*. Vous pouvez alors appeler la méthode *RaiseInvalidOp* quand la comparaison n’a aucun sens.

Par exemple, la méthode *CompareOp* suivante vient de l’objet *TComplexVariantType* de l’unité VarCmplx. Elle ne supporte qu’un test d’égalité ou d’inégalité :

```
function TComplexVariantType.CompareOp(const Left, Right: TVarData;
  const Operator: Integer): Boolean;
begin
  Result := False;
  if (Left.VType = VarType) and (Right.VType = VarType) then
    case Operator of
      opCmpEQ:
        Result := TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);
      opCmpNE:
        Result := not
TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);
    else
      RaiseInvalidOp;
    end
  else
    RaiseInvalidOp;
end;
```

Remarquez que les types d’opérandes qui supportent ces deux implémentations sont très limités. Comme avec les opérations binaires, vous pouvez utiliser les méthodes *RightPromotion* et *LeftPromotion* pour limiter les cas à considérer, en forçant un transtypage avant que *Compare* ou *CompareOp* ne soit appelée.

Implémentation d’opérations unaires

Pour que le type variant personnalisé puisse fonctionner avec les opérateurs unaires standard (-, not), vous devez redéfinir la méthode *UnaryOp*. *UnaryOp* possède deux paramètres : la valeur de l’opérande et l’opérateur. Implémentez cette méthode pour effectuer l’opération et renvoyer le résultat en utilisant la même variable que celle qui contenait l’opérande.

Par exemple, la méthode *UnaryOp* suivante vient de *TComplexVariantType*, définie dans l'unité VarCmplx :

```
procedure TComplexVariantType.UnaryOp(var Right: TVarData; const Operator: TVarOp);
begin
  if Right.VType = VarType then
    case Operator of
      opNegate:
        TComplexVarData(Right).VComplex.DoNegate;
    else
      RaiseInvalidOp;
    end
  else
    RaiseInvalidOp;
end;
```

Remarquez que pour l'opérateur logique **not**, qui n'a pas de sens pour des valeurs complexes, cette méthode appelle *RaiseInvalidOp* pour provoquer une erreur d'exécution.

Copie et effacement des variants personnalisés

En plus du transtypage et de l'implémentation des opérateurs, vous devez indiquer comment copier et effacer les variants de votre type Variant personnalisé.

Pour indiquer comment copier la valeur du variant, implémentez la méthode *Copy*. En général, c'est une opération simple, même si vous ne devez pas oublier de libérer la mémoire de toute classe ou structure utilisée pour contenir la valeur du variant :

```
procedure TComplexVariantType.Copy(var Dest: TVarData; const Source: TVarData;
  const Indirect: Boolean);
begin
  if Indirect and VarDataIsByRef(Source) then
    VarDataCopyNoInd(Dest, Source)
  else
    with TComplexVarData(Dest) do
    begin
      VType := VarType;
      VComplex := TComplexData.Create(TComplexVarData(Source).VComplex);
    end;
end;
```

Remarque Le paramètre *Indirect* de la méthode *Copy* signale que la copie doit prendre en compte le cas où le variant contient une référence indirecte à ses données.

Conseil Si votre type variant personnalisé n'alloue pas de mémoire pour contenir ses données (les données tiennent entièrement dans l'enregistrement *TVarData*), votre implémentation de la méthode Copy peut simplement appeler la méthode *SimplisticCopy*.

Pour indiquer comment effacer la valeur du variant, implémentez la méthode *Clear*. Comme avec la méthode Copy, la seule chose compliquée à faire est de libérer d'éventuelles ressources allouées pour stocker les données du variant.

```
procedure TComplexVariantType.Clear(var V: TVarData);
begin
  V.VType := varEmpty;
  FreeAndNil(TComplexVarData(V).VComplex);
end;
```

Vous aurez également besoin d'implémenter la méthode *IsClear*. De cette façon, vous pourrez détecter d'éventuelles valeurs incorrectes ou des valeurs spéciales qui représentent des données vides :

```
function TComplexVariantType.IsClear(const V: TVarData): Boolean;
begin
  Result := (TComplexVarData(V).VComplex = nil) or
            TComplexVarData(V).VComplex.IsZero;
end;
```

Chargement et enregistrement des valeurs des variants personnalisés

Par défaut, quand on affecte au variant personnalisé la valeur d'une propriété publiée, il est transtypé en chaîne quand la propriété est enregistrée dans un fichier fiche, et reconverti à partir de la chaîne quand la propriété est lue dans ce fichier fiche. Vous pouvez cependant fournir votre propre mécanisme de chargement et d'enregistrement des valeurs des variants personnalisés pour utiliser une représentation plus naturelle. Pour ce faire, le descendant de *TCustomVariantType* doit implémenter l'interface *IVarStreamable* à partir de Classes.pas.

IVarStreamable définit deux méthodes, *StreamIn* et *StreamOut*, pour lire et écrire la valeur d'un variant dans un flux. Par exemple, *TComplexVariantType* de l'unité VarCmplx implémente la méthode *IVarStreamable* comme ceci :

```
procedure TComplexVariantType.StreamIn(var Dest: TVarData; const Stream: TStream);
begin
  with TReader.Create(Stream, 1024) do
  try
    with TComplexVarData(Dest) do
    begin
      VComplex := TComplexData.Create;
      VComplex.Real := ReadFloat;
      VComplex.Imaginary := ReadFloat;
    end;
  finally
    Free;
  end;
end;

procedure TComplexVariantType.StreamOut(const Source: TVarData; const Stream: TStream);
begin
  with TWriter.Create(Stream, 1024) do
  try
    with TComplexVarData(Source).VComplex do
    begin
      WriteFloat(Real);
      WriteFloat(Imaginary);
    end;
```

```
  finally
    Free;
  end;
end;
```

Remarquez comment ces méthodes créent un objet lecteur ou écrivain pour que le paramètre *Stream* gère les détails de lecture et d'écriture des valeurs.

Utilisation du descendant de TCustomVariantType

Dans la section d'initialisation de l'unité qui définit votre descendant de *TCustomVariantType*, créez une instance de votre classe. Lorsque vous instanciez votre objet, il se recense automatiquement avec le système de traitement des Variants afin que le nouveau type Variant soit activé. Par exemple, voici la section d'initialisation de l'unité VarCmplx :

```
initialization
  ComplexVariantType := TComplexVariantType.Create;
```

Dans la section de finalisation de l'unité qui définit votre descendant de *TCustomVariantType*, libérez l'instance de votre classe. Cela permet de dérecenser automatiquement le type variant. Voici la section finalisation de l'unité VarCmplx :

```
finalization
  FreeAndNil(ComplexVariantType);
```

Ecriture d'utilitaires fonctionnant avec un type variant personnalisé

Une fois que vous avez créé un descendant de *TCustomVariantType* pour implémenter votre type variant personnalisé, il est possible d'utiliser ce dernier dans des applications. Mais, sans un petit nombre d'utilitaires, ce n'est pas si simple.

Par exemple, sans fonction utilitaire, le seul moyen de créer une instance de votre type variant personnalisé est d'utiliser la procédure globale *VarCast* sur un variant source d'un autre type. C'est une bonne idée de concevoir une méthode qui crée une instance de votre type variant personnalisé à partir d'une valeur ou d'un ensemble de valeurs approprié. Cette fonction ou ensemble de fonctions remplit la structure que vous avez définie pour stocker les données de votre variant personnalisé. Par exemple, la fonction suivante pourrait être utilisée pour créer un variant à valeur complexe :

```
function VarComplexCreate(const AReal, AImaginary: Double): Variant;
begin
  VarClear(Result);
  TComplexVarData(Result).VType := ComplexVariantType.VarType;
  TComplexVarData(ADest).VComplex := TComplexData.Create(ARead, AImaginary);
end;
```

Cette fonction n'existe pas en réalité dans l'unité VarCmplx, mais c'est une synthèse de méthodes existantes, utilisée pour simplifier l'exemple. Remarquez que le variant renvoyé est transtypé dans l'enregistrement qui a été défini pour correspondre à la structure *TVarData* (*TComplexVarData*), puis rempli.

Un autre utilitaire à créer est celui qui renvoie le code de type de votre nouveau type Variant. Ce code de type n'est pas une constante. Il est généré automatiquement quand vous instanciez votre descendant de *TCustomVariantType*. Il est donc utile pour fournir un moyen de déterminer facilement le code de type de votre variant personnalisé. La fonction suivante de l'unité VarCmplx illustre la façon de l'écrire, en renvoyant simplement la propriété *VarType* du descendant de *TCustomVariantType* :

```
function VarComplex: TVarType;
begin
  Result := ComplexVariantType.VarType;
end;
```

Deux autres utilitaires standard fournis pour la plupart des variants personnalisés vérifient si un variant donné est du type personnalisé et transtypent un variant arbitraire dans le nouveau type personnalisé. Voici l'implémentation de ces utilitaires à partir de l'unité VarCmplx :

```
function VarIsComplex(const AValue: Variant): Boolean;
begin
  Result := (TVarData(AValue).VType and varTypeMask) = VarComplex;
end;

function VarAsComplex(const AValue: Variant): Variant;
begin
  if not VarIsComplex(AValue) then
    VarCast(Result, AValue, VarComplex)
  else
    Result := AValue;
end;
```

Remarquez qu'ils utilisent des fonctionnalités standard communes à tous les variants : le membre *VType* de l'enregistrement *TVarData* et la fonction *VarCast*, qui fonctionne à cause des méthodes implémentées dans le descendant de *TCustomVariantType* pour le transtypage des données.

En plus des utilitaires standard mentionnés plus haut, vous pouvez écrire un nombre quelconque d'utilitaires spécifiques à votre nouveau type variant personnalisé. Par exemple, l'unité VarCmplx définit un grand nombre de fonctions qui implémentent des opérations de calcul sur les variants à valeur complexe.

Support des propriétés et des méthodes dans les variants personnalisés

Certains variants ont des propriétés et des méthodes. Par exemple, quand la valeur d'un variant est une interface, vous pouvez utiliser le variant pour lire ou écrire les valeurs des propriétés de cette interface et appeler ses méthodes. Même si votre type variant personnalisé ne représente pas une interface, vous pouvez lui donner des propriétés et des méthodes qu'une application pourra utiliser.

Utilisation de TInvokeableVariantType

Pour fournir le support des propriétés et méthodes, la classe que vous créez pour activer le nouveau type variant personnalisé doit descendre de *TInvokeableVariantType* et non directement de *TCustomVariantType*.

TInvokeableVariantType définit quatre méthodes :

- *DoFunction*
- *DoProcedure*
- *GetProperty*
- *SetProperty*

que vous pouvez implémenter pour supporter les propriétés et les méthodes dans votre type variant personnalisé.

Par exemple, l'unité VarConv utilise *TInvokeableVariantType* comme classe de base pour *TConvertVariantType* de sorte que les variants personnalisés résultant puissent supporter les propriétés. L'exemple suivant présente l'accès en lecture pour ces propriétés :

```
function TConvertVariantType.GetProperty(var Dest: TVarData;
  const V: TVarData; const Name: String): Boolean;
var
  LType: TConvType;
begin
  // supporte...
  //   'Value'
  //   'Type'
  //   'TypeName'
  //   'Family'
  //   'FamilyName'
  //   'As[Type]'
  Result := True;
  if Name = 'VALUE' then
    Variant(Dest) := TConvertVarData(V).VValue
  else if Name = 'TYPE' then
    Variant(Dest) := TConvertVarData(V).VConvType
  else if Name = 'TYPENAME' then
    Variant(Dest) := ConvTypeToDescription(TConvertVarData(V).VConvType)
  else if Name = 'FAMILY' then
    Variant(Dest) := ConvTypeToFamily(TConvertVarData(V).VConvType)
  else if Name = 'FAMILYNAME' then
    Variant(Dest) :=
ConvFamilyToDescription(ConvTypeToFamily(TConvertVarData(V).VConvType))
  else if System.Copy(Name, 1, 2) = 'AS' then
  begin
    if DescriptionToConvType(ConvTypeToFamily(TConvertVarData(V).VConvType),
                             System.Copy(Name, 3, MaxInt), LType) then
      VarConvertCreateInto(Variant(Dest), Convert(TConvertVarData(V).VValue,
                                  TConvertVarData(V).VConvType, LType), LType)
    else
      Result := False;
  end
  else
    Result := False;
```

```
end;
```

La méthode *GetProperty* vérifie le paramètre *Name* pour déterminer quelle propriété est demandée. Elle récupère ensuite l'information de l'enregistrement *TVarData* du Variant (*V*), et la renvoie sous forme du Variant (*Dest*). Remarquez que cette méthode supporte les propriétés dont les noms sont générés de façon dynamique à l'exécution (As[Type]), en se basant sur la valeur en cours du variant personnalisé.

De même, les méthodes *SetProperty*, *DoFunction* et *DoProcedure* sont suffisamment génériques pour que vous puissiez générer de façon dynamique les noms des méthodes ou répondre aux divers nombres et types de paramètres.

Utilisation de TPublishableVariantType

Si le type variant personnalisé stocke ses données en utilisant l'instance d'un objet, alors il existe un moyen plus simple d'implémenter des propriétés, si ce sont aussi les propriétés de l'objet qui représente les données du variant. Si vous utilisez *TPublishableVariantType* comme classe de base pour votre type variant personnalisé, alors il vous suffit d'implémenter la méthode *GetInstance* et toutes les propriétés publiées de l'objet qui représente les données du variant seront implémentées automatiquement pour les variants personnalisés.

Par exemple, comme on l'a vu dans "Stockage des données d'un type Variant personnalisé" à la page 4-30, *TComplexVariantType* stocke les données d'un variant à valeur complexe en utilisant une instance de *TComplexData*. *TComplexData* possède un certain nombre de propriétés publiées (*Real*, *Imaginary*, *Radius*, *Theta* et *FixedTheta*), qui fournissent des informations sur la valeur complexe. *TComplexVariantType* descend de *TPublishableVariantType* et implémente la méthode *GetInstance* pour renvoyer l'objet *TComplexData* (de TypInfo.pas) qui est stocké dans l'enregistrement *TVarData* du variant à valeur complexe :

```
function TComplexVariantType.GetInstance(const V: TVarData): TObject;
begin
  Result := TComplexVarData(V).VComplex;
end;
```

TPublishableVariantType fait le reste. Il redéfinit les méthodes *GetProperty* et *SetProperty* pour qu'elles utilisent les informations de type à l'exécution (RTTI) de l'objet *TComplexData* pour lire et écrire les valeurs de propriétés.

Remarque Pour que *TPublishableVariantType* fonctionne, l'objet qui contient les données du variant personnalisé doit être compilé avec RTTI. Cela signifie qu'il doit être compilé en utilisant la directive {$M+} ou descendre de *TPersistent*.

Utilisation des chaînes

Delphi dispose de divers types caractère ou chaîne qui sont apparus au fur et à mesure de l'évolution du langage Pascal Objet. Cette section offre un aperçu de ces types, de leur rôle et de leurs utilisations. Pour des détails sur la syntaxe du langage, voir dans l'aide en ligne du langage Pascal objet Types de chaînes.

Types caractère

Delphi a trois types caractère : *Char*, *AnsiChar* et *WideChar*.

Le type caractère *Char* provient du Pascal standard, il a été utilisé dans Turbo Pascal puis en Pascal Objet. Plus tard, le Pascal Objet a ajouté les types *AnsiChar* et *WideChar* comme types caractère spécifiques pour gérer les représentations standard des caractères sous Windows. *AnsiChar* a été introduit pour gérer un jeu de caractères ANSI standard sur 8 bits et *WideChar* pour gérer un jeu de caractères Unicode standard sur 16 bits. Les caractères de type *WideChar* sont également appelés caractères étendus. Les caractères étendus sont codés sur deux octets afin que le jeu de caractères puisse représenter davantage de caractères différents. Quand *AnsiChar* et *WideChar* ont été implémentés, *Char* est devenu le type caractère par défaut représentant l'implémentation dont l'utilisation est conseillée pour un système donné. Si vous utilisez *Char* dans votre application, n'oubliez pas que son implémentation risque de changer dans les futures versions de Delphi.

Remarque Pour la programmation multiplate-forme : Le caractère étendu de Linux, wchar_t, utilise 32 bits par caractère. Le standard Unicode sur 16 bits géré par les caractères étendus du Pascal Objet est un sous-ensemble du standard UCS sur 32 bits supporté par Linux et les bibliothèques GNU. Les données en caractères étendus du Pascal doivent être traduites en 32 bits par caractère avant d'être transmises à une fonction du SE en tant que wchar_t.

Le tableau suivant décrit brièvement ces types caractère :

Tableau 4.2 Types caractère du Pascal Objet

Type	Octets	Contenu	Utilisation
Char	1	Un seul caractère	Type de caractère par défaut
AnsiChar	1	Un seul caractère	Caractère 8 bits
WideChar	2	Un seul caractère Unicode	Standard Unicode 16 bits.

Pour davantage d'informations sur l'utilisation de ces types caractère, voir dans l'aide en ligne du *Guide du langage Pascal Objet* la rubrique Types caractèrePour davantage d'informations sur les caractères Unicode, voir dans l'aide en ligne du *Guide du langage Pascal Objet* la rubrique A propos des jeux de caractères étendus.

Types chaîne

Delphi offre trois catégories de types que vous pouvez utiliser lorsque vous travaillez avec des chaînes :

- les pointeurs de caractère,
- les types chaîne,
- les classes chaîne.

Cette section présente les types chaîne et décrit leur utilisation avec les pointeurs de caractère. Pour des informations sur les classes chaîne, voir dans l'aide en ligne la rubrique TStrings.

Delphi dispose de trois implémentations de chaîne : les chaînes courtes, les chaînes longues et les chaînes étendues. Il existe plusieurs types chaîne qui représentent ces implémentations. De plus, le mot réservé **string** correspond par défaut à l'implémentation de chaîne actuellement recommandée.

Chaînes courtes

String a été le premier type chaîne utilisé en Turbo Pascal. **String** était à l'origine implémenté avec une chaîne courte. Les chaînes courtes allouent entre 1 et 256 octets : le premier octet contenant la longueur de la chaîne, les octets restants contenant les caractères de la chaîne :

```
S: string[0..n]// le type string original
```

Quand les chaînes longues ont été implémentées, **string** a été modifié pour exploiter par défaut une implémentation de chaîne longue et *ShortString* a été introduit comme type permettant la compatibilité ascendante. *ShortString* est un type prédéfini pour une chaîne de longueur maximum :

```
S: string[255]// Le type ShortString
```

La quantité de mémoire allouée pour un *ShortString* est statique, c'est-à-dire qu'elle est déterminée à la compilation. Par contre, l'emplacement de la mémoire d'un *ShortString* peut être allouée dynamiquement (par exemple si vous utilisez un *PShortString* qui est un pointeur sur un *ShortString*). Le nombre d'octets de stockage employés par une variable de type chaîne courte correspond à la longueur maximum du type chaîne courte plus un. Pour le type prédéfini *ShortString*, la taille est donc de 256 octets.

Les chaînes courtes, déclarées en utilisant la syntaxe **string**[0..n] et le type prédéfini *ShortString* existent essentiellement pour proposer une compatibilité ascendante avec les versions précédentes de Delphi et de Borland Pascal.

Une directive de compilation, $H, détermine si le mot réservé **string** correspond à une chaîne longue ou courte. A l'état par défaut, {$H+}, **string** représente une chaîne longue. Vous pouvez le changer en un *ShortString* en utilisant la directive {$H-}. L'état {$H-} est surtout pratique pour utiliser du code écrit pour des versions précédentes du Pascal Objet qui utilisaient par défaut le type chaîne courte. Cependant, les chaînes courtes peuvent être utiles dans des structures de données où vous avez besoin d'un composant de taille fixe ou dans des DLL quand vous ne voulez pas utiliser l'unité *ShareMem* (voir aussi dans l'aide en ligne Gestion de la mémoire). Vous pouvez redéfinir localement la signification des définitions de type chaîne pour générer des chaînes courtes. Vous pouvez aussi changer les déclarations de chaînes courtes en string[255] ou en *ShortString* qui sont dépourvues d'ambiguïtés et indépendantes de la directive $H.

Pour davantage d'informations sur les chaînes et le type *ShortString*, voir dans l'aide en ligne du *Guide du langage Pascal Objet* la rubrique Chaînes courtes.

Chaînes longues

Les chaînes longues sont des chaînes allouées dynamiquement dont la longueur maximum est 2 gigaoctets, mais la limite pratique dépend en général de la quantité de mémoire disponible. Comme les chaînes courtes, les chaînes longues utilisent des caractères Ansi sur 8 bits et un indicateur de longueur. A la différence des chaînes courtes, les chaînes longues n'ont pas un élément zéro contenant la longueur dynamique de la chaîne. Pour connaître la longueur d'une chaîne longue, vous devez utiliser la fonction standard *Length* et pour spécifier sa longueur vous devez utiliser la procédure standard *SetLength*. Les chaînes longues utilisent le comptage de références et, comme les *PChars*, ce sont des chaînes à zéro terminal. Pour davantage d'informations sur l'implémentation des chaînes longues, voir dans l'aide en ligne du *Guide du langage Pascal Objet* la rubrique Chaînes longues.

Les chaînes longues sont désignées par le mot réservé **string** et par l'identificateur prédéfini *AnsiString*. Dans les nouvelles applications, il est conseillé d'utiliser le type chaîne longue. Tous les composants de la VCL sont compilés de cette manière, généralement en utilisant **string**. Si vous écrivez des composants, ils doivent également utiliser des chaînes longues tout comme doit le faire le code recevant des données provenant de propriétés de type chaîne. Si vous voulez écrire du code spécifique qui utilise systématiquement une chaîne longue, vous devez utiliser *AnsiString*. Si vous voulez écrire du code flexible qui vous permet de changer facilement le type quand une nouvelle implémentation de chaîne deviendra la norme, vous devez alors utiliser **string**.

Chaînes étendues

Le type *WideChar* permet de représenter des chaînes de caractères étendus comme des tableaux de *WideChars*. Les chaînes étendues sont des chaînes composées de caractères Unicode sur 16 bits. Comme les chaînes longues, les chaînes étendues sont allouées dynamiquement avec une longueur maximale de 2 gigaoctets, mais dont la limite pratique dépend en général de la quantité de mémoire disponible. Dans Delphi, les chaînes étendues n'implémentent pas le comptage de références. Chaque affectation d'une chaîne étendue dans une variable chaîne étendue crée une copie des données de la chaîne. Dans Kylix, les chaînes étendues implémentent le comptage de références.

La mémoire allouée dynamiquement qui contient la chaîne est libérée quand la chaîne étendue sort de la portée. Dans tous les autres domaines, les chaînes étendues possèdent les mêmes attributs que les chaînes longues. Le type chaîne étendue est désigné par l'identificateur prédéfini *WideString*.

Comme la version 32 bits de OLE (Windows seulement) utilise Unicode pour toutes les chaînes, les chaînes doivent être du type chaîne étendue dans tous les propriétés et paramètres de méthodes OLE automation. En outre, la plupart des fonctions API OLE utilisent des chaînes étendues à zéro terminal.

Pour davantage d'informations, voir dans le *Guide du langage Pascal Objet* la rubrique Chaînes étendues.

Types PChar

Un *PChar* est un pointeur sur une chaîne à zéro terminal de caractères de type *Char*. Chacun des trois types caractère dispose d'un type de pointeur prédéfini :

- Un *PChar* est un pointeur sur une chaîne à zéro terminal de caractères 8 bits.
- Un *PAnsiChar* est un pointeur sur une chaîne à zéro terminal de caractères 8 bits.
- Un *PWideChar* est un pointeur sur une chaîne à zéro terminal de caractères 16 bits.

PChar est, avec les chaînes courtes, l'un des types chaîne qui existaient à l'origine dans le Pascal Objet. Il a été créé tout d'abord comme type compatible avec le langage C et l'API Windows.

Chaînes ouvertes

Le type *OpenString* est obsolète mais vous pouvez le rencontrer dans du code ancien. Il n'existe que pour la compatibilité 16 bits et n'est autorisé que dans les paramètres. *OpenString* était utilisé, avant l'implémentation des chaînes longues, pour permettre le transfert comme paramètre d'une chaîne courte de taille indéterminée. Par exemple, la déclaration suivante :

```
procedure a(v : openstring);
```

permet de transmettre comme paramètre une chaîne de longueur quelconque. En son absence, la longueur de chaîne des paramètres formel et réel doivent correspondre exactement. Vous n'avez pas besoin d'utiliser *OpenString* dans les nouvelles applications que vous écrivez.

Pour d'autres informations sur la directive de compilation {$P+/-} voir "Directives de compilation portant sur les chaînes" à la page 4-54.

Routines de la bibliothèque d'exécution manipulant des chaînes

La bibliothèque d'exécution propose de nombreuses routines de manipulation des chaînes spécialisées pour les différents types chaîne. Il y a des routines pour les chaînes étendues, les chaînes longues et les chaînes à zéro terminal (c'est-à-dire *PChar*). Les routines gérant les types *PChar* utilisent le zéro terminal pour déterminer la longueur des chaînes. Pour davantage d'informations sur les chaînes à zéro terminal, voir Utilisation des chaînes à zéro terminal dans l'aide en ligne du *Guide du langage Pascal Objet*.

La bibliothèque d'exécution propose également des routines de formatage de chaîne. Il n'y a pas de catégorie de routines pour les types *ShortString*. Néanmoins, certaines routines prédéfinies dans le compilateur gèrent le type *ShortString*. C'est, par exemple, le cas des fonctions standard *Low* et *High*.

Comme les chaînes longues et étendues sont les plus fréquemment utilisées, les sections suivantes décrivent les routines les manipulant.

Routines manipulant les caractères étendus

Quand vous manipulez des chaînes dans une application, vous devez vous assurer que votre code peut gérer les chaînes rencontrées sur les diverses cibles locales. Il est parfois nécessaire d'utiliser les caractères étendus et les chaînes étendues. En fait, l'une des manières de gérer les jeux de caractères idéographiques consiste à convertir tous les caractères vers un schéma de codage à base de caractères étendus comme Unicode. La bibliothèque d'exécution contient les fonctions de chaînes de caractères étendus suivantes pour faire les conversions entre chaînes de caractères sur un seul octet standard (ou les chaînes MBCS) et chaînes Unicode :

- StringToWideChar
- WideCharLenToString
- WideCharLenToStrVar
- WideCharToString
- WideCharToStrVar

L'utilisation d'un schéma de codage avec des caractères étendus présente cet avantage que vous pouvez avoir sur les chaînes des présupposés qui ne sont pas valables dans les systèmes MBCS. Il y a en effet une relation directe entre le nombre d'octets de la chaîne et son nombre de caractères. Il n'y a pas le risque, comme avec les jeux de caractères MBCS, de couper un caractère en deux ou de confondre le deuxième octet d'un caractère avec le début d'un autre caractère.

Un inconvénient de l'utilisation des caractères étendus est que Windows 95 ne supporte pas les appels des fonctions de l'API relatives aux caractères étendus. Pour cette raison, les composants de la VCL représentent toutes les valeurs de chaînes sous forme de chaînes sur un seul octet ou MBCS. La conversion entre le système de caractères étendus et le système MBCS chaque fois que vous définissez une propriété de chaîne ou lisez sa valeur nécessiterait d'énormes quantités de code supplémentaire et ralentirait votre application. Mais, vous pouvez choisir la conversion en caractères étendus pour certains algorithmes de traitement des chaînes qui tirent profit de la correspondance 1:1 entre caractères et *WideChar*.

Remarque Habituellement, les composants CLX représentent les valeurs chaîne sous forme de chaînes étendues.

Routines usuelles de manipulation des chaînes longues

Il est possible de regrouper les chaînes manipulant des chaînes longues dans plusieurs catégories fonctionnelles. Dans ces catégories, certaines routines sont utilisées dans le même but mais varient dans l'utilisation de critères particuliers dans leurs calculs. Les tableaux suivants énumèrent ces routines en les regroupant selon les catégories suivantes :

- Comparaison
- Conversion majuscules/minuscules
- Modification
- Sous-chaînes

Quand c'est approprié, les tableaux indiquent également si la routine satisfait les critères suivants :

- Différence majuscules/minuscules : si les paramètres de localisation sont utilisés, ils déterminent la définition des caractères majuscules/minuscules. Si la routine n'utilise pas les paramètres de localisation, les analyses sont fondées sur la valeur scalaire des caractères. Si la routine ne tient pas compte des différences majuscules/minuscules, il y a une fusion logique des caractères majuscules et minuscules déterminée par un modèle prédéfini.
- Utilisation des paramètres de localisation : cela permet de personnaliser votre application pour des localisations spécifiques, en particulier dans le cas des environnements pour les langues asiatiques. Dans la plupart des localisations, les caractères minuscules sont censés être inférieurs aux caractères majuscules correspondants. C'est l'opposé de l'ordre ASCII dans lequel les caractères minuscules sont supérieurs aux caractères majuscules. Les routines qui utilisent la localisation Windows commencent généralement par Ansi (Ansi*XXX*).
- Gestion des jeux de caractères multi-octets (MBCS) : les MBCS sont utilisés pour écrire du code pour les localisations extrême-orientales. Les caractères multi-octets sont représentés par un mélange de codes de caractère sur un à deux octets, le nombre d'octets ne correspond donc pas systématiquement à la longueur de la chaîne. Les routines qui gèrent les MBCS analysent les caractères sur un ou deux octets.

  ByteType et *StrByteType* déterminent si un octet donné est l'octet de tête d'un caractère sur deux octets. Faites attention en manipulant des caractères multi-octets à ne pas tronquer une chaîne en coupant en deux un caractère utilisant deux octets. Ne transmettez pas de caractères comme paramètre d'une fonction ou d'une procédure puisque la taille d'un caractère ne peut être déterminée à l'avance. Il faut, à la place, transmettre un pointeur sur un caractère ou une chaîne. Pour davantage d'informations sur MBCS, voir "Codage de l'application" à la page 12-2 au chapitre 12, "Création d'applications internationales".

Tableau 4.3 Routines de comparaison de chaînes

Routine	Différence MAJ/min	Utilisation des paramètres de localisation	Gestion MBCS
AnsiCompareStr	Oui	Oui	Oui
AnsiCompareText	Non	Oui	Oui
AnsiCompareFileName	Non	Oui	Oui
CompareStr	Oui	Non	Non
CompareText	Non	Non	Non

Tableau 4.4 Routines de conversion majuscules/minuscules

Routine	Utilisation des paramètres de localisation	Gestion MBCS
AnsiLowerCase	Oui	Oui
AnsiLowerCaseFileName	Oui	Oui
AnsiUpperCaseFileName	Oui	Oui
AnsiUpperCase	Oui	Oui
LowerCase	Non	Non
UpperCase	Non	Non

Les routines utilisées pour les noms de fichier sous forme de chaîne : *AnsiCompareFileName, AnsiLowerCaseFileName* et *AnsiUpperCaseFileName* utilisent la localisation Windows. Vous devez toujours utiliser des noms de fichier portables car la localisation (le jeu de caractères) utilisée pour les noms de fichier peut changer selon l'interface utilisateur par défaut.

Tableau 4.5 Routines de modification de chaîne

Routine	Différence MAJ/min	Gestion MBCS
AdjustLineBreaks	ND	Oui
AnsiQuotedStr	ND	Oui
StringReplace	précisé par un indicateur	Oui
Trim	ND	Oui
TrimLeft	ND	Oui
TrimRight	ND	Oui
WrapText	ND	Oui

Tableau 4.6 Routines de sous-chaînes

Routine	Différence MAJ/min	Gestion MBCS
AnsiExtractQuotedStr	ND	Oui
AnsiPos	Oui	Oui
IsDelimiter	Oui	Oui
IsPathDelimiter	Oui	Oui
LastDelimiter	Oui	Oui
QuotedStr	Non	Non

Tableau 4.7 Routines de manipulation de chaînes

Routine	Différence MAJ/min	Gestion MBCS
AnsiContainsText	Non	Oui
AnsiEndsText	Non	Non
AnsiIndexText	Non	Oui
AnsiMatchText	Non	Oui
AnsiResemblesText	Non	Non
AnsiStartsText	Non	Oui
IfThen	ND	Oui
LeftStr	Oui	Non
RightStr	Oui	Non
SoundEx	ND	Non
SoundExInt	ND	Non
DecodeSoundExInt	ND	Non
SoundExWord	ND	Non
DecodeSoundExWord	ND	Non
SoundExSimilar	ND	Non
SoundExCompare	ND	Non

Déclaration et initialisation de chaînes

Quand vous déclarez une chaîne longue :

```
S: string;
```

il n'est pas nécessaire de l'initialiser. Les chaînes longues sont automatiquement initialisées vides. Pour tester si une chaîne longue est vide, vous pouvez utiliser la variable *EmptyStr* :

```
S = EmptyStr;
```

ou la comparer à une chaîne vide :

```
S = '';
```

Une chaîne vide ne contient pas de données utilisables. Donc, essayer d'accéder par indice à une chaîne vide est similaire à l'accès à **nil** et provoque une violation d'accès :

```
var
  S: string;
begin
  S[i];    // cela provoque une violation d'accès
  // instructions
end;
```

De même, si vous transtypez une chaîne vide en un *PChar*, le résultat est un pointeur **nil**. Donc, si vous transmettez un tel *PChar* à une routine qui doit le lire ou l'écrire, la routine doit gérer la valeur **nil** :

```
var
  S: string;   // chaîne vide
begin
  proc(PChar(S));  // assurez-vous que proc peut gérer nil
  // instructions
end;
```

Si ce n'est pas le cas, vous devez initialiser la chaîne :

```
S := 'plus nil';
proc(PChar(S));   // proc n'a plus besoin de gérer nil
```

ou vous devez en spécifier la longueur en utilisant la procédure *SetLength* :

```
SetLength(S, 100);// attribue à la chaîne S une longueur dynamique 100
proc(PChar(S));// proc n'a plus besoin de gérer nil
```

Quand vous utilisez *SetLength*, les caractères existant déjà dans la chaîne sont préservés mais le contenu de l'espace nouvellement alloué est indéterminé. Après un appel à *SetLength*, *S* référence obligatoirement une chaîne unique, c'est-à-dire une chaîne dont le compteur de références a la valeur un. Pour connaître la longueur d'une chaîne, utilisez la fonction *Length*.

N'oubliez pas qu'une chaîne **string** déclarée de la manière suivante :

```
S: string[n];
```

est implicitement une chaîne courte et pas une chaîne longue de longueur *n*. Pour déclarer une chaîne longue ayant spécifiquement la longueur *n*, déclarez une variable de type **string**, puis utilisez la procédure *SetLength* :

```
S: string;
SetLength(S, n);
```

Mélange et conversion de types chaîne

Il est possible de mélanger dans des expressions et des affectations des chaînes courtes, longues et étendues ; le compilateur génère automatiquement le code pour effectuer les conversions de type chaîne nécessaires. Par contre, si vous affectez une valeur chaîne à une variable chaîne courte, n'oubliez pas que la valeur chaîne est tronquée si elle excède la longueur maximum déclarée de la variable de type chaîne courte.

Les chaînes longues sont déjà allouées dynamiquement. N'oubliez pas que si vous utilisez l'un des types de pointeur prédéfinis (comme *PAnsiString*, *PString* ou *PWideString)*, vous introduisez un niveau d'indirection supplémentaire. Vous ne devez le faire qu'en connaissance de cause.

Des fonctions supplémentaires (*CopyQStringListToTstrings, Copy TStringsToQStringList, QStringListToTStringList*) sont fournies pour la conversion des types de chaînes Qt sous-jacents et des types de chaînes CLX. Ces fonctions sont localisés dans Qtypes.pas.

Conversions de chaînes en PChar

La conversion de chaînes longues en *PChar* n'est pas effectuée automatiquement. Certaines différences entre les chaînes et les *PChar* peuvent rendre la conversion problématique :

- Les chaînes longues utilisent le comptage de références, mais pas les *PChar*.
- L'affectation d'une chaîne longue copie les données alors qu'un *PChar* est un pointeur sur la mémoire.
- Les chaînes longues sont à zéro terminales et contiennent également la longueur de la chaîne alors que les *PChars* sont seulement à zéro terminal.

Cette section présente ce qui dans ces différences peut causer des erreurs délicates.

Dépendances de chaîne

Il est parfois nécessaire de convertir des chaînes longues en chaînes à zéro terminal, par exemple si vous utilisez une fonction qui attend un *PChar*. Si vous devez transtyper une chaîne en *PChar*, sachez que vous êtes responsable de la durée de vie du *PChar* résultant. Comme les chaînes longues utilisent le comptage de références, le transtypage d'une chaîne en un *PChar* augmente de un les dépendances de la chaîne sans augmenter également le compteur de références. Quand le compteur de références atteint zéro, la chaîne est détruite même s'il y a encore des dépendances portant dessus. Le transtypage en *PChar* disparaît également, et ce alors même que la routine à laquelle vous l'avez transmis l'utilise peut-être encore. Par exemple :

```
procedure my_func(x: string);
begin
  // faire quelque chose avec x
  some_proc(PChar(x)); // transtype la chaîne en PChar
  // vous devez maintenant garantir que la chaîne existe
  // tant que la procédure some_proc a besoin de l'utiliser
end;
```

Renvoi d'une variable locale PChar

Une erreur courante quand vous utilisez un *PChar* est de stocker dans une structure de données ou de renvoyer comme résultat une variable locale. Une fois votre routine achevée, le *PChar* disparaît car c'est un simple pointeur mémoire et non une copie de chaîne utilisant le comptage de références. Par exemple :

```
function title(n: Integer): PChar;
var
  s: string;
begin
  s := Format('titre - %d', [n]);
  Result := PChar(s); // A NE PAS FAIRE
end;
```

Cet exemple renvoie un pointeur sur une donnée chaîne qui est libérée à l'achèvement de la fonction *title*.

Transfert d'une variable locale comme PChar

Si vous avez une variable locale chaîne qui doit être initialisée en appelant une fonction qui attend un paramètre *PChar*. Une solution consiste à créer une variable locale **array of char** et à la transmettre à la fonction. Il faut ensuite affecter cette variable à la chaîne :

```
// version VCL
// MAXSIZE est une constante définie par ailleurs
var
  i: Integer;
  buf: array[0..MAX_SIZE] of char;
  S: string;
begin
  i := GetModuleFilename(0, @buf, SizeOf(buf));// traite @buf comme un PChar
  S := buf;
  // instructions
end;
```

Ou, pour les programmes multiplates-formes, le code est presque identique :

```
// FillBuffer est une fonction définie par ailleurs
function FillBuffer(Buf:PChar;Count:Integer):Integer
begin
  . . .
end;

// MAX_SIZE est une constante définie par ailleurs
var
  i: Integer;
  buf: array[0..MAX_SIZE] of char;
  S: string;
begin
  i := FillBuffer(0, @buf, SizeOf(buf));// traite @buf comme un PChar
  S := buf;
  // instructions
end;
```

Cette manière de procéder est envisageable si la taille du tampon reste suffisamment petite pour être allouée sur la pile. Elle est également fiable car la conversion est automatique entre un type **array of char** et un type **string**. A la fin de l'exécution de *GetModuleFilename* (ou de *FillBuffer* dans la version multiplate-forme), la longueur (*Length*) de la chaîne indique correctement le nombre d'octets écrits dans *buf*.

Pour éliminer le surcoût de la copie du tampon, il est possible de transtyper la chaîne en un *PChar* (si vous êtes certain que la routine n'a pas besoin que le *PChar* reste en mémoire). Mais dans ce cas, la synchronisation de la longueur de la chaîne n'est pas effectuée automatiquement comme c'est le cas lors de l'affectation de **array of char** vers **string**. Vous devez réinitialiser la longueur (*Length*) de la chaîne afin de refléter la longueur réelle de la chaîne. Si vous

utilisez une fonction qui renvoie le nombre d'octets copiés, une seule ligne de code suffit à le faire :

```
var
  S: string;
begin
  SetLength(S, MAX_SIZE); // avant de transtyper en PChar, vérifiez que la chaîne
                          // n'est pas vide
  SetLength(S, GetModuleFilename( 0, PChar(S), Length(S) ) );
  // instructions
end;
```

Directives de compilation portant sur les chaînes

Les directives de compilation suivantes affectent les types caractère et chaîne.

Tableau 4.8 Directives de compilation portant sur les chaînes

Directive	Description
{$H+/-}	La directive de compilation $H contrôle si le mot réservé **string** représente une chaîne courte ou une chaîne longue. A l'état par défaut, {$H+}, **string** représente une chaîne longue. Vous pouvez le changer en *ShortString* en utilisant la directive {$H-}.
{$P+/-}	La directive $P, proposée pour la compatibilité ascendante avec les versions précédentes, n'a de sens que pour le code compilé à l'état {$H-}. $P contrôle la signification des paramètres variables déclarés en utilisant le mot réservé string en étant à l'état {$H-}. A l'état {$P-}, les paramètres variables déclarés en utilisant le mot réservé string sont des paramètres variables normaux. Par contre, à l'état {$P+}, ce sont des paramètres chaîne ouverte. Indépendamment de l'état de la directive $P, il est toujours possible d'utiliser l'identificateur OpenString pour déclarer des paramètres chaîne ouverte.

{$V+/-}	La directive $V contrôle comment s'effectue la vérification de type pour les paramètres chaîne courte transmis comme paramètre variable. A l'état {$V+}, une vérification de type stricte est effectuée et exige que les paramètres formel et réel soient exactement du même type chaîne.
	A l'état {$V-}, toute chaîne courte est autorisée comme paramètre réel, même si sa longueur maximum déclarée n'est pas la même que celle du paramètre formel. Attention : cela peut entraîner une corruption de la mémoire. Par exemple :
	`var S: string[3];`
	`procedure Test(var T: string);`
	`begin`
	`  T := '1234';`
	`end;`
	`begin`
	`  Test(S);`
	`end.`

Tableau 4.8 Directives de compilation portant sur les chaînes (suite)

Directive	Description
{$X+/-}	La directive de compilation {$X+} permet de gérer les chaînes à zéro terminal en activant des règles particulières qui s'appliquent au type prédéfini *PChar* et aux tableaux de caractères d'indice de base zéro. Ces règles permettent d'utiliser des tableaux de caractères d'indice de base zéro ou des pointeurs de caractère avec les routines *Write*, *Writeln*, *Val*, *Assign* et *Rename* de l'unité *System*.

Chaînes et caractères : sujets apparentés

Les rubriques suivantes du *Guide du langage Pascal Objet* abordent les chaînes et les jeux de caractères. Voir aussi chapitre 12, "Création d'applications internationales"

- A propos des jeux de caractères étendus (Aborde les jeux de caractères internationaux)
- Utilisation de chaînes à zéro terminal (Contient des informations sur les tableaux de caractères)
- Chaînes de caractères
- Pointeurs de caractères
- Opérateurs de chaînes

Utilisation des fichiers

Cette section décrit les manipulations de fichier en distinguant entre la manipulation de fichiers disque et les opérations d'entrées/sorties comme la lecture ou l'écriture dans des fichiers. La première section décrit les routines de la bibliothèque d'exécution et de l'API Windows que vous utiliserez pour des opérations courantes impliquant la manipulation de fichiers sur disque. La section suivante est une présentation des types fichier utilisés pour les entrées/sorties. Enfin, la dernière section présente la méthode conseillée pour manipuler les entrées/sorties de fichier, à savoir l'utilisation de flux fichier.

Au contraire du langage Pascal Objet, le système d'exploitation Linux distingue majuscules et minuscules. Faites attention à la casse des caractères lorsque vous travaillez avec des fichiers dans des applications multiplates-formes.

Remarque Les versions précédentes du langage Pascal Objet effectuaient les opérations sur les fichiers mêmes et non pas sur des paramètres nom de fichier, comme c'est le cas aujourd'hui. Avec ces types fichier, vous deviez trouver le fichier, l'affecter à une variable fichier avant de pouvoir, par exemple, renommer le fichier.

Manipulation de fichiers

Plusieurs opérations courantes portant sur les fichiers sont prédéfinies dans la bibliothèque d'exécution Pascal Objet. Les procédures et fonctions manipulant les fichiers agissent à un niveau élevé. Pour la plupart des routines, il vous suffit de

spécifier le nom du fichier et la routine fait pour vous les appels nécessaires au système d'exploitation. Dans certains cas, vous utiliserez à la place des handles de fichiers. Le Pascal Objet propose des routines pour la plupart des manipulations de fichier. Quand ce n'est pas le cas, d'autres routines sont décrites.

Attention Au contraire du langage Pascal Objet, le système d'exploitation Linux distingue majuscules et minuscules. Faites attention à la casse des caractères lorsque vous travaillez avec des fichiers dans des applications multiplates-formes.

Suppression d'un fichier

La suppression efface un fichier du disque et retire son entrée du répertoire du disque. Il n'y a pas d'opération inverse pour restaurer un fichier supprimé : les applications doivent donc généralement demander à l'utilisateur de confirmer les suppressions de fichiers. Pour supprimer un fichier, transmettez le nom du fichier à la fonction *DeleteFile* :

```
DeleteFile(NomFichier);
```

DeleteFile renvoie *True* si le fichier a été supprimé et *False* sinon (par exemple, si le fichier n'existe pas ou s'il est en lecture seule). *DeleteFile* supprime le fichier du disque nommé *NomFichier*.

Recherche d'un fichier

Trois routines permettent de chercher un fichier : *FindFirst*, *FindNext* et *FindClose*. *FindFirst* recherche la première instance d'un nom de fichier ayant un ensemble spécifié d'attributs dans un répertoire spécifié. *FindNext* renvoie l'entrée suivante correspondant au nom et aux attributs spécifiés dans un appel précédent de *FindFirst*. *FindClose* libère la mémoire allouée par *FindFirst*. Vous devez toujours utiliser *FindClose* pour clore une séquence *FindFirst*/*FindNext*. Si vous voulez simplement savoir si un fichier existe, utilisez la fonction *FileExists* qui renvoie *True* si le fichier existe et *False* sinon.

Les trois routines de recherche de fichier attendent dans leurs paramètres un *TSearchRec*. *TSearchRec* définit les informations du fichier recherché par *FindFirst* ou *FindNext*. *TSearchRec* a la déclaration suivante :

```
type
  TFileName = string;
  TSearchRec = record
    Time: Integer;// Time contient l'indicateur horaire du fichier.
    Size: Integer;// Size contient la taille, en octets, du fichier.
    Attr: Integer;// Attr représente les attributs du fichier.
    Name: TFileName;//Name contient le nom de fichier et l'extension.
    ExcludeAttr: Integer;
    FindHandle: THandle;
    FindData: TWin32FindData;// FindData contient des informations complémentaires
    // comme l'heure de création du fichier, l'heure du dernier accès, les noms de fichier
    // long et court.
  end;
```

Si un fichier est trouvé, les champs du paramètre de type *TSearchRec* sont modifiés pour décrire le fichier trouvé. Vous pouvez comparer *Attr* aux

constantes ou aux valeurs d'attributs suivantes afin de déterminer si un fichier a un attribut donné :

Tableau 4.9 Constantes et valeur d'attribut

Constante	Valeur	Description
faReadOnly	$00000001	Fichiers en lecture seule
faHidden	$00000002	Fichiers cachés
faSysFile	$00000004	Fichiers système
faVolumeID	$00000008	Fichier ID de volume
faDirectory	$00000010	Fichiers répertoire
faArchive	$00000020	Fichier archive
faAnyFile	$0000003F	Tout fichier

Pour tester un attribut, combinez la valeur du champ *Attr* et la constante d'attribut avec l'opérateur **and**. Si le fichier a cet attribut, le résultat est supérieur à 0. Par exemple, si le fichier trouvé est un fichier caché, l'expression suivante a la valeur *True* : (*SearchRec.Attr* and *faHidden* > 0). Il est possible de combiner les attributs en combinant avec l'opérateur OR leurs constantes ou valeurs. Par exemple, pour rechercher les fichiers en lecture seule et les fichiers cachés en plus des fichiers normaux, transmettez (*faReadOnly* or *faHidden*) au paramètre *Attr*.

Exemple : Cet exemple utilise une fiche contenant un libellé, un bouton nommé *Search* et un bouton nommé *Again*. Quand l'utilisateur clique sur le bouton *Search*, le premier fichier du répertoire spécifié est trouvé et son nom et sa taille en octets sont affichés dans l'intitulé du libellé. A chaque fois que l'utilisateur clique sur le bouton *Again*, le nom et la taille du fichier correspondant suivant sont affichés dans le libellé :

```
var
  SearchRec: TSearchRec;

procedure TForm1.SearchClick(Sender: TObject);
begin
  FindFirst('c:\Program Files\delphi6\bin\*.*', faAnyFile, SearchRec);
  Label1.Caption := SearchRec.Name + ' occupe ' + IntToStr(SearchRec.Size) + ' octets';
end;

procedure TForm1.AgainClick(Sender: TObject);
begin
  if (FindNext(SearchRec) = 0)
    Label1.Caption := SearchRec.Name + ' occupe ' + IntToStr(SearchRec.Size) +  ' octets';
  else
    FindClose(SearchRec);
end;
```

Dans les applications multiplates-formes, vous devez remplacer les chemins d'accès codés en dur comme c:\Program Files\delphi6\bin*.* par le chemin d'accès correct du système ou utiliser les variables d'environnement (page

Variables d'environnement, quand vous avez choisi Outils | Options d'environnement) pour les représenter.

Modification d'un nom de fichier

Pour changer le nom d'un fichier, utilisez simplement la fonction *RenameFile* :

```
function RenameFile(const AncienNomFichier, NouveauNomFichier: string): Boolean;
```

qui renomme le fichier nommé *AncienNomFichier*, en *NouveauNomFichier*. Si l'opération réussit, *RenameFile* renvoie *True*. Si le fichier ne peut être renommé, par exemple parce qu'il existe déjà un fichier portant le nom *NouveauNomFichier*, elle renvoie *False*. Par exemple :

```
if not RenameFile('ANCNOM.TXT','NOUVNOM.TXT') then
  ErrorMsg('Erreur en renommant le fichier!');
```

Il n'est pas possible de renommer (déplacer) un fichier entre des lecteurs en utilisant *RenameFile*. Pour ce faire, vous devez commencer par copier le fichier, puis supprimer le fichier original.

Remarque *RenameFile*, dans la VCL, est une enveloppe autour de la fonction *MoveFile* de l'API Windows, et donc *MoveFile* ne fonctionne pas non plus d'un lecteur à l'autre.

Routines date-heure de fichier

Les routines *FileAge*, *FileGetDate* et *FileSetDate* agissent sur les valeurs date-heure du système d'exploitation. *FileAge* renvoie l'indicateur de date et d'heure d'un fichier ou -1 si le fichier est inexistant. *FileSetDate* définit l'indicateur de date et d'heure du fichier spécifié et renvoie zéro en cas de réussite ou un code d'erreur en cas d'échec. *FileGetDate* renvoie l'indicateur de date et d'heure d'un fichier ou -1 si le handle est invalide.

Comme la plupart des routines de manipulation de fichier, *FileAge* utilise un nom de fichier sous forme de chaîne. Par contre, *FileGetDate* et *FileSetDate* attendent un paramètre de type *Handle*. Pour obtenir l'accès à un *Handle* de fichier Windows, au choix

- Appelez la fonction *CreateFile* de l'API Windows. *CreateFile* est une fonction 32 bits uniquement qui crée ou ouvre un fichier et renvoie un *Handle* qui peut être utilisé pour accéder au fichier.
- Instanciez *TFileStream* pour créer ou ouvrir un fichier. Ensuite, utilisez la propriété *Handle* comme vous le feriez avec un *Handle* de fichier Windows. Voir "Utilisation des flux fichier" à la page 4-60, pour plus d'informations.

Copie d'un fichier

La bibliothèque d'exécution ne fournit pas de routine pour copier un fichier. Mais, si vous écrivez des applications uniquement pour Windows, vous pouvez appeler directement la fonction *CopyFile* de l'API Windows pour copier un fichier. Comme la plupart des routines de la bibliothèque d'exécution Delphi, *CopyFile* prend comme paramètre un nom de fichier et non un *Handle*. Faites attention lorsque vous copiez un fichier, car les attributs du fichier existants sont

copiés dans le nouveau fichier, mais pas les attributs de sécurité. *CopyFile* est également utile pour déplacer des fichiers d'un lecteur à un autre, car ni la fonction *RenameFile* de Delphi, ni la fonction *MoveFile* de l'API Windows ne peut renommer ou déplacer des fichiers entre lecteurs. Pour plus d'informations, voir l'aide en ligne de Windows.

Types fichier et E/S de fichier

Vous pouvez utiliser trois types fichier pour les entrées/sorties (E/S) de fichier : les types de fichier Pascal, les handles de fichier et les objets flux fichier. Le tableau suivant les décrit.

Tableau 4.10 Types de fichiers et E/S de fichier

Type de fichier	Description
Types de fichier Pascal	Dans l'unité System. Ces types sont utilisés pour les variables fichier, généralement de la forme "F: Text:" ou "F: File". Il existe trois formes de ces fichiers : typé, texte et sans type. De nombreuses routines de gestion de fichier, comme *AssignPrn* ou *writeln*, les utilisent. Ces types de fichier sont obsolètes et incompatibles avec les handles des fichiers Windows. Si vous avez besoin de travailler avec ces types de fichier, consultez le *Guide du langage Pascal Objet*.
Handles de fichier	Dans l'unité Sysutils. Beaucoup de routines utilisent un handle pour identifier le fichier. Vous obtenez le handle lorsque vous ouvrez ou créez le fichier (par exemple, en utilisant FileOpen ou FileCreate). Lorsque vous avez le handle, vous disposez de routines pour travailler sur le contenu du fichier étant donné son handle (écrire une ligne, lire le texte, etc.). En programmation Windows, les handles de fichier Pascal Objet sont des enveloppes pour le type handle de fichier Windows. Les routines de gestion des handles de fichier de la bibliothèque d'exécution qui utilisent les Handles de fichier Windows sont généralement des enveloppes autour des fonctions de l'API Windows. Par exemple, FileRead appelle la fonction ReadFile de Windows. Comme les fonctions Delphi utilisent la syntaxe du Pascal Objet et fournissent occasionnellement les valeurs par défaut des paramètres, elles peuvent facilement servir d'interface à l'API de Windows. L'utilisation de ces routines est immédiate et si vous connaissez bien les routines de fichier de l'API Windows, vous vous en servirez volontiers pour travailler avec les E/S de fichiers.
Flux fichier	Les flux fichier sont des instances d'objet de la classe *TFileStream* utilisées pour accéder aux informations de fichiers disque. Les flux fichier sont portables et proposent une approche de haut niveau des opérations d'E/S de fichier. *TFileStream* a une propriété *Handle* qui vous donne accès au handle de fichier. La section suivante décrit *TFileStream*.

Utilisation des flux fichier

TFileStream est une classe permettant aux applications de lire et d'écrire dans un fichier résidant sur le disque. Elle est utilisée pour une représentation objet de haut niveau des flux fichier. *TFileStream* offre de nombreuses fonctionnalités : la persistance, l'interaction avec d'autres flux et les E/S de fichier.

- *TFileStream* est un descendant des classes flux. En tant que tel, il hérite de la capacité à stocker de manière persistante les propriétés des composants. C'est un des avantages de l'utilisation des flux fichier. Les classes flux peuvent travailler avec les classes *TFiler*, *TReader* et *TWriter* pour lire des flux d'objets depuis un disque. Quand vous avez un flux fichier, vous pouvez donc utiliser le même code que celui employé par le mécanisme de flux de composant. Pour davantage d'informations sur l'utilisation du système de flux de composant, voir l'aide en ligne sur les classes *TStream*, *TFiler*, *TReader*, *TWriter*, et *TComponent*.
- *TFileStream* peut facilement interagir avec d'autres classes flux. Si, par exemple, vous voulez écrire sur disque un bloc de mémoire dynamique, vous pouvez le faire en utilisant un *TFileStream* et un *TMemoryStream*.
- *TFileStream* propose les méthodes et propriétés de base pour les entrées/sorties de fichier. Les sections suivantes abordent cet aspect des flux fichier.

Création et ouverture de fichiers

Pour créer ou ouvrir un fichier et disposer d'un handle pour le fichier, il suffit d'instancier un *TFileStream*. Cela crée ou ouvre le fichier spécifié et propose des méthodes qui permettent de lire ou d'écrire dans le fichier. Si le fichier ne peut être ouvert, *TFileStream* déclenche une exception.

```
constructor Create(const filename: string; Mode: Word);
```

Le paramètre *Mode* spécifie comment le fichier doit être ouvert à la création du flux fichier. Le paramètre *Mode* est constitué d'un mode d'ouverture et d'un mode de partage reliés par un ou logique. Le mode d'ouverture doit prendre l'une des valeurs suivantes :

Tableau 4.11 Modes d'ouverture

Valeur	Signification
fmCreate	TFileStream crée un fichier portant le nom spécifié. S'il existe déjà un fichier portant ce nom, il est ouvert en mode écriture.
fmOpenRead	Ouvre le fichier en lecture seulement.
fmOpenWrite	Ouvre le fichier en écriture seulement. L'écriture dans le fichier remplace son contenu actuel.
fmOpenReadWrite	Ouvre le fichier pour en modifier le contenu et non pour le remplacer.

Le mode de partage peut prendre l'une des valeurs suivantes et comporte les limites énumérées ci-dessous :

Tableau 4.12 Modes de partage

Valeur	Signification
fmShareCompat	Le partage est compatible avec la manière dont les FCB sont ouverts.
fmShareExclusive	En aucun cas, une autre application ne peut ouvrir le fichier.
fmShareDenyWrite	Les autres applications peuvent ouvrir le fichier en lecture, mais pas en écriture.
fmShareDenyRead	Les autres applications peuvent ouvrir le fichier en écriture, mais pas en lecture.
fmShareDenyNone	Rien n'empêche les autres applications de lire ou d'écrire dans le fichier.

Notez que le mode de partage utilisable dépend du mode d'ouverture utilisé. Le tableau suivant montre les modes de partage associés à chaque mode d'ouverture.

Tableau 4.13 Modes de partage disponible pour chaque mode d'ouverture

Mode d'ouverture	fmShareCompat	fmShareExclusive	fmShareDenyWrite	fmShareDenyRead	fmShareDenyNone
fmOpenRead	Non utilisable	Non utilisable	Disponible	Non utilisable	Disponible
fmOpenWrite	Disponible	Disponible	Non utilisable	Disponible	Disponible
fmOpenReadWrite	Disponible	Disponible	Disponible	Disponible	Disponible

Les constantes d'ouverture et de partage de fichier sont définies dans l'unité *SysUtils*.

Utilisation du handle de fichier

Quand vous instanciez *TFileStream*, vous avez accès au handle de fichier. Le handle de fichier est contenu dans la propriété *Handle*. *Handle* est en lecture seule et indique le mode d'ouverture du fichier. Si vous voulez modifier les attributs du handle de fichier, vous devez créer un nouvel objet *flux fichier*.

Certaines routines de manipulation de fichier attendent comme paramètre un handle de fichier de fenêtre. Quand vous disposez d'un flux fichier, vous pouvez utiliser la propriété *Handle* dans toutes les situations où vous utiliseriez un handle de fichier de fenêtre. Attention, à la différence des handles de flux, les flux fichiers ferment le handle de fichier quand l'objet est détruit.

Lecture et écriture de fichiers

TFileStream dispose de plusieurs méthodes permettant de lire et d'écrire dans les fichiers. Elles sont différenciées selon qu'elles se trouvent dans l'une ou l'autre des situations suivantes :

- Renvoi du nombre d'octets lus ou écrits.
- Nécessité de connaître le nombre d'octets.
- Déclenchement d'une exception en cas d'erreur.

Read est une fonction qui lit jusqu'à *Count* octets dans le fichier associé à un flux fichier en commençant à la position en cours (*Position)* et les place dans *Buffer*. *Read* déplace ensuite la position en cours dans le fichier du nombre d'octets effectivement lus. *Read* a le prototype suivant :

```
function Read(var Buffer; Count: Longint): Longint; override;
```

Read est utile quand le nombre d'octets du fichier est inconnu. *Read* renvoie le nombre d'octets effectivement transférés qui peut être inférieur à *Count* si le marqueur de fin de fichier a été atteint.

Write est une fonction qui écrit *Count* octets de *Buffer* dans le fichier associé au flux fichier en commençant à la position en cours (*Position)*. *Write* a le prototype suivant :

```
function Write(const Buffer; Count: Longint): Longint; override;
```

Après avoir écrit dans le fichier, *Write* avance la position en cours dans le fichier du nombre d'octets écrits et renvoie le nombre d'octets effectivement écrits qui peut être inférieur à *Count* si la fin du tampon a été atteinte.

La paire de procédures correspondantes *ReadBuffer* et *WriteBuffer* ne renvoient pas comme *Read* et *Write* le nombre d'octets lus ou écrits. Ces procédures sont utiles dans les situations où le nombre d'octet est connu et obligatoire, par exemple pour la lecture de structures. *ReadBuffer* et *WriteBuffer* déclenchent une exception en cas d'erreur (*EReadError* et *EWriteError*), alors que les méthodes *Read* et *Write* ne le font pas. Les prototypes de *ReadBuffer* et *WriteBuffer* sont :

```
procedure ReadBuffer(var Buffer; Count: Longint);

procedure WriteBuffer(const Buffer; Count: Longint);
```

Ces méthodes appellent les méthodes *Read* et *Write* pour réaliser la lecture ou l'écriture.

Lecture et écriture de chaînes

Si vous transmettez une chaîne à une fonction de lecture ou d'écriture, vous devez veiller à utiliser la bonne syntaxe. Les paramètres *Buffer* des routines de lecture et d'écriture sont des paramètres, respectivement, **var** et **const**. Ce sont des paramètres sans type, les routines utilisent donc l'adresse des variables.

Le type chaîne longue est le plus couramment utilisé pour la manipulation de chaînes. Cependant, la transmission d'une chaîne longue comme paramètre *Buffer* ne génère pas le résultat adéquat. Les chaînes longues contiennent une taille, un compteur de références et un pointeur sur les caractères de la chaîne. De ce fait, déréférencer une chaîne longue ne donne pas seulement l'élément pointeur. Vous devez tout d'abord transtyper la chaîne en *Pointer* ou *PChar*, puis alors seulement la déréférencer. Par exemple :

```
procedure caststring;
var
  fs: TFileStream;
const
  s: string = 'Hello';
begin
```

```
  fs := TFileStream.Create('temp.txt', fmCreate or fmOpenWrite);
  fs.Write(s, Length(s));// cela donne un résultat faux
  fs.Write(PChar(s)^, Length(s));// cela donne le bon résultat
end;
```

Déplacements dans un fichier

La plupart des mécanismes d'E/S de fichier proposent le moyen de se déplacer à l'intérieur d'un fichier afin de pouvoir lire ou écrire à un emplacement particulier du fichier. *TFileStream* propose pour ce faire la méthode *Seek*. *Seek* a le prototype suivant :

```
function Seek(Offset: Longint; Origin: Word): Longint; override;
```

Le paramètre *Origin* indique la manière d'interpréter le paramètre *Offset*. *Origin* peut prendre l'une des valeurs suivantes :

Valeur	Signification
soFromBeginning	Offset part du début de la ressource. Seek se déplace sur la position Offset. Offset doit être >= 0.
soFromCurrent	Offset part de la position en cours dans la ressource. Seek se déplace sur la position Position + Offset.
soFromEnd	Offset part de la fin de la ressource. Offset doit être <= 0 afin d'indiquer le nombre d'octets avant la fin du fichier.

Seek réinitialise la valeur de *Position* dans le flux en la déplaçant du décalage spécifié. *Seek* renvoie la nouvelle valeur de la propriété *Position*, la nouvelle position en cours dans la ressource.

Position et taille de fichier

TFileStream dispose de propriétés qui contiennent la position en cours dans le fichier et sa taille. Ces propriétés sont utilisées par les méthodes *de lecture et d'écriture et par Seek*.

La propriété *Position* de *TFileStream* est utilisée pour indiquer le décalage en cours dans le flux, exprimé en octets, à partir du début des données du flux. *Position* a la déclaration suivante :

```
property Position: Longint;
```

La propriété *Size* indique la taille en octets du flux. Elle est utilisée comme marqueur de fin de fichier afin de tronquer le fichier. *Size* a la déclaration suivante :

```
property Size: Longint;
```

Size est utilisée de manière interne par les routines qui lisent et écrivent dans le flux.

L'initialisation de la propriété *Size* modifie la taille du fichier. Si la taille du fichier ne peut être modifiée, une exception est déclenchée. Ainsi, tenter de modifier la taille d'un fichier ouvert dans le mode *fmOpenRead* déclenche une exception.

Copie

CopyFrom copie le nombre spécifié d'octets d'un flux (fichier) dans un autre.

```
function CopyFrom(Source: TStream; Count: Longint): Longint;
```

L'utilisation de *CopyFrom* évite à l'utilisateur qui veut copier des données de créer un tampon, d'y placer les données, de les écrire puis de libérer le tampon.

CopyFrom copie *Count* octets de *Source* dans le flux. *CopyFrom* déplace ensuite la position en cours de *Count* octets et renvoie le nombre d'octets copiés. Si *Count* vaut 0, *CopyFrom* initialise la position dans *Source* à 0 avant de lire puis de copier dans le flux la totalité des données de *Source*. Si *Count* est supérieur ou inférieur à 0, *CopyFrom* lit à partir de la position en cours dans *Source*.

Conversion de mesures

L'unité ConvUtils déclare une fonction *Convert* que vous pouvez utiliser pour convertir des mesures entre plusieurs unités. Vous pouvez effectuer des conversions entre unités compatibles, par exemple des pouces et des pieds ou des jours et des semaines. Les unités mesurant les même types de choses sont dites appartenir à la même *famille de conversion*. Les unités que vous convertissez doivent appartenir à la même famille de conversion. Pour plus d'informations sur les conversions, reportez-vous à la section suivante Exécution des conversions et à *Convert* dans l'aide en ligne.

L'unité StdConvs définit plusieurs familles de conversion et les unités de mesure de chacune de ces familles. De plus, vous pouvez créer vos propres familles de conversion et leurs unités associées en utilisant les fonctions *RegisterConversionType* et *RegisterConversionFamily*. Pour savoir comment étendre les conversions et les unités de conversion, reportez-vous à la section Ajout de nouveaux types de mesure et à Convert dans l'aide en ligne.

Exécution des conversions

Vous pouvez utiliser la fonction *Convert* pour exécuter des conversions simples ou complexes. Elle emploie une syntaxe simple et une syntaxe moins simple réservée aux conversions entre types de mesure complexes.

Exécution des conversions simples

Vous pouvez utiliser la fonction *Convert* pour convertir une mesure d'une unité dans une autre. La fonction *Convert* effectue des conversions entre unités mesurant le même type d'objet (distance, surface, temps, température, etc.).

Pour utiliser *Convert*, vous devez spécifier l'unité à partir de laquelle se fait la conversion et celle dans laquelle elle se fait. Vous utilisez le type *TConvType* pour identifier les unités de mesure.

Par exemple, ce qui suit convertit une température des degrés Fahrenheit en degrés Kelvin :

```
TempInKelvin := Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
```

Exécution des conversions complexes

Vous pouvez aussi utiliser la fonction *Convert* pour effectuer des conversions plus complexes entre des rapports de deux types de mesure. C'est le cas lorsque vous effectuez des conversions de miles par heure en mètres par minute pour exprimer une vitesse ou de gallons par minute en litres par heure pour exprimer un débit.

Par exemple, l'appel suivant convertit des miles par gallon en kilomètres par litre :

```
nKPL := Convert(StrToFloat(Edit1.Text), duMiles, vuGallons, duKilometers, vuLiter);
```

Les unités que vous convertissez doivent appartenir à la même famille de conversion (elles doivent mesurer la même chose). Si les unités ne sont pas compatibles, *Convert* déclenche une exception *EConversionError*. Vous pouvez vérifier si deux valeurs *TConvType* appartiennent à la même famille de conversion en appelant *CompatibleConversionTypes*.

L'unité StdConvs définit plusieurs familles de valeurs *TConvType*. Voir Variables des familles de conversion dans l'aide en ligne pour avoir la liste des familles d'unités de mesure prédéfinies et les unités de mesure de chaque famille.

Ajout de nouveaux types de mesure

Si vous voulez effectuer des conversions entre unités de mesure non encore définies dans l'unité StdConvs, il vous faut créer une nouvelle famille de conversion pour représenter ces unités de mesure (valeurs *TConvType*). Quand deux valeurs *TConvType* sont recensées dans la même famille de conversion, la fonction *Convert* peut effectuer la conversion entre ces mesures en utilisant les unités que représentent ces valeurs *TConvType*.

Vous devez d'abord obtenir les valeurs de *TConvFamily* recensant une famille de conversion à l'aide de la fonction *RegisterConversionFamily*. Une fois que vous avez obtenu une valeur de *TConvFamily* (en recensant une nouvelle famille de conversion ou en utilisant une des variables globales de l'unité StdConvs), vous pouvez utiliser la fonction *RegisterConversionType* pour ajouter les nouvelles unités à la famille de conversion. Les exemples suivants montrent cela.

Pour avoir davantage d'exemples, reportez-vous au code source de l'unité de conversions standard (stdconvs.pas). (Le source n'est pas inclus dans toutes les versions de Delphi.)

Création d'une famille de conversion simple et ajout d'unités

Vous pouvez, par exemple, créer une nouvelle famille de conversion et ajouter de nouveaux types de mesure lorsque vous effectuez des conversions entre de longues périodes de temps (mois ou siècles), qui risquent d'être inexactes.

Pour mieux comprendre, la famille *cbTime* utilise le jour comme unité de base. L'unité de base est celle qui est utilisée pour effectuer toutes les conversions à l'intérieur de cette famille. Donc, toutes les conversions doivent être faites en jours. Une perte de précision peut se produire lors de conversions utilisant

comme unités le mois ou plus (année, décade, siècle, millénaire), car la conversion entre jours et mois, jours et années, etc. n'est pas exacte. Les mois ont différentes longueurs, les années ont des facteurs de correction pour les années bissextiles, les secondes supplémentaires, etc.

Si vous utilisez uniquement des unités de mesure égales ou supérieures au mois, vous pouvez créer une famille de conversion plus précise en prenant l'année comme unité de base. Cet exemple crée une nouvelle famille de conversion nommée *cbLongTime*.

Déclaration des variables

D'abord, vous devez déclarer des variables pour les identificateurs. Les identificateurs sont utilisés dans la nouvelle famille de conversion LongTime et les unités de mesure qui sont ses membres :

```
var
   cbLongTime: TConvFamily;
   ltMonths: TConvType;
   ltYears: TConvType;
   ltDecades: TConvType;
   ltCenturies: TConvType;
   ltMillennia: TConvType;
```

Recensement de la famille de conversion

Ensuite, recensez la famille de conversion :

```
cbLongTime := RegisterConversionFamily ('Longues durées');
```

Bien que la procédure *UnregisterConversionFamily* soit fournie, il n'est pas nécessaire de dé-recenser les familles de conversion sauf si l'unité qui les définit est supprimée à l'exécution. Elles sont automatiquement nettoyées quand l'application s'arrête.

Recensement des unités de mesure

Ensuite, vous devez recenser les unités de mesure dans la famille de conversion que vous venez de créer. Utilisez la fonction *RegisterConversionType*, qui recense les unités de mesure dans une famille spécifiée. Vous devez définir l'unité de base, ici l'année, et les autres unités en utilisant un facteur indiquant leur rapport avec l'unité de base. Le facteur de *ltMonths* est ainsi 1/12 puisque l'unité de base de la famille LongTime est l'année. Vous fournissez également la description des unités dans lesquelles vous effectuez la conversion.

Le code de recensement des unités de mesure est indiqué ici :

```
ltMonths:=RegisterConversionType(cbLongTime,'Months',1/12);
ltYears:=RegisterConversionType(cbLongTime,'Years',1);
ltDecades:=RegisterConversionType(cbLongTime,'Decades',10);
ltCenturies:=RegisterConversionType(cbLongTime,'Centuries',100);
ltMillennia:=RegisterConversionType(cbLongTime,'Millennia',1000);
```

Utilisation des nouvelles unités

Vous pouvez maintenant utiliser les unités que vous venez de recenser afin d'effectuer des conversions. La fonction *Convert* globale peut effectuer la conversion entre tous les types recensés dans la famille de conversion *cbLongTime*.

Ainsi, au lieu d'utiliser l'appel *Convert* suivant,

```
Convert(StrToFloat(Edit1.Text),tuMonths,tuMillennia);
```

vous pouvez utiliser celui-ci pour obtenir une meilleure précision :

```
Convert(StrToFloat(Edit1.Text),ltMonths,ltMillennia);
```

Utilisation d'une fonction de conversion

Dans les cas où la conversion est plus complexe, vous pouvez utiliser une syntaxe différente et spécifier une fonction qui effectue la conversion au lieu d'utiliser un facteur de conversion. Par exemple, vous ne pouvez pas convertir des températures à l'aide d'un facteur de conversion, puisque les échelles de température ont des origines différentes.

Cet exemple, pris dans l'unité StdConvs, montre comment recenser un type de conversion en fournissant des fonctions de conversion depuis et vers les unités de base.

Déclaration des variables

D'abord, vous devez déclarer des variables pour les identificateurs. Les identificateurs sont utilisés dans la famille de conversion *cbTemperature* et les unités de mesure qui sont ses membres :

```
var
   cbTemperature: TConvFamily;
   tuCelsius: TConvType;
   tuKelvin: TConvType;
   tuFahrenheit: TConvType;
```

Remarque Les unités de mesure présentées ici forment un sous-ensemble des unités de température réellement recensées dans l'unité *StdConvs*.

Recensement de famille de conversion

Ensuite, recensez la famille de conversion :

```
cbTemperature := RegisterConversionFamily ('Temperature');
```

Recensement de l'unité de base

Ensuite, définissez et recensez l'unité de base de la famille de conversion, qui est dans l'exemple le degré Celsius. Remarquez que dans le cas de l'unité de base, nous pouvons utiliser un simple facteur de conversion, car il n'y a pas de conversion à faire :

```
tuCelsius:=RegisterConversionType(cbTemperature,'Celsius',1);
```

Ecriture des méthodes de conversion vers et depuis l'unité de base

Vous devez écrire le code qui effectue la conversion pour chaque échelle de température vers et depuis les degrés Celsius, car elle ne peut se baser sur un simple facteur. Ces fonctions sont prises dans l'unité StdConvs :

```
function FahrenheitToCelsius(const AValue: Double): Double;
begin
  Result := ((AValue - 32) * 5) / 9;
end;

function CelsiusToFahrenheit(const AValue: Double): Double;
begin
  Result := ((AValue * 9) / 5) + 32;
end;

function KelvinToCelsius(const AValue: Double): Double;
begin
  Result := AValue - 273.15;
end;

function CelsiusToKelvin(const AValue: Double): Double;
begin
  Result := AValue + 273.15;
end;
```

Recensement des autres unités

Maintenant que vous avez les fonctions de conversion, vous pouvez recenser les autres unités de mesure dans la famille de conversion. Vous incluez également la description des unités.

Le code de recensement des autres unités de la famille est indiqué ici :

```
tuKelvin := RegisterConversionType(cbTemperature, 'Kelvin', KelvinToCelsius,
CelsiusToKelvin);
  tuFahrenheit := RegisterConversionType(cbTemperature, 'Fahrenheit', FahrenheitToCelsius,
CelsiusToFahrenheit);
```

Utilisation des nouvelles unités

Vous pouvez maintenant utiliser les unités que vous venez de recenser pour effectuer des conversions dans vos applications. La fonction *Convert* globale peut effectuer la conversion entre tous les types recensés dans la famille de conversion *cbTemperature*. Par exemple, le code suivant permet de convertir en degrés Kelvin une valeur exprimée en degrés Fahrenheit.

```
Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
```

Utilisation d'une classe pour gérer les conversions

Vous pouvez toujours utiliser des fonctions de conversion pour recenser une unité de conversion. Mais, il existe des cas où il est nécessaire de créer un nombre inutilement grand de fonctions qui font toutes essentiellement la même chose.

Si vous devez écrire un ensemble de fonctions de conversion qui diffèrent uniquement par la valeur d'un paramètre ou d'une variable, vous pouvez créer

une classe pour gérer ces conversions. Par exemple, il existe un ensemble de techniques standard pour les conversions entre les différentes monnaies européennes depuis l'introduction de l'euro. Bien que les facteurs de conversion restent constants (contrairement au facteur de conversion entre les dollars et les euros), vous ne pouvez pas utiliser un simple facteur de conversion pour convertir correctement le monnaies européennes pour deux raisons :

- La conversion doit arrondir à un nombre de décimales spécifique à la monnaie.
- L'approche facteur de conversion utilise un facteur inverse à celui spécifié par les conversions standard en euro.

Mais, tout peut être géré par les fonctions de conversion comme ceci :

```
function FromEuro(const AValue: Double, Factor, FRound): Double;
begin
  Result := RoundTo(AValue * Factor, FRound);
end;

function ToEuro(const AValue: Double, Factor): Double;
begin
  Result := AValue / Factor;
end;
```

Le problème est que cette approche nécessite des paramètres supplémentaires pour la fonction de conversion, ce qui signifie que vous ne pouvez pas recenser simplement la même fonction pour chaque monnaie européenne. Afin d'éviter d'écrire deux nouvelles fonctions de conversion pour chacune des monnaies, vous pouvez utiliser le même couple de fonctions en les faisant membres d'une classe.

Création de la classe de conversion

La classe doit être un descendant de *TConvTypeInfo*. *TConvTypeFactor* définit deux méthodes, *ToCommon* et *FromCommon*, pour convertir vers et depuis l'unité de base d'une famille de conversion (dans ce cas, vers et depuis des euros). Comme les fonctions que vous utilisez directement pour le recensement d'une unité de conversion, ces méthodes n'ont pas de paramètre supplémentaire, et vous devez fournir le nombre de décimales de l'arrondi et le facteur de conversion en tant que membres de votre classe de conversion. C'est illustré dans l'exemple EuroConv du répertoire demos\ConvertIt (voir euroconv.pas) :

```
type
  TConvTypeEuroFactor = class(TConvTypeFactor)
  private
    FRound: TRoundToRange;
  public
    constructor Create(const AConvFamily: TConvFamily;
      const ADescription: string; const AFactor: Double;
      const ARound: TRoundToRange);
    function ToCommon(const AValue: Double): Double; override;
    function FromCommon(const AValue: Double): Double; override;
  end;
end;
```

Le constructeur attribue des valeurs à ces membres privés :

```
constructor TConvTypeEuroFactor.Create(const AConvFamily: TConvFamily;
  const ADescription: string; const AFactor: Double;
  const ARound: TRoundToRange);
begin
  inherited Create(AConvFamily, ADescription, AFactor);
  FRound := ARound;
end;
```

Les deux fonctions de conversion utilisent tout simplement ces membres privés :

```
function TConvTypeEuroFactor.FromCommon(const AValue: Double): Double;
begin
  Result := SimpleRoundTo(AValue * Factor, FRound);
end;

function TConvTypeEuroFactor.ToCommon(const AValue: Double): Double;
begin
  Result := AValue / Factor;
end;
```

Déclaration des variables

Maintenant que vous avez une classe de conversion, commencez comme avec toute autre famille de conversion, en déclarant les identificateurs :

```
var
  euEUR: TConvType; { Euros }
  euBEF: TConvType; { Francs belges }
  euDEM: TConvType; { Marks allemands}
  euGRD: TConvType; { Drachmes }
  euESP: TConvType; { Pesetas }
  euFFR: TConvType; { Francs français }
  euIEP: TConvType; { Livres irlandaises }
  euITL: TConvType; { Lires }
  euLUF: TConvType; { Francs luxembourgeois }
  euNLG: TConvType; { Florins }
  euATS: TConvType; { Schillings autrichiens }
  euPTE: TConvType; { Escudos }
  euFIM: TConvType; { Marks finlandais }
  euUSD: TConvType; { Dollars US }
  euGBP: TConvType; { Livres irlandaises }
  euJPY: TConvType; { Yens }
```

Recensez la famille de conversion et les autres unités

Vous êtes maintenant prêt à recenser la famille de conversion et les unités monétaires européennes, en utilisant votre nouvelle classe de conversion :

```
cbEuro := RegisterConversionFamily ('Monnaie européenne');
...
// Types de conversion des diverses monnaies
euEUR := RegisterEuroConversionType(cbEuro, SEURDescription, EURToEUR, EURSubUnit);
euBEF := RegisterEuroConversionType(cbEuro, SBEFDescription, BEFToEUR, BEFSubUnit);
euDEM := RegisterEuroConversionType(cbEuro, SDEMDescription, DEMToEUR, DEMSubUnit);
euGRD := RegisterEuroConversionType(cbEuro, SGRDDescription, GRDToEUR, GRDSubUnit);
```

```
euESP := RegisterEuroConversionType(cbEuro, SESPDescription, ESPToEUR, ESPSubUnit);
euFFR := RegisterEuroConversionType(cbEuro, SFFRDescription, FFRToEUR, FFRSubUnit);
euIEP := RegisterEuroConversionType(cbEuro, SIEPDescription, IEPToEUR, IEPSubUnit);
euITL := RegisterEuroConversionType(cbEuro, SITLDescription, ITLToEUR, ITLSubUnit);
euLUF := RegisterEuroConversionType(cbEuro, SLUFDescription, LUFToEUR, LUFSubUnit);
euNLG := RegisterEuroConversionType(cbEuro, SNLGDescription, NLGToEUR, NLGSubUnit);
euATS := RegisterEuroConversionType(cbEuro, SATSDescription, ATSToEUR, ATSSubUnit);
euPTE := RegisterEuroConversionType(cbEuro, SPTEDescription, PTEToEUR, PTESubUnit);
euFIM := RegisterEuroConversionType(cbEuro, SFIMDescription, FIMToEUR, FIMSubUnit);
euUSD := RegisterEuroConversionType(cbEuro, SUSDDescription,
                                    ConvertUSDToEUR, ConvertEURToUSD);
euGBP := RegisterEuroConversionType(cbEuro, SGBPDescription,
                                    ConvertGBPToEUR, ConvertEURToGBP);
euJPY := RegisterEuroConversionType(cbEuro, SJPYDescription,
                                    ConvertJPYToEUR, ConvertEURToJPY);
```

Remarquez que *RegisterEuroConversionType* est une fonction enveloppe qui simplifie le recensement des types monétaires. Voir l'exemple de code pour plus de détails.

Utilisation des nouvelles unités

Vous pouvez maintenant utiliser les unités que vous venez de recenser pour effectuer des conversions dans vos applications. La fonction *Convert* globale peut convertir toutes les monnaies européennes que vous avez recensées dans la famille cbEuro. Par exemple, le code suivant convertit en marks allemands une valeur exprimée en lires :

```
Edit2.Text = FloatToStr(Convert(StrToFloat(Edit1.Text), euITL, euDEM));
```

Définition des types de données

Pascal Objet dispose de nombreux types de données prédéfinis. Vous pouvez utiliser ces types prédéfinis pour créer de nouveaux types qui correspondent aux besoins spécifiques de votre application. Pour une présentation de ces types, voir le *Guide du langage Pascal Objet*.

Chapitre

5

Création d'applications, de composants et de bibliothèques

Ce chapitre donne un aperçu de la manière d'utiliser Delphi pour créer des applications, des bibliothèques et des composants.

Création d'applications

L'utilisation principale de Delphi est la conception et la génération des types d'applications suivants :

- Les applications d'interface utilisateur graphique
- Les applications console
- Les applications service (pour les applications Windows seulement)
- Paquets et DLL

Les applications d'interface utilisateur graphique (GUI) ont en général une interface qui facilite leur utilisation. Les applications console s'exécutent dans une fenêtre console. Les applications service s'exécutent en tant que services Windows. Ces applications sont compilées en tant qu'exécutables, avec du code de démarrage.

Vous pouvez créer d'autres types de projets, comme les paquets et les DLL, bibliothèques de liaison dynamique. Ces applications produisent du code exécutable sans code de démarrage. Reportez-vous à "Création de paquets et de DLL" à la page 10.

Applications d'interface utilisateur graphique

Une application d'interface utilisateur graphique, GUI, est une application conçue en utilisant des fonctionnalités graphiques, fenêtres, menus, boîtes de dialogue, et

des fonctionnalités qui rendent cette application facile à utiliser. Quand vous compilez une application GUI, un fichier exécutable contenant du code de démarrage est créé. Généralement, l'exécutable fournit les fonctions de base de votre programme et les programmes simples sont souvent composés uniquement d'un fichier exécutable. Vous pouvez aussi étendre une application en appelant des DLL, des paquets ou d'autres bibliothèques complétant un exécutable.

Delphi offre deux modèles d'interface utilisateur d'application :

- L'interface de document unique (abrégé en anglais par SDI)
- L'interface de document multiple (abrégé en anglais par MDI)

Outre le modèle d'implémentation de votre application, le comportement de votre projet à la conception, comme celui de l'application à l'exécution, peut être manipulé par des options de projet de l'EDI.

Modèles d'interfaces utilisateur

Toute fiche peut être implémentée comme une fiche d'interface de document multiple (MDI) ou comme une fiche d'interface de document unique (SDI). Dans une application MDI, plusieurs documents ou fenêtres enfant peuvent être ouverts dans une seule fenêtre parent. Cela est courant dans les applications comme les tableurs ou les traitements de texte. Par contre, une application SDI ne contient normalement qu'une seule vue de document. Pour faire de votre fiche une application SDI, affectez la valeur *fsNormal* à la propriété *FormStyle* de votre objet *Form*.

Pour davantage d'informations sur le développement de l'interface utilisateur d'une application, voir chapitre 6, "Conception de l'interface utilisateur des applications".

Applications SDI

Pour créer une nouvelle application SDI :

1. Sélectionnez Fichier | Nouveau | Autre pour afficher la boîte de dialogue Nouveaux éléments.
2. Cliquez sur l'onglet Projets et sélectionnez Application SDI.
3. Choisissez OK.

Par défaut, la propriété *FormStyle* de l'objet *Form* a la valeur *fsNormal*, Delphi suppose que toute nouvelle application est une application SDI.

Applications MDI

Pour créer une nouvelle application MDI :

1. Sélectionnez Fichier | Nouveau | Autre pour afficher la boîte de dialogue Nouveaux éléments.
2. Cliquez sur l'onglet Projets et sélectionnez Application MDI.
3. Choisissez OK.

Les applications MDI nécessitent plus de réflexion et sont plus complexes à concevoir que les applications SDI. Les applications MDI contiennent des fenêtres enfant qui se trouvent dans la fenêtre client ; la fiche principale contient des fiches enfant. Affectez la propriété *FormStyle* de l'objet *TForm* pour spécifier si la fiche est un enfant (*fsMDIForm*) ou si c'est la fiche principale (*fsMDIChild*). Pour éviter d'avoir à redéfinir à plusieurs reprises les propriétés des fenêtres enfant, vous avez intérêt à définir une classe de base pour les fiches enfant et à dériver chaque fiche enfant de cette classe.

Définition des options de l'EDI, du projet et de la compilation

Choisissez Projet | Options pour spécifier les diverses options de votre projet. Pour plus d'informations, voir l'aide en ligne.

Définition des options de projet par défaut

Pour modifier les options de projet par défaut qui s'appliquent à tout nouveau projet, définissez les options de la boîte de dialogue Options de projet, puis cochez la case Défaut en bas à droite de la fenêtre. Tous les nouveaux projets utiliseront ensuite les options en cours comme options par défaut.

Modèles de programmation

Les modèles de programmation sont des structures communément appelées "squelettes" que vous pouvez ajouter au code source puis remplir. Certains modèles de code standard, comme les déclarations de tableaux, de classes ou de fonction, ainsi que de nombreuses instructions, sont livrés avec Delphi.

Vous pouvez aussi écrire vos propres modèles de code pour les structures que vous utilisez souvent. Par exemple, si vous voulez utiliser une boucle **for** dans votre code, insérez le modèle suivant :

```
for := to do
begin

end;
```

Pour insérer un modèle de code dans l'éditeur de code, appuyez sur *Ctrl-j* et sélectionnez le modèle que vous voulez utiliser. Vous pouvez ajouter vos propres modèles à cette collection. Pour ajouter un modèle :

1. Sélectionnez Outils | Options de l'éditeur.
2. Choisissez l'onglet Audit de code.
3. Dans la section Modèles, choisissez Ajouter.
4. Saisissez le nom du modèle après Raccourci et entrez une description brève du nouveau modèle.
5. Ajoutez le modèle de code dans la boîte de saisie Code.
6. Choisissez OK.

Applications console

Les applications console sont des programmes 32 bits exécutés sans interface graphique, généralement dans une fenêtre console. Habituellement, ces applications ne nécessitent pas une saisie utilisateur importante et accomplissent un jeu limité de fonctions.

Pour créer une nouvelle application console :

1 Choisissez Fichier | Nouveau | Autre puis sélectionnez Application console dans la boîte de dialogue Nouveaux éléments.

Delphi crée alors un fichier projet pour le type de fichier source spécifié et affiche l'éditeur de code.

Remarque A la création d'une nouvelle application console, l'EDI ne crée pas une nouvelle fiche. Seul l'éditeur de code est affiché.

Applications service

Les applications service reçoivent les requêtes des applications client, traitent ces requêtes et renvoient les informations aux applications client. Habituellement, elles s'exécutent en arrière-plan, sans nécessiter de saisie utilisateur importante. Un serveur Web, FTP ou de messagerie électronique est un exemple d'application service.

Pour créer une application qui implémente un service Win32, choisissez Fichier | Nouveau puis sélectionnez Application service dans la boîte de dialogue Nouveaux éléments. Cela ajoute à votre projet une variable globale appelée *Application* de type *TServiceApplication*.

Une fois une application service créée, une fenêtre apparaît dans le concepteur qui correspond à un service (*TService*). Implémentez le service en initialisant ses propriétés et ses gestionnaires d'événements dans l'inspecteur d'objets. Vous pouvez ajouter des services supplémentaires en choisissant Service dans la boîte de dialogue Nouveaux éléments. N'ajoutez pas de services à une application qui n'est pas une application service. En effet, même si un objet *TService* est ajouté, l'application ne génère pas les événements nécessaires, ni ne fait les appels Windows appropriés au service.

Une fois que votre application service est construite, vous pouvez installer ses services avec le SCM (Service Control Manager). Les autres applications peuvent alors lancer vos services en envoyant des requêtes au SCM.

Pour installer les services de votre application, exécutez-la à l'aide de l'option /INSTALL. L'application installe ses services puis quitte, en affichant un message de confirmation si les services sont correctement installés. Vous pouvez supprimer l'affichage du message de confirmation en exécutant l'application service à l'aide de l'option /SILENT.

Pour désinstaller les services de votre application, exécutez-la depuis la ligne de commande à l'aide de l'option /UNINSTALL. (Vous pouvez aussi utiliser

l'option /SILENT pour supprimer le message de confirmation lors de la désinstallation).

Exemple Le service suivant contient un *TServerSocket* dont le port est initialisé à 80. C'est le port par défaut des navigateurs Web pour envoyer des requêtes à des serveurs Web et celui utilisé par les serveurs Web pour répondre aux navigateurs Web. Cet exemple spécifique produit, dans le répertoire C:\Temp, un document texte appelé WebLog*xxx*.log (où *xxx* correspond au ThreadID). Il ne doit y avoir qu'un seul serveur surveillant un port donné, donc si vous utilisez déjà un serveur Web, vous devez vous assurer qu'il n'est pas à l'écoute (le service doit être arrêté).

Pour voir les résultats : ouvrez un navigateur Web sur la machine locale, et pour l'adresse, entrez "localhost" (sans les guillemets). Eventuellement, le navigateur va faire une erreur de dépassement de délai mais vous devez obtenir un fichier appelé weblog*xxx*.log dans le répertoire C:\temp.

1. Pour créer l'exemple, choisissez Fichier|Nouveau et sélectionnez Application Service dans la boîte de dialogue Nouveaux éléments. Une fenêtre nommée Service1 apparaît. Ajoutez un composant ServerSocket de la page Internet de la palette de composants à la fenêtre service (Service1).
2. Ajoutez ensuite une donnée membre privée de type *TMemoryStream* à la classe TService1. La section interface de l'unité doit ressembler à :

```
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, SvcMgr, Dialogs,
  ScktComp;

type
  TService1 = class(TService)
    ServerSocket1: TServerSocket;
    procedure ServerSocket1ClientRead(Sender: TObject;
      Socket: TCustomWinSocket);
    procedure Service1Execute(Sender: TService);
  private
    { Déclarations privées }
    Stream: TMemoryStream; // Ajoutez cette ligne
  public
    function GetServiceController: PServiceController; override;
    { Déclarations publiques }
  end;

var
  Service1: TService1;
```

3. Sélectionnez ensuite ServerSocket1, le composant ajouté à l'étape 1. Dans l'inspecteur d'objets, double-cliquez sur l'événement *OnClientRead* et ajoutez le gestionnaire d'événement suivant :

```
procedure TService1.ServerSocket1ClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
var
  Buffer: PChar;
```

```
begin
  Buffer := nil;

while Socket.ReceiveLength > 0 do begin
    Buffer := AllocMem(Socket.ReceiveLength);
    try
      Socket.ReceiveBuf(Buffer^, Socket.ReceiveLength);
      Stream.Write(Buffer^, StrLen(Buffer));
    finally
      FreeMem(Buffer);
    end;

  Stream.Seek(0, soFromBeginning);
  Stream.SaveToFile('c:\Temp\Weblog' + IntToStr(ServiceThread.ThreadID) + '.log');
  end;
end;
```

4 Sélectionnez enfin Service1 en cliquant sur la zone client de la fenêtre (mais pas sur le composant ServiceSocket). Dans l'inspecteur d'objets, double-cliquez sur l'événement *OnExecute* et ajoutez le gestionnaire d'événement suivant :

```
procedure TService1.Service1Execute(Sender: TService);
begin
  Stream := TMemoryStream.Create;
  try
    ServerSocket1.Port := 80; // port WWW
    ServerSocket1.Active := True;

    while not Terminated do begin
      ServiceThread.ProcessRequests(True);
    end;

    ServerSocket1.Active := False;
  finally
    Stream.Free;
  end;
end;
```

Quand vous écrivez votre application service, vous devez tenir compte des éléments suivants :

- Threads de service
- Propriétés de nom d'un service
- Débogage des services

Threads de service

Chaque service dispose de son propre thread (*TServiceThread*), donc si votre application service implémente plusieurs services, vous devez vous assurer que l'implémentation de vos services est compatible avec l'utilisation de threads. La classe *TServiceThread* est ainsi conçue de façon à implémenter le service dans le gestionnaire d'événement *OnExecute*de *TService*. Le thread du service dispose de sa propre méthode *Execute* qui contient une boucle appelant les gestionnaires *OnStart* et *OnExecute* du service avant de traiter de nouvelles requêtes.

Comme le traitement des requêtes de service peut prendre longtemps et que l'application service peut recevoir simultanément plusieurs requêtes d'un ou de plusieurs clients, il est plus efficace de lancer un nouveau thread (dérivé de *TThread* et non de *TServiceThread*) pour chaque requête et de déplacer l'implémentation du service dans la méthode *Execute* du nouveau thread. Cela permet à la boucle *Execute* du thread du service de traiter continuellement de nouvelles requêtes sans avoir à attendre la fin du gestionnaire *OnExecute* du service. Cette manière de procéder est illustrée par l'exemple suivant :

Exemple Ce service sonne tous les 500 millisecondes depuis le thread standard. Il gère la pause, la reprise et l'arrêt du thread quand on indique au service de se suspendre, de reprendre ou de s'arrêter.

1. Choisissez Fichier | Nouveau | Autre et sélectionnez Application Service dans la boîte de dialogue Nouveaux éléments. Une fenêtre nommée Service1 apparaît.
2. Dans la section interface de votre unité, déclarez un nouveau descendant de *TThread* nommé TSparkyThread. C'est le thread qui réalise le travail pour le service. Il doit être déclaré comme suit :

```
TSparkyThread = class(TThread)
    public
      procedure Execute; override;
  end;
```

3. Ensuite, dans la section implémentation de l'unité, créez une variable globale pour une instance de TSparkyThread :

```
var
  SparkyThread: TSparkyThread;
```

4. Ajoutez le code suivant à la section implémentation pour la méthode Execute de TSparkyThread (la fonction thread) :

```
procedure TSparkyThread.Execute;
begin
  while not Terminated do
  begin
    Beep;
    Sleep(500);
  end;
end;
```

5. Sélectionnez la fenêtre service (Service1) et double-cliquez sur l'événement OnStart dans l'inspecteur d'objets. Ajoutez le gestionnaire d'événement OnStart suivant :

```
procedure TService1.Service1Start(Sender: TService; var Started: Boolean);
begin
  SparkyThread := TSparkyThread.Create(False);
  Started := True;
end;
```

6. Double-cliquez sur l'événement OnContinue dans l'inspecteur d'objets. Ajoutez le gestionnaire d'événement OnContinue suivant :

```
procedure TService1.Service1Continue(Sender: TService; var Continued: Boolean);
```

```
begin
  SparkyThread.Resume;
  Continued := True;
end;
```

7 Double-cliquez sur l'événement OnPause dans l'inspecteur d'objets. Ajoutez le gestionnaire d'événement OnPause suivant :

```
procedure TService1.Service1Pause(Sender: TService; var Paused: Boolean);
begin
  SparkyThread.Suspend;
  Paused := True;
end;
```

8 Enfin, double-cliquez sur l'événement OnStop dans l'inspecteur d'objets. Ajoutez le gestionnaire d'événement OnStop suivant :

```
procedure TService1.Service1Stop(Sender: TService; var Stopped: Boolean);
begin
  SparkyThread.Terminate;
  Stopped := True;
end;
```

Dans le cadre du développement d'applications serveur, la décision de lancer un nouveau thread dépend de la nature du service rendu, du nombre prévu de connexions et du nombre prévu de processeurs dont dispose la machine exécutant le service.

Propriétés de nom d'un service

La VCL propose des classes permettant de créer des applications service sur le plate-forme Windows (non disponible pour les applications multiplates-formes). Il s'agit de *TService* et de *TDependency*. Quand vous utilisez ces classes, les diverses propriétés de nom peuvent être source de confusion. Cette section décrit leurs différences.

Les services ont des noms d'utilisateur (appelés Nom de démarrage du service) qui sont associés à des mots de passe, des noms d'affichage utilisés pour l'affichage dans les fenêtres gestionnaire et éditeur et des noms réels (le nom du service). Les dépendances peuvent être des services ou des groupes d'ordre de chargement. Elles ont également des noms et des noms d'affichage. De plus, comme les objets service dérivent de *TComponent*, ils héritent de la propriété *Name*. Les paragraphes suivants décrivent ces diverses propriétés de nom.

Propriétés de TDependency

La propriété *DisplayName* de *TDependency* est à la fois le nom d'affichage et le nom réel du service. Elle est presque toujours identique à la propriété *Name* *TDependency*.

Propriétés de nom de TService

La propriété *Name* de *TService* est héritée de *TComponent*. C'est le nom du composant et également le nom du service. Pour les dépendances qui sont des

services, cette propriété est identique aux propriétés *Name* et *DisplayName* de *TDependency*.

TService::*DisplayName* est le nom affiché dans la fenêtre du gestionnaire de service. Il diffère souvent du nom réel du service (*TService.NameTDependency.DisplayName*, *TDependency.Name*). Remarquez que généralement le nom d'affichage (*DisplayName)* n'est pas le même pour le service et pour la dépendance.

Les noms de démarrage de service sont distincts du nom d'affichage et du nom réel du service. Un *ServiceStartName* est la valeur saisie du nom d'utilisateur dans la boîte de dialogue de démarrage sélectionnée depuis le gestionnaire de contrôle de service.

Débogage des services

Le débogage des applications service peut être difficile car il nécessite des intervalles de temps courts :

1. Lancez d'abord l'application dans le débogueur. Patientez quelques secondes jusqu'à la fin du chargement.
2. Démarrez rapidement le service à partir du panneau de configuration ou de la ligne de commande :

```
start MyServ
```

Vous devez lancer le service rapidement (dans les 15 à 30 secondes du démarrage de l'application) car l'application se terminera si aucun service n'est lancé.

Une autre approche consiste à s'attacher au processus de l'application service lorsqu'elle est déjà en exécution. (C'est-à-dire, démarrer d'abord le service, puis effectuer l'attachement au débogueur). Pour effectuer l'attachement au processus d'application service, choisissez Exécuter | Attacher au processus et sélectionnez l'application service dans la boîte de dialogue résultante.

Dans certains cas, cette seconde approche peut échouer en raison de droits insuffisants. Si cela se produit, vous pouvez utiliser le gestionnaire de contrôle de service pour permettre à votre service de fonctionner avec le débogueur :

1. Créez une clé (options d'exécution du fichier image) dans l'emplacement de registre suivant :
   `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion`
2. Créez une sous-clé portant le même nom que votre service (par exemple, MONSERV.EXE). Ajoutez à cette sous-clé une valeur de type REG_SZ, nommée Debugger. Utilisez le chemin complet de Delphi32.exe comme valeur de chaîne.
3. Dans l'applet Services du panneau de configuration, sélectionnez votre service, cliquez sur l'option de démarrage et activez l'option permettant au service d'interagir avec le bureau.

Création de paquets et de DLL

Les bibliothèques de liaison dynamique (DLL) sont des modules de code compilés qui fonctionnent en conjonction avec un exécutable pour proposer des fonctionnalités à une application. Vous pouvez créer des DLL dans des programmes multiplates-formes. Cependant, sous Linux, les DLL et les paquets sont recompilés en tant qu'objets partagés.

Les paquets sont des DLL spéciales utilisées par les applications Delphi, par l'EDI ou les deux à la fois. Il y a deux sortes de paquets : les paquets d'exécution et les paquets de conception. Les paquets d'exécution fournissent des fonctionnalités à un programme lors de son exécution. Les paquets de conception permettent d'étendre les fonctionnalités de l'EDI.

Les directives de compilation suivantes peuvent être placées dans les fichiers des projets bibliothèque :

Tableau 5.1 Directives de compilation pour les bibliothèques

Directive de compilation	Description
{$LIBPREFIX 'chaîne'}	Ajoute le préfixe spécifié au nom du fichier de destination. Par exemple, vous pouvez spécifier {$LIBPREFIX 'dcl'} pour un paquet de conception, ou utiliser {$LIBPREFIX ' '} pour éliminer totalement le préfixe.
{$LIBSUFFIX 'chaîne'}	Ajoute le suffixe spécifié au nom du fichier de destination, avant l'extension. Par exemple, utilisez {$LIBSUFFIX '-2.1.3'} dans quelquechose.pas pour générer quelquechose-2.1.3.bpl.
{$LIBVERSION 'chaîne'}	Ajoute une seconde extension au nom du fichier de destination après l'extension .bpl. Par exemple, utilisez {$LIBVERSION '2.1.3'} dans quelquechose.pas pour générer quelquechose.bpl.2.1.3.

Pour davantage d'informations sur les paquets, voir chapitre 11, "Utilisation des paquets et des composants".

Utilisation des paquets et des DLL

Pour la plupart des applications écrites avec Delphi, les paquets offrent une plus grande flexibilité et sont plus simples à créer que les DLL. Dans certaines situations, les DLL sont mieux adaptées à vos projets que des paquets :

- Votre module de code doit être appelé par une application qui n'a pas été conçue avec Delphi.
- Vous étendez les fonctionnalités d'un serveur Web.
- Vous créez un module de code qui doit être utilisé par des développeurs extérieurs.
- Votre projet est un conteneur OLE.

Vous ne pouvez pas transmettre des informations de type à l'exécution (RTTI) entre DLL ou d'une DLL à un exécutable. C'est parce que les DLL maintiennent leurs propres informations sur les symboles. Si vous avez besoin de transmettre un objet *TStrings* à partir d'une DLL en utilisant un opérateur **is** ou **as**, créez un paquet plutôt qu'une DLL. Les paquets partagent les informations sur les symboles.

Ecriture d'applications de bases de données

Remarque Les versions de Delphi n'offrent pas toutes le support des bases de données.

Un des atouts de Delphi est sa possibilité de créer des applications de bases de données sophistiquées. Delphi fournit des outils intégrés permettant de vous connecter aux serveurs et bases de données SQL, comme Oracle, Sybase, InterBase, MySQL, MS-SQL, Informix et DB2, tout en assurant un partage des données transparent entre les différentes applications.

Delphi comprend de nombreux composants permettant d'accéder aux bases de données et de représenter les informations qu'elles contiennent. Sur la palette de composants, les composants base de données sont regroupés selon le mécanisme et la fonction d'accès aux données.

Tableau 5.2 Page base de données de la palette de composants

Page de palette	Contenu
BDE	Composants qui utilisent le moteur de bases de données Borland (BDE), une importante API permettant d'interagir avec les bases de données. Le moteur BDE supporte la plus vaste gamme de fonctions et la plupart des utilitaires dont Database Desktop, l'explorateur de base de données, le moniteur SQL et l'administrateur BDE. Pour plus de détails, voir chapitre 20, "Utilisation du moteur de bases de données Borland".
ADO	Composants qui utilisent les objets de données ActiveX (ADO), développés par Microsoft, pour accéder aux informations des bases de données. De nombreux pilotes ADO sont disponibles pour la connexion à différents serveurs de bases de données. Les composants ADO vous permettent d'intégrer votre application à l'environnement ADO. Pour plus de détails, voir chapitre 21, "Utilisation des composants ADO".
dbExpress	Composants multiplates-formes qui utilisent dbExpress pour accéder aux informations des bases de données. Les pilotes dbExpress fournissent un accès rapide aux bases de données, mais doivent être utilisés avec *TClientDataSet* et *TDataSetProvider* pour effectuer des mises à jour. Pour plus de détails, voir chapitre 22, "Utilisation d'ensembles de données unidirectionnels".
InterBase	Composants qui accèdent directement aux bases de données InterBase, sans passer par une couche moteur distincte.Pour plus d'informations sur l'utilisation des composants InterBase, voir l'aide en ligne.

Tableau 5.2 Page base de données de la palette de composants (suite)

Page de palette	Contenu
AccèsBD	Composants qui peuvent être utilisés avec n'importe quel mécanisme d'accès aux données comme *TClientDataSet* et *TDataSetProvider*. Voir chapitre 23, "Utilisation d'ensembles de données client", pour avoir des informations sur les ensembles de données client. Voir chapitre 24, "Utilisation des composants fournisseur"pour avoir des informations sur les fournisseurs.
ContrôleBD	Contrôles orientés données qui peuvent accéder aux informations d'une source de données. Voir chapitre 15, "Utilisation de contrôles de données", pour plus de détails.

Lorsque vous concevez une application de base de données, vous devez choisir le mécanisme d'accès aux données à utiliser. Chaque mécanisme d'accès aux données diffère par l'éventail des fonctions prises en charge, la facilité de déploiement et la capacité des pilotes à gérer divers serveurs de bases de données.

Voir partie II, "Développement d'applications de bases de données", dans ce manuel, pour plus de détails sur la façon d'utiliser Delphi pour créer des applications base de données client ou serveur. Reportez-vous à "Déploiement d'applications de bases de données" à la page 13-7, pour avoir des informations sur le déploiement.

Distribution d'applications de bases de données

Delphi permet de créer des applications de bases de données distribuées en utilisant un ensemble de composants liés. Il est ainsi possible d'écrire des applications de bases de données en utilisant divers protocoles de communication, dont DCOM, CORBA, TCP/IP et SOAP.

Pour davantage d'informations sur la conception d'applications de bases de données distribuées, voir chapitre 25, "Création d'applications multiniveaux".

Le déploiement d'applications de bases de données nécessite souvent le déploiement du moteur de bases de données Borland (BDE) en plus des fichiers de l'application. Pour des informations sur le déploiement du BDE, voir "Déploiement d'applications de bases de données" à la page 13-7.

Création d'applications serveur Web

Les applications serveur Web sont des applications s'exécutant sur des serveurs qui fournissent du contenu web, c'est-à-dire des pages HTML ou des documents XML, sur Internet. Les applications serveur Web sont par exemple des applications qui contrôlent l'accès à un site web, génèrent des bons de commande ou répondent à des demandes d'informations.

Vous pouvez créer différents types d'applications serveur Web en utilisant les technologies Delphi suivantes :

- Agent Web
- WebSnap
- InternetExpress
- Services Web

Utilisation de l'agent Web

Vous pouvez utiliser l'agent Web (appelé aussi architecture NetCLX) pour créer des applications serveur Web comme les applications CGI ou les bibliothèques de liaison dynamiques (DLL). Ces applications serveur web peuvent ne contenir aucun composant visuel. Les composants de la page Internet de la palette de composants vous permettent de créer des gestionnaires d'événements, de construire par programme des documents HTML ou XML et de les transférer au client.

Pour créer une nouvelle application serveur Web en utilisant l'architecture agent Web, sélectionnez Fichier | Nouveau | Autre et sélectionnez Application serveur Web dans la boîte de dialogue Nouveaux éléments. Choisissez ensuite le type d'application serveur Web :

Tableau 5.3 Applications serveur Web

Type d'application serveur Web	Description
DLL ISAPI/NSAPI	Les applications serveur Web ISAPI et NSAPI sont des DLL qui sont chargées par le serveur Web. Les informations de requête client sont transmises à la DLL sous forme de structure et sont évaluées par TISAPIApplication. Chaque message de requête est géré dans un thread d'exécution distinct. Sélectionner ce type d'application ajoute l'en-tête bibliothèque des fichiers du projet et les entrées nécessaires à la liste uses et à la clause exports du fichier projet.
Exécutable autonome CGI	Les applications serveur Web CGI sont des applications console qui reçoivent les requêtes des clients sur l'entrée standard, traitent ces requêtes et renvoient le résultat sur la sortie standard afin qu'il soit renvoyé au client. Sélectionner ce type d'application ajoute les entrées nécessaires à la clause **uses** du fichier projet et ajoute au source la directive $APPTYPE appropriée.
Exécutable autonome Win-CGI	Les applications serveur Web Win-CGI sont des applications Windows qui reçoivent les requêtes des clients à partir d'un fichier de paramètres de configuration (INI) écrit par le serveur et qui écrivent le résultat dans un fichier que le serveur renvoie au client. Le fichier INI est évalué par TCGIApplication. Chaque message de requête est géré par une instance distincte de l'application. Sélectionner ce type d'application ajoute les entrées nécessaires à la clause **uses** du fichier projet et ajoute au source la directive $APPTYPE appropriée.

Tableau 5.3 Applications serveur Web (suite)

Type d'application serveur Web	Description
Module Apache partagé (DLL)	La sélection de ce type d'application configure votre projet comme une DLL. Les applications serveur Web Apache sont des DLL chargées par le serveur Web. Les informations sont transmises à la DLL, traitées puis renvoyées au client par le serveur Web.
Exécutable débogueur d'application Web	La sélection de ce type d'application configure un environnement pour développer et tester des applications serveur Web. Les applications Débogueur d'application Web sont des fichiers exécutables chargées par le serveur Web. Ce type d'application n'est pas destiné au déploiement.

Les applications CGI et Win-CGI utilisent davantage de ressources système sur le serveur, les applications complexes sont mieux gérées si elles sont créées sous la forme d'applications ISAPI, NSAPI ou Apache DLL. Si vous écrivez des applications multiplates-formes, vous devez sélectionner autonome CGI ou Module Apache partagé (DLL) pour le développement de serveur Web. Vous avez les mêmes options lorsque vous créez des applications WebSnap et service Web.

Pour plus d'informations sur la construction d'applications serveur Web, voir chapitre 27, "Création d'applications Internet".

Création d'applications WebSnap

WebSnap fournit un ensemble de composants et d'experts permettant de construire des serveurs Web évolués qui interagissent avec les navigateurs web. Les composants WebSnap génèrent du HTML ou d'autre contenu mime pour les pages Web. WebSnap est destiné au développement côté serveur. WebSnap ne peut pas, pour l'instant, être utilisé dans des applications multiplates-formes.

Pour créer une nouvelle application WebSnap, sélectionnez Fichier | Nouveau | Autre et sélectionnez l'onglet WebSnap dans la boîte de dialogue Nouveaux éléments. Choisissez Application WebSnap. Ensuite, sélectionnez le type d'application serveur Web (ISAPI/NSAPI, CGI, Win-CGI, Apache). Voir le tableau 5.3, "Applications serveur Web", pour plus de détails.

Pour plus d'informations sur WebSnap, voir chapitre 29, "Utilisation de WebSnap".

Utilisation d'InternetExpress

InternetExpress est un ensemble de composants permettant d'étendre l'architecture d'application serveur Web de base pour qu'elle agisse en tant que client d'une application serveur. Vous utilisez InternetExpress pour les applications dans lesquelles les clients dans un navigateur peuvent accéder aux données d'un fournisseur, résoudre les mises à jour du fournisseur tout en s'exécutant sur un client.

Les applications InternetExpress génèrent des pages HTML qui associent HTML, XML et javascript. HTML régit la disposition et l'aspect des pages affichées dans le navigateur des utilisateurs finals. XML code les paquets de données et les paquets delta qui représentent les informations base de données. Javascript permet aux contrôles HTML d'interpréter et de manipuler les données des paquets XML sur la machine client.

Pour plus d'informations sur InternetExpress, voir "Construction des applications Web avec InternetExpress" à la page 25-38.

Création d'applications services Web

Les services Web sont des applications modulaires indépendantes qui peuvent être publiées ou invoquées sur un réseau (comme le web). Les services Web fournissent des interfaces bien définies qui décrivent les services fournis. Vous utilisez les services Web pour fournir ou utiliser des services programmables sur Internet en faisant appel aux derniers standards comme XML, XML Schema, SOAP (Simple Object Access Protocol) et WSDL (Web Service Definition Language).

Les services Web utilisent SOAP, un protocole léger standard permettant d'échanger des informations dans une environnement distribué. Il utilise HTTP comme protocole de communication et XML pour coder les appels des procédures distantes.

Vous pouvez utiliser Delphi pour construire des serveurs qui implémentent des services Web et des clients qui font appel à ces services. Vous pouvez écrire des clients pour que des serveurs quelconques implémentent des services Web qui répondent aux messages SOAP et des serveurs Delphi pour publier des services Web à utiliser par des clients quelconques.

Pour plus d'informations sur services Web, voir chapitre 31, "Utilisation de services Web".

Ecriture d'applications en utilisant COM

COM (Component Object Model) propose une architecture d'objet distribué sous Windows conçue pour assurer une interopérabilité des objets en utilisant des routines prédéfinies appelées des interfaces. Les applications COM utilisent des objets implémentés par un processus différent ou, si vous utilisez DCOM, sur une machine différente. Vous pouvez aussi utiliser COM+, ActiveX et les pages Active Server.

COM est un modèle de composant logiciel indépendant du langage qui permet l'interaction entre des composants logiciels et des applications s'exécutant sous Windows. L'aspect fondamental de COM est de permettre la communication entre composants, entre applications et entre clients et serveurs, par le biais d'interfaces clairement définies. Les interfaces offrent aux clients un moyen de demander à un composant COM quelles fonctionnalités il supporte à l'exécution.

Pour fournir d'autres fonctionnalités à votre composant, il suffit d'ajouter une autre interface pour ces fonctionnalités.

Utilisation de COM et de DCOM

Delphi contient des classes et des experts qui simplifient la création d'applications COM, OLE ou ActiveX. Vous pouvez créer des clients ou des serveurs COM qui implémentent des objets COM, des serveurs d'automatisation (dont les objets Active Server), des contrôles ActiveX ou des ActiveForms. COM sert également de base à d'autres technologies comme l'automatisation, les contrôles ActiveX, les documents Active et les répertoires Active.

L'utilisation de Delphi pour créer des applications basées sur COM offre de nombreuses possibilités, allant de l'amélioration de la conception de logiciel en utilisant des interfaces de manière interne dans une application, à la création d'objets qui peuvent interagir avec d'autres objets utilisant l'API COM du système, comme les extensions du shell Win9x ou la gestion multimedia DirectX. Les applications peuvent accéder aux interfaces des composants COM se trouvant sur le même ordinateur que l'application ou sur un autre ordinateur du réseau, en utilisant un mécanisme nommé DCOM (Distributed COM).

Pour davantage d'informations sur COM et les contrôles ActiveX, voir chapitre 33, "Présentation des technologies COM", chapitre 38, "Création d'un contrôle ActiveX" et "Distribution d'une application client en tant que contrôle ActiveX" à la page 25-37.

Pour davantage d'informations sur DCOM, voir "Utilisation de connexions DCOM" à la page 25-10.

Utilisation de MTS et de COM+

Il est possible d'étendre les applications COM en utilisant des services spéciaux pour gérer les objets dans un environnement distribué important. Ces services sont, entre autres, des services de transaction, la sécurité et la gestion des ressources proposées par Microsoft Transaction Server (MTS) (pour les versions de Windows antérieures à Windows 2000) ou COM+ (pour Windows 2000 ou plus).

Pour davantage d'informations sur MTS et COM+, voir chapitre 39, "Création d'objets MTS ou COM+" et "Utilisation des modules de données transactionnels" à la page 25-7.

Utilisation de modules de données

Un module de données ressemble à une fiche spéciale qui ne contient que des composants non-visuels. Tous les composants d'un module de données *peuvent* être placés dans des fiches ordinaires avec des contrôles visuels. Les modules de données constituent un moyen d'organisation utile quand vous envisagez de

réutiliser des groupes d'objets de base de données ou système ou si vous souhaitez isoler les parties d'une application qui gèrent la connexion aux bases de données ou les règles de fonctionnement.

Il y a plusieurs types de modules de données, dont standard, distant, modules Web, modules applet et services, selon l'édition de Delphi que vous avez. Chaque type de module de données a une fonction spéciale.

- Les modules de données standard sont particulièrement utiles aux applications de bases de données à un ou à deux niveaux, mais peuvent être utilisés pour organiser les composants non visuels de n'importe quelle application. Pour plus d'informations, voir "Création et modification de modules de données standard" à la page 5-17.
- Les modules de données distants constituent la base d'un serveur d'application dans une application de base de données multiniveau. Ils ne sont pas disponibles dans toutes les éditions. En plus de contenir les composants non visuels du serveur d'application, les modules de données distants exposent l'interface utilisée par les clients pour communiquer avec le serveur d'application. Pour plus d'informations sur leur utilisation, voir "Ajout d'un module de données distant à un projet serveur d'application" à la page 5-21.
- Les modules Web constituent la base des applications serveur Web. En plus de contenir les composants qui créent le contenu des messages de réponse HTTP, ils gèrent la répartition des messages HTTP issus des applications client. Voir chapitre 27, "Création d'applications Internet" pour plus d'informations sur l'utilisation des modules Web.
- Les modules applet constituent la base des applets de panneau de configuration. En plus de contenir les composants non visuels qui implémentent le panneau de configuration, ils définissent les propriétés qui déterminent la façon dont l'icône de l'applet apparaît dans le panneau de configuration et contiennent les événements qui sont appelés quand les utilisateurs exécutent l'applet. Pour plus d'informations sur les modules applet, voir l'aide en ligne.
- Les services encapsulent des services individuels dans une application de service NT. En plus de contenir les composants non visuels qui implémentent un service, les services contiennent les événements qui sont appelés quand le service est démarré ou arrêté. Pour davantage d'informations sur les services, voir "Applications service" à la page 5-4.

Création et modification de modules de données standard

Pour créer un module de données standard pour un projet, choisissez Fichier | Nouveau | Module de données. Delphi ouvre un conteneur module de données sur le bureau, affiche le fichier unité du nouveau module dans l'éditeur de code et ajoute le module au projet en cours.

En conception, un module de données ressemble à une fiche Delphi standard, avec un fond blanc et pas de grille d'alignement. Comme avec les fiches, vous

pouvez placer des composants non visuels de la palette dans un module et modifier leurs propriétés dans l'inspecteur d'objets. Vous pouvez redimensionner un module de données pour l'adapter aux composants que vous lui ajoutez.

Vous pouvez aussi cliquer avec le bouton droit sur un module pour afficher son menu contextuel. Le tableau suivant résume les options du menu contextuel d'un module de données :

Tableau 5.4 Options du menu contextuel des modules de données

Elément de menu	Utilisation
Edition	Affiche un menu contextuel grâce auquel vous pouvez couper, copier, coller, supprimer et sélectionner les composants du module de données.
Position	Aligne les composants s non visuels sur la grille invisible du module (*Aligner sur la grille) ou*selon des critères que vous indiquez dans la boîte de dialogue Alignement (*Aligner*).
Ordre de tabulation	Vous permet de changer l'ordre dans lequel la focalisation parcourt les composants quand vous appuyez sur la touche Tab.
Ordre de création	Vous permet de modifier l'ordre dans lequel les composants d'accès aux données sont créés au démarrage.
Revenir à hérité	Annule les modifications apportées à un module hérité d'un autre module dans le référentiel d'objets, et revient au module tel qu'il était à l'origine lors de l'héritage.
Ajouter au référentiel	Stocke un lien vers le module de données dans le référentiel d'objets.
Voir comme texte	Affiche la représentation textuelle des propriétés du module de données.
DFM texte	Bascule entre les formats (binaire ou texte) dans lesquels est enregistrée la fiche.

Pour davantage d'informations sur les modules de données, voir l'aide en ligne.

Nom d'un module de données et de son fichier unité

La barre de titre d'un module de données affiche le nom du module. Le nom par défaut est "DataModule*N*", où *N* est un nombre représentant le plus petit numéro d'unité non utilisé dans le projet. Par exemple, si vous commencez un nouveau projet et si vous lui ajoutez un module avant de faire quoi que ce soit d'autre, le nom du module de données est par défaut "DataModule2". Le fichier unité correspondant à *DataModule2* est par défaut "Unit2".

Il vaut mieux renommer les modules de données et les fichiers unités correspondants, pendant la conception, pour qu'ils soient plus descriptifs. En particulier, il faut renommer les modules de données que vous ajoutez au référentiel d'objets pour éviter les conflits de nom avec d'autres modules de données du référentiel ou des applications qui utilisent vos modules.

Pour renommer un module de données :

1. Sélectionnez le module.
2. Modifiez la propriété *Name* du module dans l'inspecteur d'objets.

Le nouveau nom du module apparaît dans la barre de titre dès que la propriété *Name* n'a plus la focalisation dans l'inspecteur d'objets.

Changer le nom d'un module de données en conception change son nom de variable dans la section interface du code. Cela change également toute utilisation du nom dans les déclarations de procédure. Vous devez changer manuellement toutes les références à ce module de données dans le code que vous avez écrit.

Pour renommer le fichier unité d'un module de données :

1 Sélectionnez le fichier unité.

Placer et nommer les composants

Vous placez des composants non visuels dans un module de données exactement comme vous les placeriez sur une fiche. Cliquez sur le composant voulu dans la page appropriée de la palette de composants, puis cliquez dans le module de données pour placer le composant. Vous ne pouvez pas placer des contrôles visuels, comme les grilles, dans un module de données. Si vous essayez de le faire, vous recevez un message d'erreur.

Pour faciliter leur utilisation, les composants d'un module de données sont affichés avec leur nom. Quand vous placez un composant pour la première fois, Delphi lui attribue un nom générique indiquant le type du composant, suivi de *1*. Par exemple, le composant *TDataSource* prend le nom *DataSource1*. Cela facilite la sélection des composants lorsque vous voulez travailler sur leurs propriétés et méthodes.

Vous pouvez encore donner à un composant un autre nom qui reflète son type et sa fonction.

Pour changer le nom d'un composant dans un module de données :

1 Sélectionnez le composant.
2 Modifiez la propriété *Name* du composant dans l'inspecteur d'objets.

Le nouveau nom du composant apparaît sous son icône dans le module de données dès que la propriété *Name* n'a plus la focalisation dans l'inspecteur d'objets.

Par exemple, supposons que votre application de base de données utilise la table CUSTOMER. Pour accéder à cette table, vous avez besoin d'au moins deux composants d'accès aux données : un composant source de données (*TDataSource)* et un composant table (*TClientDataSet)*. Quand vous placez ces composants dans votre module de données, Delphi leur attribue les noms *DataSource1* et *ClientDataSet1*. Pour refléter le type des composants et la base de données à laquelle ils accèdent, CUSTOMER, vous pourriez changer leurs noms en *CustomerSource* et *CustomerTable*.

Utilisation des propriétés et événements des composants dans un module de données

Placer des composants dans un module de données centralise leur comportement pour l'application toute entière. Par exemple, vous pouvez utiliser les propriétés des composants ensemble de données, comme *TClientDataSet*, pour contrôler les données disponibles dans les composants source de données qui utilisent ces ensembles de données. Définir la propriété *ReadOnly* d'un ensemble de données par *True* empêche les utilisateurs de modifier les données qu'ils voient dans un contrôle orienté données se trouvant dans une fiche. Vous pouvez aussi appeler l'éditeur de champs d'un ensemble de données, en double-cliquant sur *ClientDataSet1*, pour limiter les champs d'une table ou d'une requête qui seront disponibles pour la source de données et donc pour les contrôles orientés données des fiches. Les propriétés que vous définissez pour les composants d'un module de données s'appliquent à toutes les fiches de votre application qui utilisent le module.

Outre les propriétés, vous pouvez écrire des gestionnaires d'événements pour les composants. Par exemple, un composant *TDataSource* peut avoir trois événements : *OnDataChange*, *OnStateChange* et *OnUpdateData*. Un composant *TClientDataSet* a plus de 20 événements possibles. Vous pouvez utiliser ces événements pour créer un ensemble cohérent de règles de gestion qui dictent les manipulation de données dans toute votre application.

Création de règles de gestion dans un module de données

Parallèlement à l'écriture de gestionnaires d'événements pour les composants d'un module de données, vous pouvez programmer des méthodes directement dans le fichier unité du module de données. Ces méthodes peuvent être appliquées en tant que règles de gestion aux fiches qui utilisent le module de données. Par exemple, vous pouvez écrire une procédure qui établisse les comptes mensuels, trimestriels et annuels. Vous pouvez appeler cette procédure depuis un gestionnaire d'événement d'un composant du module de données. Les prototypes des procédures et fonctions que vous écrivez pour un module de données doivent figurer dans la déclaration de **type** du module :

```
type
  TCustomerData = class(TDataModule)
    Customers: TClientDataSet;
    Orders: TClientDataSet;
    ⋮
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
    procedure LineItemsCalcFields(DataSet: TDataSet); { Un procédure que vous ajoutez }
  end;

var
  CustomerData: TCustomerData;
```

Les procédures et fonctions que vous écrivez doivent suivre la section implémentation du code du module.

Accès à un module de données depuis une fiche

Pour associer des contrôles visuels d'une fiche à un module de données, vous devez tout d'abord ajouter le module de données à la clause **uses** de la fiche. Pour ce faire, vous pouvez procéder de plusieurs manières :

- Ouvrez le fichier unité de la fiche dans l'éditeur de code et ajoutez le nom du module de données à la clause **uses** de la section **interface**.
- Cliquez sur le fichier unité de la fiche, choisissez Fichier | Utiliser l'unité, puis entrez le nom d'un module ou choisissez-le dans la boîte liste de la boîte de dialogue Utiliser l'unité.
- Pour les composants base de données, cliquez dans le module de données sur un composant ensemble de données ou requête pour ouvrir l'éditeur de champs et faire glisser dans la fiche des champs de l'éditeur. Delphi vous demande alors de confirmer l'ajout de ce module dans la clause **uses** de la fiche puis crée des contrôles (par exemple, des boîtes de saisie) pour chaque champ.

Par exemple, si vous avez ajouté le composant *TClientDataSet* à votre module de données, double-cliquez sur lui pour ouvrir l'éditeur de champs. Sélectionnez un champ et faites-le glisser dans la fiche. Une boîte de saisie apparaît.

Comme la source de données n'est pas encore définie, Delphi ajoute un nouveau composant source de données, *DataSource1*, à la fiche et définit la propriété *DataSource* de la boîte de saisie par *DataSource1*. La source de données définit automatiquement sa propriété *DataSet* par le composant ensemble de données, *ClientDataSet1*, dans le module de données.

Vous pouvez définir la source de données *avant* de faire glisser un champ sur la fiche, en ajoutant un composant *TDataSource* au module de données. Définissez la propriété *DataSet* de la source de données par *ClientDataSet1.* Une fois que vous avez fait glissé un champ dans la fiche, la boîte de saisie apparaît avec sa propriété *TDataSource* déjà définie par *DataSource1*. Grâce à cette méthode, votre modèle d'accès aux données est plus propre.

Ajout d'un module de données distant à un projet serveur d'application

Certaines éditions de Delphi vous permettent d'ajouter des *modules de données distants* à des projets de serveur d'applications. Un module de données distant dispose d'une interface à laquelle les clients d'une application multiniveau peuvent accéder au travers d'un réseau.

Pour ajouter un module de données distant à un projet :

1 Choisissez Fichier | Nouveau | Autre.

2 Sélectionnez la page Multi-niveaux dans la boîte de dialogue Nouveaux éléments.

3 Double-cliquez sur le type de module voulu (Module de données CORBA, Module de données distant ou Module de données transactionnel) pour ouvrir l'expert Module de données distant.

Une fois le module de données distant ajouté à un projet, vous pouvez l'utiliser comme un module de données standard.

Pour davantage d'informations sur les applications de bases de données multiniveaux, voir chapitre 25, "Création d'applications multiniveaux".

Utilisation du référentiel d'objets

Le référentiel d'objets (Outils | Référentiel) vous permet facilement de partager ou de copier des fiches, des boîtes de dialogue ou des modules de données. Il propose également des modèles de projet comme point de départ pour de nouveaux projets et des experts qui guident l'utilisateur dans la création de fiches ou de projets. Le référentiel est stocké dans DELPHI32.DRO (placé par défaut dans le répertoire BIN), c'est un fichier texte qui contient des références aux éléments apparaissant dans le référentiel et dans la boîte de dialogue Nouveaux éléments.

Partage d'éléments dans un projet

Il est également facile de partager des éléments *à l'intérieur* d'un projet sans avoir à les ajouter au référentiel d'objets. quand vous ouvrez la boîte de dialogue Nouveaux éléments (Fichier | Nouveau | Autre), l'onglet d'une des pages porte le nom de votre projet. Cette page énumère toutes les fiches, boîtes de dialogue et modules de données de votre projet. Vous pouvez alors dériver un nouvel élément d'un élément existant et le personnaliser si nécessaire.

Ajout d'éléments au référentiel d'objets

Vous pouvez ajouter vos propres projets, fiches, cadres et modules de données à ceux qui existent déjà dans le référentiel d'objets. Pour ajouter un élément au référentiel d'objets,

1 Si l'élément est un projet ou dans un projet, ouvrez le projet.

2 Pour un projet, choisissez Projet | Ajouter au référentiel. Pour une fiche ou un module de données, cliquez sur l'élément avec le bouton droit de la souris puis choisissez Ajouter au référentiel.

3 Entrez une description, un titre et un auteur.

4 Décidez dans quelle page cet élément doit apparaître dans la boîte de dialogue Nouveaux éléments, entrez ou sélectionnez la page dans la boîte à

options Page. Si vous entrez un nom de page inexistant, Delphi crée une nouvelle page.

5 Choisissez Parcourir pour sélectionner une icône représentant l'objet dans le référentiel d'objets.

6 Choisissez OK.

Partage d'objets par une équipe de développement

Vous pouvez partager des objets dans un groupe de travail ou une équipe de développement en utilisant un référentiel accessible depuis un réseau. Pour utiliser un référentiel partagé, tous les membres de l'équipe doivent sélectionner le même répertoire de Référentiel partagé dans la boîte de dialogue Options d'environnement :

1 Choisissez Outils | Options d'environnement.

2 Dans la page Préférences, repérez le volet Référentiel partagé. Dans le volet boîte de saisie Répertoire, entrez le nom du répertoire où doit se trouver le référentiel partagé. Assurez-vous que le répertoire spécifié est bien accessible pour tous les membres de l'équipe.

Lors de l'ajout du premier élément au référentiel, Delphi crée, s'il n'existe pas déjà, un fichier DELPHI32.DRO dans le répertoire spécifié par Référentiel partagé.

Utilisation d'un élément du référentiel d'objets dans un projet

Pour accéder aux éléments du référentiel d'objets, choisissez Fichier | Nouveau. La boîte de dialogue Nouveaux éléments apparaît et affiche tous les éléments disponibles. Selon le type d'élément que vous souhaitez utiliser, il y a jusqu'à trois options pour ajouter un élément à votre projet :

- Copier
- Hériter
- Utiliser

Copie d'un élément

Choisissez Copier pour obtenir une réplique exacte de l'élément sélectionné et ajouter la copie à votre projet. Les modifications ultérieures de l'élément du référentiel d'objets ne sont pas répercutées sur votre copie. De même, les modifications apportées à la copie n'affectent pas l'élément original dans le référentiel d'objets.

Copier est la seule option possible pour les modèles de projet.

Héritage d'un élément

Choisissez Hériter pour dériver une nouvelle classe de l'élément sélectionné dans le référentiel d'objets et ajouter la nouvelle classe à votre projet. Quand vous

recompilez votre projet, toutes les modifications apportées à l'élément du référentiel d'objets sont reportées dans votre classe dérivée. Les modifications faites dans la classe dérivée n'affectent pas l'élément partagé du référentiel d'objets.

Hériter est une option proposée pour les fiches, les boîtes de dialogue et les modules de données, mais pas pour les modèles de projet. C'est la *seule* option utilisable pour réutiliser les éléments du projet en cours.

Utilisation d'un élément

Choisissez Utiliser quand vous voulez que l'objet sélectionné fasse lui-même partie de votre projet. Les modifications faites à l'élément apparaissent dans tous les projets dans lesquels l'élément a été ajouté en utilisant l'option Hériter ou Utiliser. Soyez prudent si vous choisissez cette option.

L'option Utiliser est disponible pour les fiches, les boîtes de dialogue et les modules de données.

Utilisation de modèles de projet

Les modèles de projet sont des projets préfabriqués que vous pouvez utiliser comme point de départ pour la création de vos projets. Pour créer un nouveau projet à partir d'un modèle,

1. Choisissez Fichier | Nouveau | Autre pour afficher la boîte de dialogue Nouveaux éléments.
2. Choisissez l'onglet Projets.
3. Sélectionnez le modèle de projet souhaité et choisissez OK.
4. Dans la boîte de dialogue Sélection du répertoire, spécifiez le répertoire des fichiers du nouveau projet.

Delphi copie les fichiers du modèle dans le répertoire spécifié, où vous pouvez ensuite les modifier. Le modèle de projet initial n'est pas affecté par vos modifications.

Modification d'éléments partagés

Si vous modifiez un élément du référentiel d'objets, vos modifications affectent tous les projets qui ultérieurement utilisent l'élément mais également tous les projets existants qui ont ajouté l'élément en utilisant les options Utiliser ou Hériter. Pour éviter de propager des modifications à d'autres projets, vous avez plusieurs solutions :

- Copier l'élément et le modifier uniquement dans le projet en cours.
- Copier l'élément dans le projet en cours, le modifier puis l'ajouter au référentiel sous un autre nom.

- Créer un composant, une DLL, un modèle de composant ou un cadre à partir de l'élément. Si vous créez un composant ou une DLL, vous pouvez le partager avec d'autres développeurs.

Spécification d'un projet par défaut, d'une nouvelle fiche et de la fiche principale

Par défaut, quand vous choisissez Fichier | Nouveau | Application ou Fichier | Nouveau | Fiche, Delphi affiche une fiche vide. Vous pouvez changer ce comportement en reconfigurant le référentiel :

1. Choisissez Outils | Référentiel
2. Si vous voulez spécifier un projet par défaut, sélectionnez la page Projets et choisissez un élément dans Objets. Sélectionnez ensuite la case à cocher Nouveau projet.
3. Pour spécifier une fiche par défaut, sélectionnez une fiche dans Objets. Pour spécifier la nouvelle fiche par défaut, (Fiche | Nouveau | Fiche), sélectionnez la case à cocher Nouvelle fiche. Pour spécifier la nouvelle fiche principale par défaut des nouveaux projets, sélectionnez la case à cocher Fiche principale.
4. Choisissez OK.

Activation de l'aide dans les applications

La VCL et CLX supportent toutes deux l'affichage de l'aide à partir d'applications utilisant un mécanisme à base d'objets pour transmettre les demandes d'aide à l'un des multiples visualiseurs d'aide externes. Pour supporter cela, une application doit contenir une classe qui implémente l'interface *ICustomHelpViewer* (et, éventuellement, une des nombreuses interfaces qui en descendent), puis se recense elle-même dans le gestionnaire d'aide global.

La VCL fournit à toutes les applications une instance de *TWinHelpViewer*, qui implémente toutes ces interfaces et fournit un lien entre les applications et WinHelp ; CLX nécessite que les développeurs d'applications fournissent leur propre implémentation.

Le gestionnaire d'aide maintient la liste des visualiseurs recensés et leur passe les requêtes dans un processus en deux phases : d'abord, il demande à chaque visualiseur s'il est capable de fournir du support sur un mot clé ou un contexte d'aide particulier ; ensuite, il passe la requête d'aide au visualiseur ayant indiqué qu'il peut fournir un tel support. Si plusieurs visualiseurs supportent le mot clé (comme c'est la cas des applications ayant recensé des visualiseurs et pour Man et pour Info), le gestionnaire d'aide peut afficher une boîte de sélection dans laquelle l'utilisateur de l'application choisit le visualiseur d'aide à invoquer. Sinon, il affiche le premier système d'aide lui ayant répondu).

Interfaces avec les systèmes d'aide

Le système d'aide permet la communication entre votre application et les visualiseurs d'aide via une série d'interfaces. Ces interfaces sont toutes définies dans HelpIntfs.pas, qui contient également l'implémentation du gestionnaire d'aide.

ICustomHelpViewer prend en charge l'affichage de l'aide selon le mot clé fourni et l'affichage d'un sommaire listant toute l'aide disponible dans un visualiseur particulier.

IExtendedHelpViewer prend en charge l'affichage de l'aide selon le numéro de contexte fourni et l'affichage des rubriques ; dans la majorité des systèmes d'aide, les rubriques fonctionnent comme des mots clés de haut niveau (par exemple, "IntToStr" pourrait être un mot clé dans le système d'aide, tandis que "Routines de manipulation de chaînes" pourrait être le nom d'une rubrique).

*ISpecialWinHelpViewe*r prend en charge la réponse aux messages WinHelp spécialisés qu'une application s'exécutant sous Windows risque de recevoir et qui ne sont pas facilement généralisables. En principe, seules les applications opérant dans l'environnement Windows ont besoin d'implémenter cette interface et, même alors, elle n'est nécessaire qu'aux applications faisant une forte utilisation des messages WinHelp non standard.

IHelpManager fournit le mécanisme permettant au visualiseur d'aide de communiquer avec le gestionnaire d'aide de l'application et demande un supplément d'information. Un *IHelpManager* est obtenu au moment où le visualiseur d'aide se recense lui-même.

IHelpSystem fournit le mécanisme permettant à *TApplication* de transmettre les requêtes d'aide au système d'aide. *TApplication* obtient une instance d'un objet qui implémente à la fois *IHelpSystem* et *IHelpManager* au chargement de l'application et exporte cette instance en tant que propriété ; cela autorise d'autres parties du code de l'application à traiter les requêtes d'aide directement lorsque c'est possible.

IHelpSelector fournit le mécanisme permettant au système d'aide d'invoquer l'interface utilisateur pour demander quel visualiseur d'aide doit être utilisé lorsque plusieurs visualiseurs sont à même de gérer une requête d'aide et d'afficher un sommaire. Cette capacité d'affichage n'est pas directement intégrée au gestionnaire d'aide pour permettre l'utilisation du même code du gestionnaire d'aide, quel que soit l'ensemble de widgets ou la bibliothèque de classes utilisé.

Implémentation de ICustomHelpViewer

L'interface *ICustomHelpViewer* contient trois types de méthodes : les méthodes servant à communiquer au gestionnaire d'aide les informations du niveau système (par exemple, des informations non liées à une requête d'aide particulière) ; les méthodes servant à afficher l'aide en fonction du mot clé fourni par le gestionnaire d'aide ; les méthodes servant à afficher le sommaire.

Communication avec le gestionnaire d'aide

ICustomHelpViewer fournit quatre fonctions servant à communiquer les informations système au gestionnaire d'aide :

- *GetViewerName*
- *NotifyID*
- *ShutDown*
- *SoftShutDown*

Le gestionnaire d'aide appelle ces fonctions dans les circonstances suivantes :

- *ICustomHelpViewer.GetViewerName : String* est appelée lorsque le gestionnaire d'aide veut connaître le nom du visualiseur (par exemple, si l'application doit afficher la liste des visualiseurs recensés). Cette information est renvoyée via une chaîne et celle-ci doit être logiquement statique (c'est-à-dire qu'elle ne peut pas être modifiée pendant que l'application s'exécute). Les jeux de caractères multi-octets ne sont pas pris en charge.
- *ICustomHelpViewer.NotifyID(const ViewerID: Integer)* est appelée **immédiatement** après le recensement pour fournir au visualiseur un cookie qui l'identifie de manière unique. Cette information doit être conservée pour une utilisation ultérieure ; si le visualiseur s'interrompt de son propre chef (et non pas en réponse à une notification du gestionnaire d'aide), il doit fournir le cookie identificateur au gestionnaire d'aide pour que celui-ci libère toutes les références au visualiseur. (Ne pas réussir à fournir le cookie, ou en fournir un mauvais, peut amener le gestionnaire d'aide à libérer les références au mauvais visualiseur.)
- *ICustomHelpViewer.ShutDown* est appelée par le gestionnaire d'aide pour signaler au visualiseur d'aide que le gestionnaire va s'interrompre et que toutes les ressources allouées par le visualiseur doivent être libérées. Il est conseillé de déléguer à cette méthode la libération de toutes les ressources.
- *ICustomHelpViewer.SoftShutDown* est appelée par le gestionnaire d'aide pour demander au visualiseur d'aide de fermer toutes les manifestations externes visibles du système d'aide (par exemple, les fenêtres affichant des informations d'aide) sans décharger le visualiseur.

Demande d'informations au gestionnaire d'aide

Les visualiseurs d'aide communiquent avec le gestionnaire d'aide via l'interface *IHelpManager*, une instance de celle-ci leur est renvoyée lorsqu'ils se recensent auprès du gestionnaire d'aide. *IHelpManager* permet au visualiseur de communiquer quatre choses : une requête pour le handle de fenêtre du contrôle actif ; une requête pour le nom du fichier d'aide supposé contenir l'aide sur le contrôle actif ; une requête pour le chemin d'accès à ce fichier ; la notification que le visualiseur d'aide va s'interrompre lui-même en réponse à autre chose qu'une demande issue du gestionnaire d'aide.

IHelpManager.GetHandle : LongInt est appelée par le visualiseur d'aide s'il veut connaître le handle du contrôle actif ; le résultat est un handle de fenêtre.

IHelpManager.GetHelpFile: String est appelée par le visualiseur d'aide s'il veut connaître le nom du fichier d'aide supposé contenir l'aide sur le contrôle actif.

IHelpManager.Release est appelée pour signaler au gestionnaire d'aide qu'un visualiseur d'aide va se déconnecter. Elle ne doit *jamais* être appelée en réponse à une requête via *ICustomHelpViewer.ShutDown* ; elle sert à annoncer au gestionnaire d'aide uniquement les déconnexions inattendues.

Affichage de l'aide sur un mot clé

Les requêtes d'aide adressées au visualiseur d'aide sont soit basées sur un *mot clé*, auquel cas le visualiseur est chargé de fournir de l'aide en fonction d'une chaîne particulière, soit basées sur un *contexte*, auquel cas le visualiseur est chargé de fournir de l'aide en fonction d'un identificateur numérique particulier. (Les contextes d'aide numériques sont la forme par défaut des requêtes d'aide des applications s'exécutant sous Windows et utilisant le système WinHelp ; bien que CLX les supporte, il n'est pas conseillé de les utiliser dans les applications CLX car la majorité des systèmes d'aide sous Linux ne les comprennent pas.) Les implémentations de *ICustomHelpViewer* sont nécessaires pour prendre en charge les requêtes basées sur les mots clés, les implémentations de *IExtendedHelpViewer* sont nécessaires pour prendre en charge les requêtes basées sur des contextes.

ICustomHelpViewer fournit trois méthodes pour traiter l'aide par mot clé :

- *UnderstandsKeyword*
- *GetHelpStrings*
- *ShowHelp*

```
ICustomHelpViewer.UnderstandsKeyword(const HelpString: String): Integer
```

est la première des trois méthodes appelées par le gestionnaire d'aide, qui appellera *chacun* des visualiseurs d'aide recensés avec la même chaîne pour demander si le visualiseur peut fournir de l'aide pour cette chaîne ; le visualiseur est supposé répondre par un entier indiquant le nombre de pages d'aide différentes qu'il peut afficher en réponse à cette demande. Le visualiseur peut utiliser la méthode qu'il veut pour le déterminer — dans l'EDI, le visualiseur HyperHelp maintient son propre index et effectue la recherche. Si le visualiseur ne dispose pas d'aide sur le mot clé, il doit renvoyer zéro. Les nombres négatifs sont interprétés comme zéro, mais ce comportement n'est pas garanti dans les versions futures.

```
ICustomHelpViewer.GetHelpStrings(const HelpString: String): TStringList
```

est appelée par le gestionnaire d'aide si plusieurs visualiseurs peuvent fournir de l'aide sur une rubrique. Le visualiseur doit renvoyer une *TStringList*. Les chaînes de la liste renvoyée doivent correspondre aux pages disponibles pour le mot clé, mais les caractéristiques de correspondance peuvent être déterminées par le visualiseur. Dans le cas du visualiseur HyperHelp, la liste de chaînes contient toujours exactement une entrée (HyperHelp a sa propre indexation, dupliquer cela ailleurs serait inutile) ; dans le cas du visualiseur de pages Man, la liste de

chaînes est constituée de plusieurs chaînes, une par section du manuel contenant une page correspondant au mot clé.

```
ICustomHelpViewer.ShowHelp(const HelpString: String)
```

est appelée par le gestionnaire d'aide s'il a besoin que le visualiseur d'aide affiche de l'aide sur un mot clé particulier. C'est le dernier appel de méthode de l'opération ; elle n'est jamais appelée sauf si *CanShowKeyword* a été invoquée au préalable.

Affichage des sommaires

ICustomHelpViewer fournit deux méthodes pour afficher les sommaires :

- *CanShowTableOfContents*
- *ShowTableOfContents*

Leur mode opératoire ressemble beaucoup à celui des requêtes d'aide par mot clé : le gestionnaire d'aide interroge d'abord tous les visualiseurs d'aide en appelant *ICustomHelpViewer.CanShowTableOfContents : Boolean*, puis invoque un visualiseur d'aide particulier en appelant *ICustomHelpViewer.ShowTableOfContents*.

Il est raisonnable pour un visualiseur de refuser de prendre en charge les demandes de sommaires. C'est ce que fait, par exemple, le visualiseur de pages Man car le concept de sommaire est trop éloigné de la façon dont fonctionnent les pages Man ; le visualiseur HyperHelp, en revanche, supporte les sommaires en passant la requête d'affichage du sommaire directement à HyperHelp. Il n'est *pas* raisonnable, cependant, pour une implémentation de *ICustomHelpViewer* de répondre aux requêtes *CanShowTableOfContents* avec une valeur *true* et d'ignorer ensuite les requêtes *ShowTableOfContents*.

Implémentation de IExtendedHelpViewer

ICustomHelpViewer est seule à fournir un support direct de l'aide par mot clé. Certains systèmes d'aide (spécialement WinHelp) opèrent en associant un nombre (appelé *ID de contexte*) aux mots clés, de manière interne au système d'aide et donc de manière invisible pour l'application. De tels systèmes nécessitent que l'application supporte l'aide par contexte, où l'application invoque le système d'aide avec un nombre plutôt qu'une chaîne, et que le système d'aide effectue la traduction du nombre.

Les applications écrites en CLX peuvent communiquer avec les systèmes utilisant l'aide par contexte, en étendant l'objet qui implémente *ICustomHelpViewer* afin qu'il implémente également *IExtendedHelpViewer*. *IExtendedHelpViewer* prend aussi en charge la communication avec les systèmes d'aide vous permettant d'aller directement aux rubriques de haut niveau au lieu d'utiliser les recherches par mot clé.

IExtendedHelpViewer expose quatre fonctions. Deux d'entre elles, *UnderstandsContext* et *DisplayHelpByContext*, sont utilisées pour supporter l'aide

par contexte ; les deux autres, *UnderstandsTopic* et *DisplayTopic*, sont utilisées pour supporter les rubriques.

Lorsque l'utilisateur d'une application appuie sur F1, le gestionnaire d'aide appelle

```
IExtendedHelpViewer.UnderstandsContext(const ContextID: Integer;
const HelpFileName: String): Boolean
```

et le contrôle actif prend en charge l'aide par contexte et non l'aide par mot clé. De même qu'avec *ICustomHelpViewer.CanShowKeyword*, le gestionnaire d'aide interroge tous les visualiseurs d'aide recensés l'un après l'autre. Mais, au contraire de *ICustomHelpViewer.CanShowKeyword*, si plusieurs visualiseurs supportent le contexte spécifié, c'est le *premier* visualiseur recensé et supportant le contexte qui est invoqué.

Le gestionnaire d'aide appelle

```
IExtendedHelpViewer.DisplayHelpByContext(const ContextID: Integer;
const HelpFileName: String)
```

après avoir consulté les visualiseurs d'aide recensés.

Les fonctions de support des rubriques se comportent de la même façon :

```
IExtendedHelpViewer.UnderstandsTopic(const Topic: String): Boolean
```

est utilisée pour demander aux visualiseurs d'aide s'ils supportent une rubrique ;

```
IExtendedHelpViewer.DisplayTopic(const Topic: String)
```

est utilisée pour invoquer le premier visualiseur recensé indiquant qu'il peut fournir de l'aide sur cette rubrique.

Implémentation de IHelpSelector

IHelpSelector est un compagnon de *ICustomHelpViewer*. Lorsque plusieurs visualiseurs recensés peuvent assurer le support du mot clé, du contexte ou de la rubrique spécifié, ou peuvent fournir un sommaire, le gestionnaire d'aide doit faire un choix entre eux. Dans le cas des contextes ou des rubriques, le gestionnaire d'aide sélectionne *toujours* le premier visualiseur d'aide prétendant assurer le support. Dans le cas des mots clés ou des sommaires, le gestionnaire d'aide, par défaut, sélectionne le premier visualiseur d'aide. Ce comportement peut être redéfini par une application.

Pour supplanter la décision du gestionnaire d'aide, une application doit recenser une classe fournissant une implémentation de l'interface *IHelpSelector*. *IHelpSelector* exporte deux fonctions : *SelectKeyword* et *TableOfContents*. Les deux acceptent comme argument un *TStrings* contenant, l'un à la suite de l'autre, soit les correspondances possibles des mots clés, soit les noms des visualiseurs pouvant fournir un sommaire. L'implémenteur est nécessaire pour renvoyer l'indice (dans le *TStrings*) représentant la chaîne sélectionnée.

Remarque Le gestionnaire d'aide risque de se tromper si les chaînes sont re-arrangées ; il est conseillé que les implémenteurs de *IHelpSelector* ne le fassent pas. Le système

d'aide ne supporte qu'un *seul* HelpSelector ; lorsque de nouveaux sélectionneurs sont recensés, tout sélectionneur existant préalablement est déconnecté.

Recensement des objets du système d'aide

Pour que le gestionnaire d'aide communique avec eux, les objets implémentant *ICustomHelpViewer, IExtendedHelpViewer, ISpecialWinHelpViewer* et *IHelpSelector* doivent se recenser auprès du gestionnaire d'aide.

Pour recenser les objets du système d'aide auprès du gestionnaire d'aide, il vous faut :

- Recenser le visualiseur d'aide
- Recenser le sélectionneur d'aide

Recensement des visualiseurs d'aide

L'unité contenant l'implémentation de l'objet doit utiliser HelpIntfs. Une instance de l'objet doit être déclarée dans la section **var** de l'unité d'implémentation.

La section initialisation de l'unité d'implémentation doit assigner la variable d'instance et la transmettre à la fonction *RegisterViewer*. *RegisterViewer* est une fonction simple, exportée par HelpIntfs.pas, qui prend un *ICustomHelpViewer* en argument et renvoie un *IHelpManager*. Le *IHelpManager* doit être enregistré pour une utilisation ultérieure.

Recensement des sélectionneurs d'aide

L'unité contenant l'implémentation de l'objet doit utiliser HelpIntfs et QForms. Une instance de l'objet doit être déclarée dans la section **var** de l'unité d'implémentation.

La section initialisation de l'unité d'implémentation doit recenser le sélectionneur d'aide via la propriété *HelpSystem* de l'objet global Application :

```
Application.HelpSystem.AssignHelpSelector(myHelpSelectorInstance)
```

Cette procédure ne renvoie pas de valeur.

Utilisation de l'aide dans une application VCL

Les sections suivantes expliquent comment utiliser l'aide dans une application VCL.

- Comment TApplication traite-il l'aide VCL ?
- Comment les contrôles traitent-ils l'aide ?
- Appel direct à un système d'aide
- Utilisation de IHelpSystem

Comment TApplication traite-il l'aide VCL ?

TApplication dans la VCL fournit quatre méthodes accessibles depuis le code de l'application :

Tableau 5.5 Méthodes d'aide de TApplication

HelpCommand	Transmet HELP_COMMAND, de style aide Windows, à WinHelp. Les demandes d'aide transmises par le biais de ce mécanisme sont passées uniquement aux implémentations de IspecialWinHelpViewer.
HelpContext	Invoque le système d'aide en utilisant une requête par contexte.
HelpKeyword	Invoque le système d'aide en utilisant une requête par mot clé.
HelpJump	Demande l'affichage d'une rubrique particulière.

Les quatre fonctions prennent les données qui leurs sont transmises et les font suivre via une donnée membre de *TApplication* qui représente le système d'aide. Cette donnée membre est directement accessible via la propriété *HelpSystem*.

Comment les contrôles traitent-ils l'aide ?

Tous les contrôles dérivant de *TControl* exposent trois propriétés qui sont utilisées par le système d'aide : *HelpSystem*, *HelpType*, *HelpContext* et *HelpKeyword*.

La propriété *HelpType* contient une instance d'un type énuméré qui détermine si le concepteur du contrôle a prévu de fournir l'aide par mot clé ou par contexte. Si *HelpType* est définie par *htKeyword*, le système d'aide s'attend à ce que le contrôle utilise l'aide par mot clé et il examine uniquement le contenu de la propriété *HelpKeyword*. En revanche, si *HelpType* est définie par *htContext*, le système d'aide s'attend à ce que le contrôle utilise l'aide par contexte et il examine uniquement le contenu de la propriété *HelpContext*.

En plus des propriétés, les contrôles exposent une seule méthode, *InvokeHelp*, qui peut être appelée pour transmettre une requête au système d'aide. Elle ne prend pas de paramètre et appelle dans l'objet global Application les méthodes qui correspondent au type d'aide supporté par le contrôle.

Les messages d'aide sont automatiquement invoqués lors de l'appui sur la touche F1 car la méthode *KeyDown* de *TWinControl* appelle *InvokeHelp*.

Utilisation de l'aide dans une application CLX

Les sections suivantes expliquent comment utiliser l'aide dans une application CLX.

- Comment TApplication traite-il l'aide CLX ?
- Comment les contrôles CLX traitent-ils l'aide ?
- Appel direct à un système d'aide
- Utilisation de IHelpSystem

Comment TApplication traite-il l'aide CLX ?

TApplication dans CLX fournit deux méthodes accessibles depuis le code de l'application :

- *ContextHelp*, qui invoque le système d'aide en utilisant une requête par contexte
- *KeywordHelp*, qui invoque le système d'aide en utilisant une requête par mot clé

Les deux fonctions prennent en argument le contexte ou le mot clé, et fait suivre la requête à une donnée membre de *TApplication*, qui représente le système d'aide. Cette donnée membre est directement accessible via la propriété en lecture seule *HelpSystem*.

Comment les contrôles CLX traitent-ils l'aide ?

Tous les contrôles dérivant de *TControl* exposent quatre propriétés qui sont utilisées par le système d'aide : *HelpType*, *HelpFile*, *HelpContext* et *HelpKeyword*. *HelpFile* est censée contenir le nom du fichier où se trouve l'aide du contrôle ; si l'aide se trouve dans un système d'aide externe ne reconnaissant pas les noms de fichiers (par exemple, le système des pages Man), la propriété doit être laissée vierge.

La propriété *HelpType* contient une instance d'un type énuméré qui détermine si le concepteur du contrôle a prévu de fournir l'aide par mot clé ou par contexte ; les deux autres propriétés lui sont liées. Si *HelpType* est définie par *htKeyword*, le système d'aide s'attend à ce que le contrôle utilise l'aide par mot clé et il examine uniquement le contenu de la propriété *HelpKeyword*. En revanche, si *HelpType* est définie par *htContext*, le système d'aide s'attend à ce que le contrôle utilise l'aide par contexte et il examine uniquement le contenu de la propriété *HelpContext*.

En plus des propriétés, les contrôles exposent une seule méthode, *InvokeHelp*, qui peut être appelée pour transmettre une requête au système d'aide. Elle ne prend pas de paramètre et appelle dans l'objet global Application les méthodes qui correspondent au type d'aide supporté par le contrôle.

Les messages d'aide sont automatiquement invoqués lors de l'appui sur la touche F1 car la méthode *KeyDown* de *TWidgetControl* appelle *InvokeHelp*.

Appel direct à un système d'aide

Pour les fonctionnalités des systèmes d'aide non fournis par la VCL ou par CLX, *TApplication* fournit une propriété, accessible en lecture seulement, qui donne un accès direct au système d'aide. Cette propriété est une instance d'une implémentation de l'interface *IHelpSystem*. *IHelpSystem* et *IHelpManager* sont implémentées par le même objet, mais une interface est utilisée pour permettre à

l'application de communiquer avec le gestionnaire d'aide, et l'autre est utilisée pour permettre aux visualiseurs d'aide de communiquer avec le gestionnaire d'aide.

Utilisation de IHelpSystem

IHelpSystem permet à l'application VCL ou CLX de réaliser trois choses :

- Fournir au gestionnaire d'aide les informations de chemin d'accès
- Fournir un nouveau sélectionneur d'aide
- Demander au gestionnaire d'aide d'afficher l'aide

Assigner un sélectionneur d'aide permet au gestionnaire d'aide de déléguer la prise de décision au cas où plusieurs systèmes d'aide externes peuvent apporter l'aide pour le même mot clé. Pour plus d'informations, voir la section "Implémentation de IHelpSelector" à la page 30.

IHelpSystem exporte quatre procédure et une fonction utilisées pour demander au gestionnaire d'aide d'afficher l'aide :

- *ShowHelp*
- *ShowContextHelp*
- *ShowTopicHelp*
- *ShowTableOfContents*
- *Hook*

Hook est entièrement destinée à la compatibilité WinHelp et ne doit pas être utilisée dans une application CLX ; elle permet le traitement des messages WM_HELP qui ne peuvent pas être directement traduits en requêtes d'aide basée sur un mot clé, un contexte ou une rubrique. Les autres méthodes prennent chacune deux arguments : le mot clé, l'ID de contexte ou la rubrique pour lequel l'aide est demandée, et le fichier d'aide dans lequel on s'attend à la trouver.

En général, sauf si vous demandez une aide par rubrique, il est aussi efficace et plus clair de transmettre les requêtes au gestionnaire d'aide via la méthode *InvokeHelp* de votre contrôle.

Personnalisation du système d'aide de l'EDI

L'EDI de Delphi supporte les visualiseurs d'aide multiples exactement comme le fait une application VCL ou CLX : il délègue les demandes d'aide au gestionnaire d'aide, qui les fait suivre aux visualiseurs d'aide recensés. L'EDI utilise le même visualiseur d'aide WinHelpViewer que VCL.

Pour installer un nouveau visualiseur d'aide dans l'EDI, faites exactement ce que vous feriez dans une application CLX à une différence près. Vous écrivez un objet qui implémente *ICustomHelpViewer* (et, éventuellement, *IExtendedHelpViewer*) pour faire suivre les requêtes d'aide au visualiseur externe de votre choix, et vous recensez le *ICustomHelpViewer* avec l'EDI.

Pour recenser un visualiseur d'aide personnalisé avec l'EDI,

1. Assurez-vous que l'unité implémentant le visualiseur d'aide contient HelpIntfs.pas.
2. Construisez l'unité dans un paquet de conception recensé avec l'EDI, et construisez le paquet en activant l'option Paquets d'exécution. (C'est nécessaire pour garantir que l'instance du gestionnaire d'aide utilisée par l'unité est la même que celle utilisée par l'EDI.)
3. Assurez-vous que le visualiseur d'aide existe comme instance globale à l'intérieur de l'unité.
4. Dans la section initialisation de l'unité, assurez-vous que l'instance est transmise à la fonction *RegisterHelpViewer*.

Chapitre

6

Conception de l'interface utilisateur des applications

Delphi vous permet de créer une interface utilisateur en sélectionnant des composants de la palette de composants et en les déposant dans des fiches. Vous faites faire aux composants ce que vous voulez en définissant leurs propriétés et en écrivant le code de leurs gestionnaires d'événements.

Contrôle du comportement de l'application

TApplication, *TScreen* et *TForm* sont les classes qui constituent la base de toutes les applications Delphi en contrôlant le comportement de votre projet. La classe *TApplication* sert de fondation à une application en fournissant les propriétés et les méthodes qui encapsulent le comportement d'un programme standard. La classe *TScreen* est utilisée à l'exécution pour gérer les fiches et les modules de données chargés, ainsi que pour maintenir des informations spécifiques au système comme la résolution écran ou les fontes utilisables à l'affichage. Des instances de la classe *TForm* servent à construire l'interface utilisateur de votre application. Les fenêtres et les boîtes de dialogue d'une application sont basées sur *TForm*.

Utilisation de la fiche principale

TForm est la classe essentielle dans la création d'applications disposant d'une interface utilisateur graphique. Lorsque vous ouvrez Delphi, ce qui affiche un projet par défaut, ou lorsque vous créez un nouveau projet, une fiche est affichée pour vous permettre de démarrer la conception de votre interface utilisateur.

La première fiche créée et enregistrée dans un projet devient, par défaut, la fiche principale du projet : c'est la première fiche créée à l'exécution. Quand vous ajoutez d'autres fiches dans vos projets, vous pouvez en choisir une autre pour servir de fiche principale à votre application. Par ailleurs, faire d'une fiche la fiche principale est le moyen le plus simple de la tester à l'exécution : à moins de changer explicitement l'ordre de création, la fiche principale est la première fiche affichée lors de l'exécution d'une application.

Pour changer la fiche principale d'un projet :

1 Choisissez Projet | Options et sélectionnez la page Fiche.

2 Dans la boîte liste Fiche principale, sélectionnez la fiche à utiliser comme fiche principale du projet et choisissez OK.

Désormais, si vous exécutez votre application, la fiche choisie comme fiche principale s'affiche.

Ajout de fiches

Pour ajouter une fiche à votre projet, sélectionnez Fichier | Nouvelle fiche. Toutes les fiches d'un projet ainsi que les unités correspondantes sont affichées dans le gestionnaire de projet (Voir | Gestionnaire de projet) et vous pouvez afficher la liste des fiches en choisissant Voir | Fiches.

Liaison de fiches

L'ajout d'une fiche au projet ajoute au fichier projet une référence à cette fiche mais pas aux autres unités du projet. Avant d'écrire du code faisant référence à la nouvelle fiche, vous devez ajouter une référence à cette fiche dans les fichiers unité des fiches y faisant référence. Cela s'appelle la *liaison de fiche*.

La liaison de fiche est fréquemment utilisée pour donner accès aux composants contenus dans une autre fiche. Par exemple, la liaison de fiche est souvent employée pour permettre à une fiche contenant des composants orientés données de se connecter aux composants d'accès aux données d'un module de données.

Pour lier une fiche à une autre fiche :

1 Sélectionnez la fiche qui fait référence à une autre.
2 Choisissez Fichier | Utiliser l'unité.
3 Sélectionnez le nom de l'unité de la fiche qui doit être référencée.
4 Choisissez OK.

Lier une fiche à une autre signifie simplement que les clauses **uses de** l'unité d'une fiche contiennent une référence àl'unité de l'autre fiche. Ainsi la fiche liée et ses composants rentrent dans la portée de la fiche.

Références circulaires d'unités

Quand deux fiches doivent se référencer mutuellement, il est possible de générer une erreur "Référence circulaire" lors de la compilation du programme. Pour éviter une telle erreur, utilisez l'une des méthodes suivantes :

- Placez les deux clauses **uses**, avec les identificateurs d'unités, dans la section **implementation** de leur fichier unité respectif. (C'est ce que fait la commande Fichier | Utiliser l'unité.)
- Placez l'une des clauses **uses** dans la section **interface** et l'autre dans la section **implementation**. (il est rarement nécessaire de placer l'identificateur de l'unité d'une autre fiche dans la section **interface**).

Ne placez pas les deux clauses **uses** dans la section **interface** de leur fichier unité respectif. Cela provoque l'erreur "Référence circulaire" à la compilation.

Cacher la fiche principale

Vous pouvez empêcher l'affichage de la fiche principale lors du démarrage de l'application. Pour ce faire, vous devez utiliser la variable globale *Application* (décrite dans la rubrique suivante).

Pour masquer la fiche principale au démarrage :

1. Choisissez Projet | Voir le source pour afficher le fichier du projet principal.
2. Ajoutez les lignes suivantes après l'appel de `Application.CreateForm` et avant l'appel de `Application.Run`.

```
Application.ShowMainForm := False;
Form1.Visible := False; { le nom de votre fiche principale peut être différent }
```

Remarque Vous pouvez initialiser la valeur de la propriété *Visible* de la fiche à *False* en utilisant à la conception l'inspecteur d'objets au lieu de la définir à l'exécution comme indiqué ci-dessus.

Manipulation de l'application

La variable globale *Application* de type *TApplication* se trouve dans chaque application utilisant la VCL ou CLX. *Application* encapsule l'application et propose de nombreux services fonctionnant en arrière-plan du programme. Ainsi, *Application* gère la manière d'appeler un fichier d'aide depuis les menus de votre programme. La compréhension du fonctionnement de *TApplication* est plus importante pour le concepteur de composants que pour le développeur d'applications autonomes, mais vous devez définir les options gérées par *Application* dans la page Application de la boîte de dialogue Options de projet (Projet | Options) quand vous créez un projet.

De plus, *Application* reçoit de nombreux événements qui s'appliquent à l'application dans son ensemble. Par exemple, l'événement *OnActivate* vous permet de réaliser des actions au démarrage de l'application, l'événement *OnIdle* vous permet d'exécuter des traitements en arrière-plan lorsque l'application n'est

pas occupée, l'événement *OnMessage* vous permet d'intercepter les messages Windows (sous Windows uniquement), l'événement *OnEvent* vous permet d'intercepter des événements, etc. Bien que vous ne puissiez pas utiliser l'EDI pour examiner les propriétés et les événements de la variable globale *Application*, un autre composant, *TApplicationEvents*, intercepte les événements et vous permet de fournir les gestionnaires d'événements à l'aide de l'EDI.

Gestion de l'écran

Une variable globale de type *TScreen*, appelée *Screen*, est créée lors de la création d'un projet. Screen encapsule l'état de l'écran dans lequel l'application s'exécute. Parmi les fonctions imparties à *Screen*, il y a

- la gestion de l'aspect du curseur,
- la taille de la fenêtre dans laquelle s'exécute l'application,
- la liste des fontes disponibles pour le périphérique écran.
- le comportement si plusieurs écrans (non disponible en programmation multiplate-forme)

Si votre application Windows s'exécute sur plusieurs moniteurs, *Screen* gère une liste des moniteurs et leurs dimensions afin que vous puissiez effectivement gérer la disposition de l'interface utilisateur.

Lors de l'utilisation de CLX pour la programmation multiplate-forme, le comportement par défaut est le suivant : les applications créent un composant écran en fonction des informations concernant le périphérique d'écran en cours et l'assigne à Screen.

Gestion de la disposition

A son niveau le plus élémentaire, vous contrôlez l'organisation de votre interface utilisateur par la manière de disposer les contrôles dans les fiches. Le choix des emplacements est reflété par les propriétés *Top*, *Left*, *Width* et *Height* des contrôles. Vous pouvez modifier ces valeurs à l'exécution afin de modifier la position ou la taille des contrôles dans les fiches.

Les contrôles disposent de nombreuses autres propriétés qui leur permettent de s'adapter automatiquement à leur contenu ou à leur conteneur. Cela vous permet d'organiser les fiches de telle manière que les différents éléments forment un tout unifié.

Deux propriétés contrôlent la position et la taille d'un contrôle relativement à celle de son parent. La propriété *Align* vous permet d'obliger un contrôle à s'adapter exactement à un côté spécifié de son parent ou à occuper toute la place disponible de la zone client du parent une fois les autres contrôles alignés. Quand le parent est redimensionné, les contrôles alignés sont automatiquement redimensionnés et restent positionnés le long d'un côté donné du parent.

Si vous voulez qu'un contrôle reste positionné relativement à un côté particulier de son parent sans toucher ce bord ou être redimensionné pour occuper la totalité du côté, vous pouvez utiliser la propriété *Anchors*.

Pour vous assurer qu'un contrôle ne devient ni trop grand ni trop petit, vous pouvez utiliser la propriété *Constraints*. *Constraints* vous permet de spécifier la hauteur maximum, la hauteur minimum, la largeur maximum et la largeur minimum du contrôle. Initialisez ces valeurs afin de limiter la taille (en pixels) de la hauteur et de la largeur du contrôle. Ainsi, en initialisant les contraintes *MinWidth* et *MinHeight* d'un objet conteneur, vous êtes certain que ses objets enfant sont toujours visibles.

La valeur de *Constraints* se propage le long de la hiérarchie parent/enfant de telle manière que la taille d'un objet peut être restreinte car il contient des enfants alignés qui ont des contraintes de taille. *Constraints* peut également empêcher un contrôle d'être mis à l'échelle dans une dimension particulière lors de l'appel de sa méthode *ChangeScale*.

TControl introduit un événement protégé, *OnConstrainedResize*, de type *TConstrainedResizeEvent* :

```
TConstrainedResizeEvent = procedure(Sender: TObject; var MinWidth, MinHeight, MaxWidth,
MaxHeight: Integer) of object;
```

Cet événement vous permet de surcharger les contraintes de taille lors d'une tentative de redimensionnement du contrôle. Les valeurs des contraintes sont transmises comme paramètres `var`, elles peuvent donc être modifiées dans le gestionnaire d'événement. *OnConstrainedResize* est publié pour les objets conteneur (*TForm*, *TScrollBox*, *TControlBar* et *TPanel*). De plus, les concepteurs de composants peuvent utiliser ou publier cet événement dans tout descendant de *TControl*.

Les contrôles dont le contenu peut changer de taille ont une propriété *AutoSize* qui force le contrôle à adapter sa taille à l'objet ou à la fonte qu'il contient.

Réponse aux notifications d'événement

Le système d'exploitation envoie une notification à votre application lorsqu'un événement se produit (un clic de souris, la frappe de touches, etc.) pendant son exécution. La façon dont les notifications d'événement sont gérées par les objets VCL et CLX est différente, mais la façon dont vous devez traiter ces notifications au niveau du composant est la même. Les composants possèdent des événements et des méthodes intégrés pour répondre aux événements les plus fréquents. Vous pouvez utiliser les méthodes fournies avec les composants dans la majorité des cas. Si vous avez besoin de gérer des événements supplémentaires, vous pouvez remplacer une méthode existante et écrire la vôtre. Sauf si vous créez vos propres composants, vous n'aurez pas à modifier le schéma de notification des événements sous-jacent.

VCL Si vous développez des applications uniquement pour Windows, il vous faut comprendre que Windows est un système d'exploitation basé sur les messages. Les messages système sont gérés par un gestionnaire de message qui traduit le message en un événement ou un gestionnaire d'événement. Le message lui-même est un enregistrement transmis par Windows à un contrôle. Par exemple, quand vous cliquez sur un bouton de la souris dans une boîte de dialogue,

Windows envoie au contrôle actif un message et l'application contenant ce contrôle réagit à ce nouvel événement. Si vous avez cliqué sur un bouton, l'événement *OnClick* peut être activé à la réception du message. Si vous avez juste cliqué dans la fiche, l'application peut ne pas tenir compte du message.

Le type enregistrement transmis par Windows à l'application est appelé un *TMsg*. Windows prédéfinit une constante pour chaque message, ces valeurs sont stockées dans le champ de message de l'enregistrement *TMsg*. Chacune de ces constantes commence par les lettres "wm".

La VCL gère automatiquement les messages sauf si vous surchargez le système de gestion des messages et créez vos propres gestionnaires d'événements. Pour davantage d'informations sur les messages et la gestion des messages, voir "Compréhension du système de gestion des messages" à la page 46-1, "Modification de la gestion des messages" à la page 46-3 et "Création de nouveaux gestionnaires de messages" à la page 46-5.

CLX Pour la programmation multiplate-forme : La notification du système d'exploitation qu'il s'est produit un événement est envoyée à la couche widget Qt sous-jacente où il est traduit en un événement et éventuellement en des objets événement par *HookEvents*. *EventFilter* est appelée automatiquement lorsqu'un événement Qt lié à la souris ou au clavier doit être géré par un contrôle CLX.

EventFilter réagit aux notifications d'événement en apportant la réponse par défaut. Généralement, cette réponse implique la transmission de l'événement à la méthode virtuelle appropriée (telle que la méthode *Click*, qui génère un événement *OnClick*).

Remarque CLX Lors de la redéfinition de la méthode *EventFilter*, vous devez appeler la méthode héritée afin que le traitement d'événement par défaut puisse intervenir.

Utilisation des fiches

Quand vous créez une fiche Delphi dans l'EDI, Delphi crée automatiquement la fiche en mémoire en ajoutant du code au point d'entrée principal de votre application. C'est généralement le comportement souhaité et vous n'avez donc rien à y changer. Ainsi, la fiche principale existe pour toute la durée du programme, il est donc peu probable que vous changiez le comportement par défaut de Delphi quand vous créez la fiche de votre fenêtre principale.

Néanmoins, il n'est pas toujours nécessaire de conserver en mémoire toutes les fiches de votre application pour toute la durée de l'exécution du programme. Si vous ne souhaitez pas avoir tout le temps en mémoire toutes les boîtes de dialogue de votre application, vous pouvez les créer dynamiquement quand vous voulez les voir apparaître.

Une fiche peut être modale ou non modale. Les fiches modales sont des fiches avec lesquelles l'utilisateur doit interagir avant de pouvoir passer à une autre fiche (par exemple une boîte de dialogue impose une saisie de l'utilisateur). Les fiches non modales sont des fenêtres visibles tant qu'elles ne sont pas masquées par une autre fenêtre, fermées ou réduites par l'utilisateur.

Contrôle du stockage en mémoire des fiches

Par défaut, Delphi crée automatiquement en mémoire la fiche principale de l'application en ajoutant le code suivant au point d'entrée principal de l'application :

```
Application.CreateForm(TForm1, Form1);
```

Cette fonction crée une variable globale portant le même nom que la fiche. Ainsi, chaque fiche d'une application a une variable globale associée. Cette variable est un pointeur sur une instance de la classe de la fiche et sert à désigner la fiche durant l'exécution de l'application. Toute unité qui inclut l'unité de la fiche dans sa clause **uses** peut accéder à la fiche par l'intermédiaire de cette variable.

Toutes les fiches créées de cette manière dans l'unité du projet apparaissent quand le programme est exécuté et restent en mémoire durant toute l'exécution de l'application.

Affichage d'une fiche créée automatiquement

Il est possible de créer une fiche au démarrage, mais de ne l'afficher que plus tard dans l'exécution du programme. Les gestionnaires d'événements de la fiche utilisent la méthode *ShowModal* pour afficher une fiche déjà chargée en mémoire :

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  ResultsForm.ShowModal;
end;
```

Dans ce cas, comme la fiche est déjà en mémoire, il n'est pas nécessaire de créer une autre instance ou de détruire cette instance.

Création dynamique de fiche

Toutes les fiches de votre application n'ont pas besoin d'être en mémoire simultanément. Pour réduire la quantité de mémoire nécessaire au chargement de l'application, vous pouvez créer certaines fiches uniquement quand vous en avez besoin. Ainsi, une boîte de dialogue n'a besoin d'être en mémoire que pendant le temps où l'utilisateur interagit avec elle.

Pour spécifier dans l'EDI que la fiche doit être créée à un autre moment pendant l'exécution, procédez de la manière suivante :

1. Sélectionnez Fichier | Nouvelle fiche dans le menu principal afin d'afficher la nouvelle fiche.
2. Retirez la fiche de la liste Fiches créées automatiquement dans la page Fiches de Projet | Options.

   Cela supprime l'appel de la fiche. Vous pouvez également retirer manuellement la ligne suivante au point d'entrée principal du programme :

   ```
   Application.CreateForm(TResultsForm, ResultsForm);
   ```

3. Appelez la fiche au moment souhaité en utilisant la méthode *Show* de la fiche si la fiche est non modale ou la méthode *ShowModal* si la fiche est modale.

Un gestionnaire d'événement de la fiche principale doit créer et détruire une instance de la fiche appelée *ResultsForm* dans l'exemple précédent. Une manière d'appeler cette fiche consiste à utiliser la variable globale comme dans le code suivant. Remarquez que *ResultsForm* étant une fiche modale, le gestionnaire utilise la méthode *ShowModal* :

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
ResultsForm:=TResultForm.Create(self);
try
  ResultsForm.ShowModal;
finally
  ResultsForm.Free;
end;
```

Dans l'exemple précédent, notez l'utilisation de **try..finally**. Placer la ligne `ResultsForm.Free;` dans la clause **finally** garantit que la mémoire allouée à la fiche est libérée même si la fiche déclenche une exception.

Dans cet exemple, le gestionnaire d'événement supprime la fiche après sa fermeture, la fiche doit donc être recréée si vous avez besoin de *ResultsForm* ailleurs dans l'application. Si la fiche était affichée en utilisant *Show*, vous ne pourriez la supprimer dans le gestionnaire d'événement car, après l'appel de *Show*, l'exécution du code du gestionnaire se poursuit alors que la fiche est toujours ouverte.

Remarque Si vous créez une fiche en utilisant son constructeur, assurez-vous que la fiche n'apparaît pas dans la liste Fiches créées automatiquement de la page Fiches de la boîte de dialogue Options de projet. En effet, si vous créez une nouvelle fiche sans avoir détruit la fiche de même nom dans la liste, Delphi crée la fiche au démarrage et le gestionnaire d'événement crée une nouvelle instance de la fiche, ce qui remplace la référence à l'instance auto-créée. L'instance auto-créée existe toujours mais l'application n'y a plus accès. A la fin du gestionnaire d'événement, la variable globale ne pointe plus sur une fiche valide. Toute tentative d'utiliser la variable globale entraînera probablement le blocage de l'application.

Création de fiches non modales comme fenêtres

Vous devez vous assurer que les variables désignant des fiches non modales existent tant que la fiche est utilisée. Cela signifie que ces variables doivent avoir une portée globale. Le plus souvent, vous utiliserez la variable globale référençant la fiche qui a été créée quand vous avez ajouté la fiche (le nom de variable qui correspond à la valeur de la propriété *Name* de la fiche). Si votre application a besoin d'autres instances de la fiche, déclarez des variables globales distinctes pour chaque instance.

Utilisation d'une variable locale pour créer une instance de fiche

Un moyen fiable de créer une seule instance d'une *fiche modale* consiste à utiliser une variable locale du gestionnaire d'événement comme référence à la nouvelle instance. Si une variable locale est employée, il importe peu que *ResultsForm* soit

ou non auto-créée. Le code du gestionnaire d'événement ne fait pas référence à la variable fiche globale. Par exemple :

```
procedure TMainForm.Button1Click(Sender: TObject);
var
  RF:TResultForm;
begin
  RF:=TResultForm.Create(self)
  RF.ShowModal;
  RF.Free;
end;
```

Remarquez que l'instance globale de la fiche n'est jamais utilisée dans cette version du gestionnaire d'événement.

Habituellement, les applications utilisent les instances globales des fiches. Cependant, si vous avez besoin d'une nouvelle instance d'une fiche modale alors que vous utilisez cette fiche dans une portion réduite de votre application (par exemple dans une seule fonction), une instance locale est normalement le moyen le plus rapide et le plus fiable de manipuler la fiche.

Bien entendu, vous ne pouvez pas utiliser de variables locales pour les fiches non modales dans les gestionnaires d'événements car elles doivent avoir une portée globale pour garantir que la fiche existe aussi longtemps qu'elle est utilisée. *Show* rend la main dès que la fiche est ouverte, donc si vous utilisez une variable locale, la variable locale sort de portée immédiatement.

Transfert de paramètres supplémentaires aux fiches

Généralement, vous créez les fiches de votre application en utilisant l'EDI. Quand elle est créée ainsi, la fiche a un constructeur qui prend un argument, *Owner*, qui est le propriétaire de la fiche créée. (Le propriétaire est l'objet application ou l'objet fiche appelant.) *Owner* peut avoir la valeur **nil**.

Pour transmettre d'autres paramètres à la fiche, créez un constructeur différent et instanciez la fiche en utilisant ce nouveau constructeur. La classe fiche exemple suivante utilise un constructeur supplémentaire proposant le paramètre supplémentaire *whichButton*. Il faut ajouter manuellement ce nouveau constructeur à la fiche.

```
TResultsForm = class(TForm)
  ResultsLabel: TLabel;
  OKButton: TButton;
  procedure OKButtonClick(Sender: TObject);
private
public
  constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
end;
```

Voici le code, créé manuellement, de ce constructeur qui passe le paramètre supplémentaire *whichButton*. Ce constructeur utilise le paramètre *whichButton* pour initialiser la propriété *Caption* d'un contrôle *Label* de la fiche.

```
constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
```

```
begin
  inherited Create(Owner);
  case whichButton of
    1: ResultsLabel.Caption := 'Vous avez choisi le premier bouton.';
    2: ResultsLabel.Caption := 'Vous avez choisi le deuxième bouton.';
    3: ResultsLabel.Caption := 'Vous avez choisi le troisième bouton.';
  end;
end;
```

Quand vous créez une instance d'une fiche disposant de plusieurs constructeurs, vous pouvez sélectionner le constructeur le mieux adapté à vos besoins. Par exemple, le gestionnaire *OnClick* suivant d'un bouton de la fiche crée une instance de *TResultsForm* en utilisant le paramètre supplémentaire :

```
procedure TMainForm.SecondButtonClick(Sender: TObject);
var
  rf: TResultsForm;
begin
  rf := TResultsForm.CreateWithButton(2, self);
  rf.ShowModal;
  rf.Free;
end;
```

Récupération des données des fiches

La plupart des applications réelles utilisent plusieurs fiches. Bien souvent, il est nécessaire de transmettre des informations entre ces différentes fiches. Il est possible de transmettre des informations à une fiche sous la forme des paramètres du constructeur de la fiche destination ou en affectant des valeurs aux propriétés de la fiche. La méthode à utiliser pour obtenir des informations d'une fiche change selon que la fiche est ou non modale.

Récupération de données dans les fiches non modales

Il est facile d'extraire des informations de fiches non modales en appelant des fonctions membre publiques de la fiche ou en interrogeant ses propriétés. Soit, par exemple, une application contenant une fiche non modale appelée *ColorForm* qui contient une boîte liste appelée *ColorListBox* contenant une liste de couleurs ("Rouge", "Vert", "Bleu", etc.). Le nom de couleur sélectionné dans *ColorListBox* est automatiquement stocké dans une propriété appelée *CurrentColor* à chaque fois que l'utilisateur sélectionne une nouvelle couleur. La déclaration de classe pour la fiche est la suivante :

```
TColorForm = class(TForm)
  ColorListBox:TListBox;
  procedure ColorListBoxClick(Sender: TObject);
private
  FColor:String;
public
  property CurColor:String read FColor write FColor;
end;
```

Le gestionnaire d'événement *OnClick* de la boîte liste, *ColorListBoxClick*, initialise la valeur de la propriété *CurrentColor* à chaque fois qu'un nouvel élément est sélectionné. Le gestionnaire d'événement obtient la chaîne dans la boîte liste qui contient le nom de couleur et l'affecte à *CurrentColor*. La propriété *CurrentColor* utilise la fonction d'affectation, *SetColor,* pour stocker la valeur réelle de la propriété dans la donnée membre privée *FColor* :

```
procedure TColorForm.ColorListBoxClick(Sender: TObject);
var
  Index: Integer;
begin
  Index := ColorListBox.ItemIndex;
  if Index >= 0 then
    CurrentColor := ColorListBox.Items[Index]
  else
    CurrentColor := '';
end;
```

Supposons maintenant qu'une autre fiche de l'application, appelée *ResultsForm*, a besoin de connaître la couleur actuellement sélectionnée dans *ColorForm* à chaque fois qu'un bouton (nommé *UpdateButton)* de *ResultsForm* est choisi. Le gestionnaire d'événement *OnClick* de *UpdateButton* doit avoir la forme suivante :

```
procedure TResultForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
begin
  if Assigned(ColorForm) then
  begin
    MainColor := ColorForm.CurrentColor;
    {faire quelque chose avec la chaîne MainColor}
  end;
end;
```

Le gestionnaire d'événement commence par vérifier que *ColorForm* existe en utilisant la fonction *Assigned*. Ensuite, il obtient la valeur de la propriété *CurrentColor* de *ColorForm.*

En procédant autrement, si *ColorForm* a une fonction publique appelée *GetColor*, une autre fiche peut obtenir la couleur en cours sans utiliser la propriété *CurrentColor* (par exemple, `MainColor := ColorForm.GetColor;`). En fait, rien n'empêche l'autre fiche d'obtenir la couleur sélectionnée dans *ColorForm* en examinant directement la valeur sélectionnée dans la boîte liste :

```
with ColorForm.ColorListBox do
  MainColor := Items[ItemIndex];
```

Néanmoins, l'utilisation d'une propriété rend l'interface avec *ColorForm* très claire et simple. Tout ce qu'une fiche a besoin de savoir sur *ColorForm*, c'est comment récupérer la valeur de *CurrentColor*.

Récupération de données dans les fiches modales

Tout comme les fiches non modales, les fiches modales contiennent souvent des informations nécessaires à d'autres fiches. Dans le cas de figure la plus classique, une fiche A lance la fiche modale B. Lors de la fermeture de B, la fiche A a besoin de savoir ce que l'utilisateur a fait dans la fiche B pour décider comment poursuivre les traitements de la fiche A. Si la fiche B est toujours en mémoire, il est possible de l'interroger via ses propriétés et ses fonctions membres tout comme les fiches non modales de l'exemple précédent. Mais comment faire si la fiche B est retirée de la mémoire une fois fermée ? Comme une fiche ne renvoie pas explicitement de valeur, il est nécessaire de préserver les informations importantes de la fiche avant de la détruire.

Pour illustrer cette manière de procéder, considérez une version modifiée de la fiche *ColorForm* conçue comme une fiche modale. Sa classe est déclarée de la manière suivante :

```
TColorForm = class(TForm)
  ColorListBox:TListBox;
  SelectButton: TButton;
  CancelButton: TButton;
  procedure CancelButtonClick(Sender: TObject);
  procedure SelectButtonClick(Sender: TObject);
private
  FColor: Pointer;
public
  constructor CreateWithColor(Value: Pointer; Owner: TComponent);
end;
```

La fiche contient une boîte liste nommée *ColorListBox* contenant une liste de noms de couleur. Quand il est choisi, le bouton nommé *SelectButton* mémorise le nom de la couleur sélectionnée dans *ColorListBox* puis ferme la fiche. *CancelButton* est un bouton qui ferme simplement la fiche.

Remarquez l'ajout à la déclaration de la classe d'un constructeur défini par l'utilisateur qui attend un argument *Pointer*. Normalement, ce paramètre *Pointer* pointe sur une chaîne gérée par la fiche qui déclenche *ColorForm*. Ce constructeur a l'implémentation suivante :

```
constructor TColorForm(Value: Pointer; Owner: TComponent);
begin
  FColor := Value;
  String(FColor^) := '';
end;
```

Le constructeur enregistre le pointeur dans une donnée membre privée *FColor* et initialise la chaîne avec une chaîne vide.

Remarque Pour utiliser ce constructeur défini par l'utilisateur, la fiche doit être créée explicitement. Ce ne peut pas être une fiche auto-créée au démarrage de l'application. Pour davantage d'informations, voir "Contrôle du stockage en mémoire des fiches" à la page 6-7.

Dans l'application, l'utilisateur sélectionne une couleur dans la boîte liste puis clique sur le bouton *SelectButton* pour enregistrer son choix et fermer la fiche. Le

gestionnaire d'événement *OnClick* du bouton *SelectButton* doit avoir la forme suivante :

```
procedure TColorForm.SelectButtonClick(Sender: TObject);
begin
  with ColorListBox do
    if ItemIndex >= 0 then
      String(FColor^) := ColorListBox.Items[ItemIndex];
  end;
  Close;
end;
```

Remarquez comment le gestionnaire d'événement stocke le nom de couleur sélectionné dans la chaîne référencée par le pointeur qui a été transmise au constructeur.

Pratiquement, pour utiliser *ColorForm*, la fiche appelante doit transmettre au constructeur un pointeur sur une chaîne existante. Supposons par exemple que *ColorForm* est instanciée par une fiche appelée *ResultsForm* en réponse au choix d'un bouton de *ResultsForm* nommé *UpdateButton*. Le gestionnaire d'événement de ce bouton doit avoir la forme suivante :

```
procedure TResultsForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
begin
  GetColor(Addr(MainColor));
  if MainColor <> '' then
    {faire quelque chose avec la chaîne MainColor}
  else
    {faire autre chose car aucune couleur n'a été sélectionnée}
end;

procedure GetColor(PColor: Pointer);
begin
  ColorForm := TColorForm.CreateWithColor(PColor, Self);
  ColorForm.ShowModal;
  ColorForm.Free;
end;
```

UpdateButtonClick crée une chaîne nommée MainColor. L'adresse de MainColor est transmise à la fonction *GetColor* qui crée *ColorForm* en transmettant comme argument au constructeur un pointeur sur MainColor. Dès que *ColorForm* est fermée, elle est détruite mais le nom de la couleur sélectionnée, s'il y en a une, est préservé dans MainColor. Sinon, MainColor contient une chaîne vide, ce qui indique clairement que l'utilisateur est sorti de *ColorForm* sans sélectionner une couleur.

Cet exemple utilise une variable chaîne pour stocker des informations provenant de la fiche modale. Il est possible d'utiliser des objets plus complexes en fonction de vos besoins. N'oubliez jamais qu'il faut laisser à la fiche appelante un moyen de savoir que la fiche modale a été fermée sans modification, ni sélection (dans l'exemple précédent en attribuant par défaut à MainColor une chaîne vide).

Réutilisation des composants et des groupes de composants

Delphi offre différentes possibilités d'enregistrer et de réutiliser le travail réalisé avec les composants :

- Les *modèles de composants* procurent une façon simple et rapide de configurer et d'enregistrer des groupes de composants. Voir "Création et utilisation des modèles de composants" à la page 6-14.
- Vous pouvez enregistrer les fiches, les modules de données et les projets dans le *Référentiel*. Vous disposez ainsi d'une base de données centrale composée d'éléments réutilisables et vous pouvez utiliser l'héritage de fiche pour diffuser les modifications. Voir "Utilisation du référentiel d'objets" à la page 5-22.
- Vous pouvez enregistrer des *cadres* sur la palette de composants ou dans le référentiel. Les cadres utilisent l'héritage de fiche et peuvent être incorporés dans des fiches ou dans d'autres cadres. Voir "Manipulation des cadres" à la page 6-15.
- Créer un *composant personnalisé* est la manière la plus complexe de réutiliser du code, mais c'est celle qui procure le plus de souplesse. Voir chapitre 40, "Présentation générale de la création d'un composant".

Création et utilisation des modèles de composants

Vous pouvez créer des modèles composés d'un ou de plusieurs composants. Après avoir organisé les composants sur une fiche, défini leurs propriétés et écrit du code pour eux, enregistrez-les sous la forme d'un *modèle de composant*. Par la suite, en sélectionnant le modèle dans la palette de composants, vous pouvez placer les composants préconfigurés sur une fiche en une seule étape ; toutes les propriétés et tout le code de gestion d'événement associés sont simultanément ajoutés à votre projet.

Une fois que vous avez placé un modèle sur une fiche, vous pouvez repositionner les composants indépendamment les uns des autres, redéfinir leurs propriétés et créer ou modifier Les gestionnaires d'événements qui leur sont associés comme si vous placiez chaque composant un par un.

Pour créer un modèle de composant,

1. Placez et organisez des composants sur une fiche. Dans l'inspecteur d'objets, définissez leurs propriétés et leurs événements comme souhaité.
2. Sélectionnez les composants. La manière la plus simple de sélectionner plusieurs composants consiste à faire glisser le pointeur de la souris au-dessus d'eux. Des poignées grises apparaissent dans les coins de chaque composant sélectionné.
3. Choisissez Composant | Créer un modèle de composant.
4. Spécifiez un nom pour le modèle dans la boîte texte Nom de composant. Le nom proposé par défaut est le type du premier composant sélectionné à l'étape 2 suivi du mot "Template". Par exemple, si vous sélectionnez un libellé

puis une boîte texte, le nom proposé est “TLabelTemplate”. Vous pouvez modifier ce nom, en veillant à ne pas utiliser un nom de composant existant.

5 Dans la boîte texte Page de palette, spécifiez la page de la palette de composants dans laquelle vous souhaitez placer le modèle. Si vous spécifiez une page qui n’existe pas, une nouvelle page est créée lorsque vous enregistrez le modèle.

6 Sous Icône de palette, sélectionnez une image bitmap pour représenter le modèle sur la palette. L’image bitmap proposée par défaut est celle utilisée par le type du premier composant sélectionné à l’étape 2. Pour rechercher d’autres images bitmap, cliquez sur Changer. Les dimensions de l’image bitmap que vous choisissez ne doivent pas dépasser 24 pixels sur 24 pixels.

7 Choisissez OK.

Pour supprimer des modèles de la palette de composants, choisissez Composant | Configurer la palette.

Manipulation des cadres

Un cadre (*TFrame*), comme une fiche, est un conteneur pour d’autres composants. Il utilise le même mécanisme de possession que les fiches lorsque les composants y sont instanciés et détruits automatiquement, et la même relation parent-enfant pour la synchronisation des propriétés de composants.

A divers égards, un cadre s’apparente davantage à un composant personnalisé qu’à une fiche. Les cadres peuvent être enregistrés sur la palette de composants pour en faciliter la réutilisation et imbriqués dans des fiches, dans d’autres cadres ou autres objets conteneur. Une fois qu’un cadre a été créé et enregistré, il continue de fonctionner comme une unité et d’hériter des modifications des composants qu’il contient (y compris d’autres cadres). Lorsqu’un cadre est incorporé dans un autre cadre ou dans une fiche, il continue d’hériter des modifications apportées au cadre dont il dérive.

Les cadres servent à organiser les groupes de contrôles utilisés en plusieurs endroits de votre application. Par exemple, si vous avez un bitmap utilisé par plusieurs fiches, vous pouvez le placer dans un cadre afin qu’une seule copie de ce bitmap soit incluse dans les ressources de votre application. Vous pouvez également décrire un ensemble de champs de saisie servant à modifier une table avec un cadre et l’utiliser chaque fois que vous souhaitez entrer les données de la table.

Création des cadres

Pour créer un cadre vide, choisissez Fichier | Nouveau cadre, ou Fichier | Nouveau et double-cliquez sur Cadre. Vous pouvez alors déposer des composants (y compris d’autres cadres) sur le nouveau cadre.

Il est généralement préférable, bien que non nécessaire, d'enregistrer les cadres en tant que partie d'un projet. Si vous souhaitez créer un projet ne contenant que des cadres et aucune fiche, choisissez Fichier | Nouveau | Application, fermez la nouvelle fiche et la nouvelle unité sans les enregistrer, puis choisissez Fichier | Nouveau | Cadre et enregistrez le projet.

Remarque Lorsque vous enregistrez des cadres, évitez d'utiliser les noms par défaut *Unit1*, *Project1* etc., car ils peuvent être à la source de conflits au moment de l'utilisation ultérieure des cadres.

A la conception, vous pouvez afficher n'importe quel cadre contenu dans le projet en cours en choisissant Voir | Fiches et en sélectionnant le cadre. Comme dans le cas des fiches et des modules de données, vous pouvez passer du concepteur de fiche au fichier fiche du cadre en cliquant avec le bouton droit et en choisissant Voir comme fiche ou Voir comme texte.

Ajout de cadres à la palette de composants

Les cadres sont ajoutés à la palette de composants comme les modèles de composants. Pour ajouter un cadre à la palette de composants, ouvrez le cadre dans le concepteur de fiche (vous ne pouvez pas utiliser un cadre incorporé dans un autre composant), cliquez avec le bouton droit sur le cadre et choisissez Ajouter à la palette. Lorsque la boîte de dialogue Information modèle de composant s'ouvre, sélectionnez un nom, une page de palette et une icône pour le nouveau modèle.

Utilisation et modification des cadres

Pour utiliser un cadre dans une application, vous devez le placer, directement ou indirectement, sur une fiche. Vous pouvez ajouter des cadres directement sur des fiches, sur d'autres cadres ou sur d'autres objets conteneur, comme des volets et des boîtes de défilement.

Le concepteur de fiche permet d'ajouter un cadre à une application de deux manières :

- Sélectionnez un cadre à partir de la palette de composants et déposez-le sur une fiche, un autre cadre ou un autre objet conteneur. Si nécessaire, le concepteur de fiche demande s'il est possible d'inclure le fichier unité du cadre dans votre projet.
- Sélectionnez *Cadres* à partir de la page Standard de la palette de composants et cliquez sur une fiche ou un autre cadre. Une boîte de dialogue s'ouvre sur une liste de cadres figurant déjà dans votre projet ; sélectionnez-en un et cliquez sur OK.

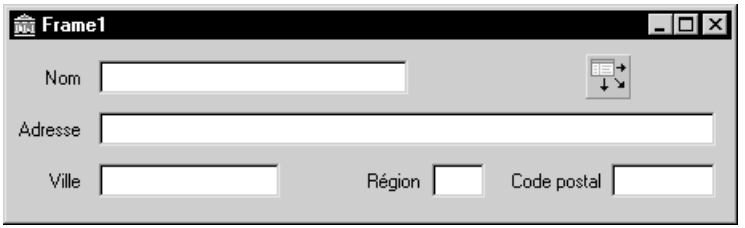
Lorsque vous déposez un cadre sur une fiche ou un autre conteneur, Delphi déclare une nouvelle classe qui descend du cadre que vous avez sélectionné. De même, lorsque vous ajoutez une nouvelle fiche à un projet, Delphi déclare une nouvelle classe qui descend de *TForm*. Cela signifie que les modifications apportées ultérieurement au cadre d'origine (ancêtre) sont répercutées sur le

cadre incorporé, mais que les modifications apportées au cadre incorporé ne sont pas répercutées sur le cadre ancêtre.

Supposons que vous souhaitiez regrouper des composants d'accès aux données et des contrôles orientés données en vue d'une utilisation fréquente, éventuellement dans plusieurs applications. Pour ce faire, vous pourriez rassembler les composants dans un modèle de composant ; mais si vous commencez à utiliser le modèle et changez d'avis ultérieurement sur l'organisation des contrôles, vous devez faire marche arrière et modifier manuellement dans chaque projet la partie sur laquelle le modèle a été placé.

Par contre, si vous placez vos composants base de données dans un cadre, les modifications ultérieures ne doivent être apportées que dans un seul endroit ; les modifications apportées à un cadre d'origine sont automatiquement répercutées sur ses descendants incorporés lors de la recompilation des projets. Parallèlement, vous pouvez modifier n'importe quel cadre incorporé sans affecter le cadre d'origine ni aucun de ses descendants incorporés. La seule restriction à la modification des cadres incorporés est que vous ne pouvez pas leur ajouter des composants.

Figure 6.1 Cadre avec des contrôles orientés données et un composant source de données



Outre une simplification de gestion, les cadres procurent une efficacité supplémentaire dans l'utilisation des ressources. Par exemple, pour utiliser une image bitmap ou un autre graphique dans une application, vous pouvez charger le graphique dans la propriété *Picture* d'un contrôle *TImage*. Toutefois, si vous utilisez fréquemment le même graphique dans une application, chaque objet *Image* que vous placez sur une fiche génère une autre copie du graphique ajouté au fichier ressource de la fiche. Cela est également vrai si vous définissez *TImage.Picture* une fois et enregistrez le contrôle *Image* en tant que modèle de composant. Une meilleure solution consiste à déposer l'objet *Image* sur un cadre, à y charger le graphique puis à utiliser le cadre là où vous souhaitez que le graphique apparaisse. Cela génère des fichiers fiche moins volumineux et présente en outre la possibilité de modifier le graphique partout où il figure en modifiant l'objet *Image* sur le cadre d'origine.

Partage des cadres

Vous pouvez partager un cadre avec les autres développeurs de deux manières :

- Ajouter le cadre au référentiel d'objet.
- Distribuer l'unité du cadre (.pas) et les fichiers fiche (.dfm ou .xfm).

Pour ajouter un cadre au référentiel, ouvrez n'importe quel projet contenant le cadre, cliquez avec le bouton droit dans le concepteur de fiche et choisissez Ajouter au référentiel. Pour plus d'informations, voir "Utilisation du référentiel d'objets" à la page 5-22.

Si vous envoyez une unité de cadre et les fichiers fiche à d'autres développeurs, ils peuvent les ouvrir et les ajouter à la palette de composants. Si le cadre contient d'autres cadres, ils devront l'ouvrir en tant que partie d'un projet.

Organisation des actions pour les barres d'outils et les menus

Delphi offre plusieurs fonctionnalités qui simplifieront votre travail de création, de personnalisation et de maintenance des menus et des barres d'outils. Ces fonctionnalités vous permettent d'organiser les listes des actions que les utilisateurs de votre application peuvent déclencher en appuyant sur un bouton dans une barre d'outils, en choisissant une commande dans un menu ou en cliquant sur une icône.

Très souvent, un même ensemble d'actions est utilisé pour plusieurs éléments de l'interface utilisateur. Par exemple, les commandes Couper, Copier et Coller apparaissent fréquemment à la fois dans un menu Edition et dans une barre d'outils. Il vous suffit d'ajouter l'action une fois pour l'utiliser dans plusieurs éléments de l'interface utilisateur de votre application.

Sur la plate-forme Windows, il existe des outils pour vous aider à définir et à grouper les actions, à créer diverses dispositions et à personnaliser les menus lors de la conception ou de l'exécution. Ces outils sont appelés outils ActionBand, les menus et les barres d'outils que vous créez en les utilisant sont appelés bandes d'action. En général, vous pouvez créer une interface utilisateur ActionBand comme suit :

- Construisez une liste d'actions afin de créer un ensemble d'actions disponibles pour votre application (utilisez le gestionnaire d'actions, *TActionManager*)
- Ajoutez les éléments de l'interface utilisateur à l'application (utilisez des composants ActionBand comme *TActionMainMenuBar*et *TActionToolBar*)
- "Glissez-déplacez" des actions du gestionnaire d'actions aux éléments de l'interface utilisateur

Le tableau suivant définit la terminologie qui s'applique à la définition des menus et des barres d'outils :

Tableau 6.1 Terminologie de la définition des actions

Terme	Définition
Action	Une réponse à ce que fait l'utilisateur, comme cliquer sur un élément de menu. De nombreuses actions standard fréquement nécessaires sont fournies ; vous pouvez les utiliser telles quelles dans vos applications. Par exemple, sont incluses les opérations sur les fichiers, comme ouvrir, enregistrer sous, exécuter et quitter, ainsi que de nombreuses autres pour l'édition, le formatage, la recherche, les dialogues ou les actions sur les fenêtres. Vous pouvez également programmer des actions personnalisées et utiliser les listes et le gestrionnaire d'actions pour y accéder.
Bande d'action	Un conteneur pour un ensemble d'actions associé à un menu ou à une barre d'outils personnalisés. Les composants ActionBand pour les menus principaux et les barres d'outils (*TActionMainMenuBar* et *TActionToolBar*) sont des exemples de bandes d'action.
Catégories d'actions	Vous permet de grouper des actions et de les introduire en tant que groupe dans un menu ou une barre d'outils. Par exemple, une des catégories d'actions standard, Search, inclut les actions Find, FindFirst, FindNext et Replace et les ajoute toutes en même temps.
Classes d'actions	Les classes qui réalisent les actions utilisées dans votre application. Toutes les actions standard sont définies dans des classes d'actions comme *TEditCopy*, *TEditCut* et *TEditUndo*. Vous pouvez utiliser ces classes en les faisant glisser de la boîte de dialogue Personnalisation à une bande d'action.
Client action	Représente la plupart du temps l'élément de menu ou le bouton qui reçoit une notification pour déclencher une action. Quand le client reçoit une commande utilisateur (par exemple, un clic de souris), il initie une action associée.
Liste d'actions	Maintiennent la liste des actions par lesquelles votre application répond à ce que fait l'utilisateur.
Gestionnaire d'actions	Groupe et organise des ensembles logiques d'actions pouvant être ré-utilisés dans les composants ActionBand. Voir *TActionManager*.
Menu	Présente la liste des commandes que l'utilisateur de l'application peut exécuter en cliquant dessus. Vous pouvez créer des menus en utilisant la classe menu ActionBand *TActionMainMenuBar*, ou en utilisant des composants multiplates-formes comme *TMainMenu* ou *TPopupMenu*.
Cible	Représente l'élément auquel s'applique l'action. La cible est généralement un contrôle, par exemple un mémo ou un contrôle de données. Certaines actions n'ont pas de cible. Ainsi, les actions standard d'aide ne tiennent pas compte de la cible mais démarrent simplement le système d'aide.
Barre d'outils	Affiche une ligne visible d'icônes de boutons qui, lorsque l'utilisateur clique dessus, font accomplir au programme une action, comme imprimer le document en cours. Vous pouvez créer des barres d'outils en utilisant le composant barre d'outils ActionBand *TActionToolBar*, ou en utilisant le composant multiplate-forme *TToolBar*.

Si vous développez pour plusieurs plates-formes, reportez-vous à "Utilisation des listes d'actions" à la page 6-26.

Qu'est-ce qu'une action ?

Pendant que vous développez votre application, vous pouvez créer un ensemble d'actions utilisables dans divers éléments de l'interface utilisateur. Vous pouvez les organiser en catégories et les insérer dans un menu en tant qu'ensemble (par exemple, Couper, Copier et Coller) ou un seul à la fois (par exemple, Outils | Personnaliser).

Une action correspond à un ou plusieurs éléments de l'interface utilisateur, comme les commandes de menu ou les boutons des barres d'outils. Les actions ont deux utilités : (1) elles représentent des propriétés communes à des éléments de l'interface utilisateur, par exemple l'état d'un contrôle activé ou coché, et (2) elles répondent lorsqu'un contrôle est déclenché, comme lorsque l'utilisateur de l'application clique sur un bouton ou choisit un élément de menu. Vous pouvez créer un catalogue d'actions disponibles dans votre application via des menus, des boutons, des barres d'outils, des menus contextuels, etc.

Les actions sont associées à d'autres composants :

- **Clients :** Un ou plusieurs clients utilisent l'action. Le client souvent représente un élément de menu ou un bouton (par exemple, *TToolButton*, *TSpeedButton*, *TMenuItem*, *TButton*, *TCheckBox, TRadioButton*, etc.). Des actions se trouvent également dans les composants ActionBand comme *TActionMainMenuBar et TActionToolBar.* Quand le client reçoit une commande de l'utilisateur (par exemple, un clic de souris), il initie une action associée. Généralement, l'événement *OnClick* d'un client est associé à l'événement *Execute* de son action.
- **Cible :** L'action agit sur la cible. La cible est généralement un contrôle, par exemple un mémo ou un contrôle de données. Les développeurs de composants peuvent créer des actions spécifiques aux besoins des contrôles qu'ils conçoivent et utilisent, puis empaqueter ces unités pour créer des applications plus modulaires. Certaines actions n'ont pas de cible. Ainsi, les actions standard d'aide ne tiennent pas compte de la cible mais démarrent simplement le système d'aide.

  Une cible peut également être un composant. Par exemple, les contrôles de données changent la cible par un ensemble de données associé.

Le client influence l'action — l'action répond lorsqu'un client déclenche l'action. L'action influence également le client — les propriétés des actions mettent à jour dynamiquement les propriétés des clients. Par exemple, si, à l'exécution, une action est désactivée (en définissant sa propriété *Enabled* par *False*), chaque client de cette action est désactivé et apparaît estompé.

Vous pouvez ajouter, supprimer et réorganiser des actions en utilisant le gestionnaire d'actions ou l'éditeur de liste d'actions (qui s'affiche lorsque vous double-cliquez sur un objet liste d'actions, *TActionList*). Ces actions sont ensuite connectées aux contrôles client.

Définition des bandes d'action

Les actions ne conservant aucune information de "disposition" (ni apparence ni position), Delphi propose les bandes d'action capables de stocker ces données. Les bandes d'action fournissent un mécanisme permettant de spécifier des informations de disposition et un jeu de contrôles. Vous pouvez afficher les actions comme des éléments de l'interface utilisateur tels que barres d'outils ou menus.

Vous organisez des ensembles d'actions en utilisant le gestionnaire d'actions (*TActionManager*). Vous pouvez soit utiliser des actions standard soit créer les vôtres.

Créez ensuite les bandes d'action :

- Utilisez *TActionMainMenuBar*pour créer un menu principal.
- Utilisez *TActionToolBar* pour créer une barre d'outils.

Les bandes d'action agissent comme conteneurs qui contiennent et affichent les ensembles d'actions. Pendant la conception, vous pouvez "glisser-déplacer" des éléments de l'éditeur du gestionnaire d'actions à la bande d'action. A l'exécution, les utilisateurs peuvent également personnaliser les menus ou les barres d'outils de l'application en utilisant une boîte de dialogue semblable à l'éditeur du gestionnaire d'actions.

Création des barres d'outils et des menus

Remarque Cette section décrit la méthode recommandée pour créer des menus et des barres d'outils dans les applications Windows. Pour le développement multiplate-forme, vous devez utiliser *TToolBar* et les composants menu, comme *TMainMenu*, et les organiser en utilisant les listes d'actions (*TActionList*). Voir "Définition des listes d'actions" à la page 6-26.

Vous utilisez le gestionnaire d'actions pour générer automatiquement des barres d'outils et des menus principaux en fonction des actions contenues dans votre application. Le gestionnaire d'actions gère les actions standard ainsi que les actions personnalisées que vous aurez éventuellement écrites. Créez ensuite les éléments de l'interface utilisateur basés sur ces actions et utilisez les bandes d'action pour afficher ces éléments d'action en tant qu'éléments dans un menu ou en tant que boutons dans une barre d'outils.

La procédure générale de création des menus, barres d'outils et autres bandes d'action comporte les étapes suivantes :

- Placer un gestionnaire d'actions sur une fiche.
- Ajouter des actions au gestionnaire d'actions, qui les organise en listes d'actions adaptées.
- Créer les bandes d'action (c'est-à-dire, le menu ou la barre d'outils) pour l'interface utilisateur.

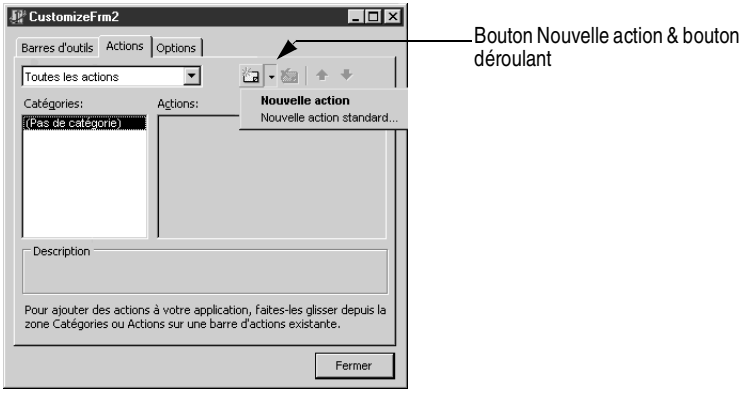
- Placer les actions dans l'interface de l'application par glisser-déplacer.

La procédure suivante décrit ces étapes en détail :

Pour créer des menus et des barres d'outils en utilisant les bandes d'action :

1. Depuis la page Supplément de la palette de composants, placez un composant gestionnaire d'actions (*TActionManager*) sur la fiche dans laquelle vous voulez créer la barre d'outils ou le menu.
2. Si vous voulez qu'il y ait des images sur le menu ou la barre d'outils, placez sur la fiche un composant liste d'images (*ImageList*) en le prenant dans la page Win32 de la palette de composants. (Vous devez ajouter les images que vous souhaitez utiliser à cette liste d'images ou utilisez les images fournies.)
3. Depuis la page Supplément de la palette de composants, placez sur la fiche un ou plusieurs des bandes d'action suivantes :
   - *TActionMainMenuBar* (pour concevoir des menus principaux)
   - *TActionToolBar* (pour concevoir des barres d'outils)
4. Connectez la liste d'images au gestionnaire d'actions : la focalisation étant sur le gestionnaire d'actions, sélectionnez dans l'inspecteur d'objets le nom de la liste d'images dans la propriété Images.
5. Ajoutez des actions au volet action de l'éditeur du gestionnaire d'actions :
   - Double-cliquez sur le gestionnaire d'actions afin d'afficher l'éditeur du gestionnaire d'actions.
   - Cliquez sur la flèche déroulante située à côté du bouton Nouvelle action (le bouton le plus à gauche dans le coin supérieur droit de la page Actions, comme illustré par la figure 6.2) et sélectionnez "Nouvelle action..." ou "Nouvelle action standard...". Une vue arborescente s'affiche. Ajoutez une ou plusieurs actions, ou catégories d'actions, au volet actions du gestionnaire d'actions. Le gestionnaire d'actions ajoute les actions à ses listes d'actions.

Figure 6.2 L'éditeur du gestionnaire d'actions.



6 "Glissez-déplacez" des actions indépendantes, ou des catégories d'actions, de l'éditeur du gestionnaire d'actions au menu ou à la barre d'outils que vous concevez.

Pour ajouter des actions définies par l'utilisateur, créez une nouvelle *TAction* en cliquant sur le bouton Nouvelle action et en écrivant le gestionnaire d'événement qui définit la réponse qu'elle donnera à son déclenchement. Voir "Que se passe-t-il lors du déclenchement d'une action ?" à la page 6-27, pour plus de détails. Lorsque vous avez défini les actions, vous pouvez les placer dans les menus ou les barres d'outils par glisser-déplacer comme les actions standard.

Ajout de couleurs, de motifs ou d'images aux menus, boutons et barres d'outils

Vous pouvez utiliser les propriétés *Background* et *BackgroundLayout* pour spécifier la couleur, le motif ou le bitmap à utiliser sur un élément de menu ou un bouton. Ces propriétés vous permettent également de définir une bannière qui se place à gauche ou à droite d'un menu.

Assignez des fonds et des dispositions aux sous-éléments à partir de leurs objets client action. Si vous voulez définir le fond des éléments d'un menu, cliquez dans le concepteur de fiches sur l'élément de menu qui contient ces éléments. Par exemple, sélectionner Fichier vous permet de changer le fond des éléments apparaissant dans le menu Fichier. Vous pouvez attribuer une couleur, un motif, ou un bitmap à la propriéte *Background* dans l'inspecteur d'objets.

Utilisez la propriété *BackgroundLayout* pour décrire comment placer le fond sur l'élément. Les couleurs ou les images peuvent être placées derrière le libellé normalement, étirées pour remplir à la zone, ou en mosaïque de petits carrés recouvrant toute la zone.

Les éléments dont le fond est normal (blNormal), étiré (blStretch), ou en mosaïque (blTile) sont affichés avec un fond transparent. Si vous créez une bannière, l'image complète est placée à gauche (blLeftBanner) ou à droite (blRightBanner) de l'élément. Vous devez vous assurer que la taille convient car elle ne'st ni réduite ni agrandie.

Pour modifier le fond d'une bande d'action (c'est-à-dire, dans un menu principal ou une barre d'outils) sélectionnez la bande d'action et choisissez la *TActionClientBar* via l'éditeur de collection de bandes d'action. Vous pouvez définir les propriétés *Background* et *BackgroundLayout* pour spécifier la couleur, le motif ou le bitmap à utiliser sur toute la barre d'outils ou tout le menu.

Ajout d'icônes aux menus et aux barres d'outils

Vous pouvez ajouter des icônes à côté des éléments de menu ou remplacer les libellés figurant dans les barres d'outils par des icônes. Vous organiserez les bitmaps ou les icônes au moyen d'une liste d'images.

1 A partir de la page Win32 de la palette de composants, placez sur la fiche un composant liste d'images.

2. Ajoutez à la liste d'images les images que vous voulez utiliser. Double-cliquez sur la liste d'images. Cliquez sur Ajouter et naviguez jusqu'aux images que vous voulez utiliser ; cliquez sur OK lorsque vous en avez terminé. Vous trouverez des exemples d'images dans Program Files\Common Files\Borland Shared\Images. (Les images des boutons comprennent deux vues pour les états actif et inactif des boutons.)
3. Depuis la page Supplément de la palette de composants, placez sur la fiche un ou plusieurs des bandes d'action suivantes :
   - *TActionMainMenuBar* (pour concevoir des menus principaux)
   - *TActionToolBar* (pour concevoir des barres d'outils)
4. Connectez la liste d'images au gestionnaire d'actions. Donnez d'abord la focalisation au gestionnaire d'actions. Ensuite, sélectionnez dans l'inspecteur d'objets le nom de la liste d'images dans la propriété Images.
5. Utilisez l'éditeur du gestionnaire d'actions pour ajouter des actions au gestionnaire d'actions. Vous pouvez associer une image à une action en définissant la propriété *ImageIndex* de l'action par le numéro d'ordre de l'image dans la liste d'images.
6. "Glissez-déplacez" des actions indépendantes ou des catégories d'actions de l'éditeur du gestionnaire d'actions au menu ou à la barre d'outils.
7. Pour les barres d'outils, où vous voulez uniquement afficher l'icône et pas le libellé : sélectionnez la bande d'action Barre d'outils et double-cliquez sur sa propriété Items. Dans l'éditeur de collections, vous pouvez sélectionner un ou plusieurs éléments et définir leurs propriétés Caption.
8. Les images apparaissent automatiquement dans le menu ou la barre d'outils.

Création de barres d'outils et de menus personnalisables par l'utilisateur

Vous pouvez utiliser les bandes d'action avec le gestionnaire d'actions pour créer des menus et des barres d'outils personnalisables. A l'exécution, les utilisateurs peuvent personnaliser les barres d'outils et les menus (bandes d'action) de l'interface utilisateur de l'application en utilisant une boîte de dialogue semblable à l'éditeur du gestionnaire d'actions.

Pour permettre aux utilisateurs de personnaliser une bande d'action de l'application qu'ils exécutent :

1. Placez un gestionnaire d'actions sur une fiche.
2. Placez vos composants bande d'action (*TActionMainMenuBar, TActionToolBar*).
3. Double-cliquez sur le gestionnaire d'actions afin d'afficher l'éditeur du gestionnaire d'actions.
   - Sélectionnez les actions que vous voulez utiliser dans votre application. Ajoutez également l'action Customize, qui apparaît à la fin de la liste des actions standard.
   - Placez sur la fiche un composant *TCustomizeDlg* depuis la page Dialogues, connectez-le au gestionnaire d'actions en utilisant sa propriété

ActionManager. Spécifiez un nom de fichier pour enregistrer dans un flux les personnalisations effectuées par les utilisateurs.

- Placez les actions dans les composants bande d'action par glisser-déplacer. (Vérifiez que vous avez ajouté l'action Customize à la barre d'outils ou au menu.)

4 Terminez l'application.

Lorsque vous compilez et exécutez l'application, les utilisateurs ont accès à une commande Personnaliser qui affiche une boîte de dialogue de personnalisation semblable à l'éditeur du gestionnaire d'actions. Ils peuvent "glisser-déplacer" des éléments de menu et créer des barres d'outils en utilisant les actions que vous avez fournies dans le gestionnaire d'actions.

Cacher les éléments et les catégories inutilisés dans les bandes d'action

Un des avantages de l'utilisation des ActionBands est que les éléments et les catégories inutilisés peuvent être cachés à l'utilisateur. Avec le temps, les bandes d'action s'adaptent aux utilisateurs de l'application en montrant les éléments qu'ils utilisent et en cachant les autres. Les éléments cachés peuvent redevenir visibles : il suffit que l'utilisateur appuie sur un bouton déroulant. De plus, l'utilisateur peut restaurer la visibilité de tous les éléments d'une bande d'action en réinitialisant les statistiques d'usage dans la boîte de dialogue de personnalisation. Le masquage des éléments fait partie du comportement par défaut des bandes d'action, mais ce comportement peut être modifié afin d'inhiber le masquage d'éléments particuliers, de tous les éléments d'une collection particulière (par exemple, le menu Fichier), ou de tous les éléments d'une bande d'action particulière.

Le gestionnaire d'actions mémorise le nombre de fois qu'une action a été invoquée par l'utilisateur en l'enregistrant dans le champ *UsageCount* du *TActionClientItem* correspondant. Le gestionnaire d'actions enregistre également le nombre de fois que l'application a été exécutée (ce qui correspond au numéro de la dernière session), ainsi que le numéro de la session où une action a été utilisée pour la dernière fois. La valeur de *UsageCount* sert à rechercher le nombre maximal de sessions pendant lesquelles un élément est inutilisé avant d'être masqué, il ensuite est comparé avec la différence entre le numéro de session actuel et le numéro de session de la dernière utilisation de l'élément. Si cette différence est supérieure au nombre défini dans *PrioritySchedule*, l'élément est caché. Les valeurs par défaut de *PrioritySchedule* sont indiquées dans le tableau suivant :

Tableau 6.2 Valeurs par défaut de la propriété PrioritySchedule du gestionnaire d'actions

Nombre de sessions dans lesquelles un élément d'une bande d'action a été utilisé	Nombre de sessions pendant lesquelles un élément restera visible après sa dernière utilisation
0, 1	3
2	6
3	9
4, 5	12

Tableau 6.2 Valeurs par défaut de la propriété PrioritySchedule du gestionnaire d'actions (suite)

Nombre de sessions dans lesquelles un élément d'une bande d'action a été utilisé	Nombre de sessions pendant lesquelles un élément restera visible après sa dernière utilisation
6-8	17
9-13	23
14-24	29
25 ou plus	31

Il est possible de désactiver le masquage d'un élément au moment de la conception. Pour empêcher le masquage d'une action particulière (et de toutes les collections qui la contiennent), recherchez son objet TActionClientItem et définissez son *UsageCount* par -1. Pour empêcher le masquage d'une collection entière d'éléments, comme le menu Fichier ou même la barre de menus principale, recherchez l'objet *TActionClients* qui lui est associé et définissez sa propriété *HideUnused* par *False*.

Utilisation des listes d'actions

Remarque Cette section concerne la définition des barres d'outils et des menus dans le développement multiplates-formes. Vous pouvez utiliser les méthodes décrites ici pour le développement Windows. Mais, l'utilisation des bandes d'action est plus simple et offre plus d'options. Les listes d'actions seront gérées automatiquement par le gestionnaire d'actions. Voir "Organisation des actions pour les barres d'outils et les menus" à la page 6-18, pour plus d'informations sur l'utilisation des bandes d'action et du gestionnaire d'actions.

Les listes d'actions maintiennent la liste des actions par lesquelles votre application répond à ce que fait l'utilisateur. En utilisant les objets action, vous centralisez les fonctions accomplies par votre application à partir de l'interface utilisateur. Cela vous permet de partager le code réalisant les actions (par exemple, lorsqu'un bouton de barre d'outils et un élément de menu effectuent la même tâche), et procure un lieu unique et centralisé d'activer et de désactiver des actions selon l'état de l'application.

Définition des listes d'actions

La définition des listes d'actions est très simple une fois comprises les étapes qu'elle implique :

- Créer la liste d'actions.
- Ajouter des actions à la liste d'actions.
- Définir des propriétés pour les actions.
- Attacher des clients à l'action.

Voici le détail de ces étapes :

1 Placez un objet *TActionList* dans votre fiche ou votre module de données. (ActionList appartient à la page Standard de la palette de composants.)

2. Double-cliquez sur l'objet *TActionList* pour afficher l'éditeur de liste d'actions.
   1. Utilisez une des actions prédéfinies énumérées dans l'éditeur : cliquez avec le bouton droit et choisissez Nouvelle action standard.
   2. Les actions prédéfinies sont groupées en catégories (ensemble de données, édition, aide et fenêtre) dans la boîte de dialogue Classes d'actions standard. Sélectionnez toutes les actions standard que vous voulez ajouter à votre liste d'actions et cliquez sur OK.

      ou

   3. Créez une nouvelle action qui vous sera propre : cliquez avec le bouton droit et choisissez Nouvelle action.
3. Définissez les propriétés de chaque action dans l'inspecteur d'objets. (Les propriétés que vous définissez affectent chaque client de l'action.)

   La propriété *Name* identifie l'action, les autres propriétés et événements (*Caption*, *Checked*, *Enabled*, *HelpContext*, *Hint*, *ImageIndex*, *ShortCut*, *Visible* et *Execute*) correspondent aux propriétés et événements de ses contrôles client. Elles portent généralement, mais pas obligatoirement, le même nom que la propriété du client. Par exemple, la propriété *Enabled* d'une action correspond à la propriété *Enabled* d'un *TToolButton*. Mais, la propriété *Checked* d'une action correspond à la propriété *Down* d'un *TToolButton*.
4. Si vous utilisez les actions prédéfinies, l'action inclut une réponse standard qui intervient automatiquement. Si vous créez votre propre action, vous devez écrire un gestionnaire d'événement définissant comment l'action répond lorsqu'elle est déclenchée. Voir "Que se passe-t-il lors du déclenchement d'une action ?" à la page 6-27, pour plus de détails.
5. Attachez les actions de la liste d'actions aux clients qui le nécessitent :
   - Cliquez sur le contrôle (par exemple, le bouton ou l'élément de menu) dans la fiche ou le module de données. Dans l'inspecteur d'objets, la propriété *Action* est la liste des actions disponibles.
   - Sélectionnez celle que vous souhaitez.

Les actions standard, comme *TEditDelete* ou *TDataSetPost*, accomplissent l'action attendue. L'aide en ligne de référence peut vous donner tous les détails sur le fonctionnement de chacune des actions standard. Si vous écrivez vos propres actions, vous aurez besoin de mieux comprendre ce qui se passe lorsqu'une action est déclenchée.

Que se passe-t-il lors du déclenchement d'une action ?

Lorsqu'un événement est déclenché, il se produit la série d'événements qui s'applique d'abord aux actions génériques. Ensuite, si l'événement ne gère pas l'action, une autre séquence d'événements intervient.

Réponse par les événements

Quand un composant ou un contrôle client est cliqué ou subit un autre type d'opération, se produit une série d'événements auxquels vous pouvez répondre. Par exemple, le code suivant est le gestionnaire d'événement pour une action qui inverse la visibilité d'une barre d'outils quand une action est exécutée :

```
procedure TForm1.Action1Execute(Sender: TObject);
begin
   { Inverse la visibilité de Toolbar1 }
   ToolBar1.Visible := not ToolBar1.Visible;
end;
```

Remarque Pour des informations générales sur les événements et les gestionnaires d'événements, voir Utilisation des événements et des gestionnaires d'événements"Utilisation des événements et des gestionnaires d'événements" à la page 3-28.

Vous pouvez fournir un gestionnaire d'événement qui réponde à l'un des trois différents niveaux : action, liste d'actions ou application. Cela ne vous concerne que si vous utilisez une nouvelle action générique et non une action standard prédéfinie. En effet, les actions standard ont un comportement intégré qui s'exécute automatiquement lorsque ces événements se produisent.

L'ordre dans lequel les gestionnaires d'événements répondront aux événements est le suivant :

- Liste d'actions
- Application
- Action

Lorsque l'utilisateur clique sur un contrôle client, Delphi appelle la méthode Execute de l'action qui s'adresse d'abord à la liste d'actions, puis à l'objet Application, enfin à l'action elle-même lorsque ni la liste d'actions ni l'application ne la gère. Delphi suit cette séquence de répartition lors de la recherche d'une façon de répondre à l'action de l'utilisateur :

1. Si vous fournissez un gestionnaire d'événement *OnExecute* pour la liste d'actions et qu'il gère l'action, l'application se poursuit.

   Le gestionnaire d'événement de la liste d'actions dispose d'un paramètre *Handled* qui renvoie *False* par défaut. Si le gestionnaire est défini et gère l'événement, il renvoie *True* et la séquence de traitement s'arrête là. Par exemple :

   ```
   procedure TForm1.ActionList1ExecuteAction(Action: TBasicAction; var Handled: Boolean);
   begin
      Handled := True;
   end;
   ```

   Si vous ne définissez pas *Handled* par *True* dans le gestionnaire d'événement de la liste d'actions, le traitement se poursuit.

2. Si vous n'avez pas écrit de gestionnaire d'événement *OnExecute* pour la liste d'actions ou si le gestionnaire ne gère pas l'action, le gestionnaire d'événement

OnActionExecute de l'application est déclenché. S'il gère l'action, l'application continue.

L'objet *Application* global reçoit un événement *OnActionExecute* si l'une des listes d'actions de l'application échoue en gérant un événement. Comme le gestionnaire d'événement *OnExecute* de la liste d'actions, le gestionnaire *OnActionExecute* dispose d'un paramètre *Handled* qui renvoie *False* par défaut. Si un gestionnaire d'événement est défini et gère l'événement, il renvoie *True* et la séquence de traitement s'arrête ici. Par exemple :

```
procedure TForm1.ApplicationExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
  { Empêche l'exécution de toutes les actions de Application }
  Handled := True;
end;
```

3 Si le gestionnaire d'événement *OnExecute* de l'application ne gère pas l'action, le gestionnaire d'événement *OnExecute* de l'action est déclenché.

Vous pouvez utiliser des actions intégrées ou créer vos propres classes d'actions qui savent comment opérer sur des classes cibles spécifiques (telles que les contrôles de saisie). Quand aucun gestionnaire d'événement n'est trouvé à n'importe quel niveau, l'application essaie ensuite de trouver une cible sur laquelle exécuter l'action. Quand l'application trouve une cible pour laquelle l'action sait quoi faire, elle déclenche l'action. Voir la section suivante pour davantage d'informations sur la manière dont l'application trouve une cible pouvant correspondre à une classe d'actions prédéfinie.

Comment les actions trouvent leurs cibles

"Que se passe-t-il lors du déclenchement d'une action ?" à la page 6-27 décrit le cycle d'exécution se produisant lorsqu'un utilisateur déclenche une action. S'il n'y a pas de gestionnaire d'événement défini pour répondre à l'action, que ce soit au niveau de la liste d'actions, de l'application ou de l'action, l'application essaie d'identifier un objet cible auquel l'action peut s'appliquer.

L'application recherche un objet cible selon la séquence suivante :

1 Contrôle actif : L'application recherche d'abord un contrôle actif comme cible potentielle.

2 Fiche active : Si l'application ne trouve pas de contrôle actif ou si le contrôle actif ne peut servir de cible, elle recherche la fiche active à l'écran (*ActiveForm*).

3 Contrôles de la fiche : Si la fiche active n'est pas une cible appropriée, l'application recherche une cible dans les autres contrôles de la fiche active.

Si aucune cible n'est trouvée, rien ne se passe lorsque l'événement est déclenché.

Certains contrôles peuvent étendre la recherche afin de déférer la cible à un composant qui leur est associé ; par exemple, les contrôles orientés données défèrent la cible au composant ensemble de données qui leur est associé. D'autre part, certaines actions prédéfinies n'utilisent pas de cible, par exemple, le dialogue d'ouverture de fichiers.

Actualisation des actions

Lorsquel'application est inactive, l'événement *OnUpdate* se produit pour chaque action liée à un contrôle ou un élément de menu affiché. Ceci permet aux applications d'exécuter un code centralisé pour l'activation et la désactivation, la sélection et la désélection, et ainsi de suite. Par exemple, le code suivant illustre le gestionnaire d'événement *OnUpdate* pour une action qui est "cochée" quand la barre d'outils est visible.

```
procedure TForm1.Action1Update(Sender: TObject);
begin
   { Indique si ToolBar1 est actuellement visible }
   (Sender as TAction).Checked := ToolBar1.Visible;
end;
```

Attention Ne placez pas de code nécessitant une exécution longue dans le gestionnaire d'événement *OnUpdate*. En effet, il est exécuté à chaque fois que l'application est inactive. Si l'exécution de ce gestionnaire d'événement est trop longue, les performances de toute l'application s'en trouvent affectées.

Classes d'actions prédéfinies

Vous pouvez ajouter des actions prédéfinies à votre application en cliquant avec le bouton droit sur le gestionnaire d'actions et en choisissant Nouvelle action standard. La boîte de dialogue Classes d'actions standard s'affiche et présente la liste des classes d'actions prédéfinies avec les actions standard qui leur sont associées. Ce sont des actions qui sont incluses dans Delphi et il existe des objets qui accomplissent ces actions automatiquement. Les actions prédéfinies sont organisées dans les classes suivantes :

Tableau 6.3 Classes d'actions

Classe	Description
Edition	Actions d'édition standard : Utilisées avec une cible contrôle de saisie. *TEditAction* est la classe de base de descendants qui redéfinissent la méthode *ExecuteTarget* afin d'implémenter les opérations copier, couper et coller en utilisant le presse-papiers.
Formatage	Actions de formatage standard : Utilisées avec le texte formaté pour appliquer des options de formatage du texte telles que gras, italique, souligné, barré, etc. *TRichEditAction*est la classe de base de descendants qui redéfinissent les méthodes *ExecuteTarget* et *UpdateTarget* afin d'implémenter le formatage de la cible.
Aide	Actions d'aide standard : Utilisées avec toute cible. *THelpAction* est la classe de base de descendants qui redéfinissent la méthode *ExecuteTarget* pour transmettre les commandes à un système d'aide.
Fenêtre	Actions de fenêtre standard : Utilisées avec les fiches d'une application MDI comme cible. *TWindowAction* est la classe de base de descendants qui redéfinissent la méthode *ExecuteTarget* pour implémenter la réorganisation, la cascade, la fermeture, la mosaïque et la réduction de fenêtres enfant MDI.

Tableau 6.3 Classes d'actions (suite)

Classe	Description
Fichier	Actions de fichiers : Utilisées avec les opérations sur les fichiers comme ouvrir, exécuter ou quitter.
Recherche	Actions de recherche : Utilisées avec les options de recherche. *TSearchAction* implémente le comportement commun des actions affichant un dialogue non modal où l'utilisateur peut entrer une chaîne pour rechercher un contrôle de saisie.
Tab	Actions des contrôles onglet : Utilisées pour le déplacement entre les onglets dans un contrôle à onglets, comme les boutons Précédent et Suivant d'un expert.
Liste	Actions des contrôles liste : Utilisées pour gérer les éléments d'une vue liste.
Dialogue	Actions de dialogue : Utilisées avec les composants dialogue. *TDialogAction* implémente le comportement commun des actions affichant un dialogue lors de leur exécution. Chaque classe dérivée représente un dialogue spécifique.
Internet	Actions Internet : Utilisées pour des fonctions comme la navigation, le téléchargement et l'envoie de mail sur Internet.
Ensemble de données	Actions d'ensembles de données : Utilisées avec un composant ensemble de données comme cible. *TDataSetAction* est la classe de base de descendants qui redéfinissent les méthodes *ExecuteTarget* et *UpdateTarget* afin d'implémenter les déplacements et les éditions de la cible. *TDataSetAction* introduit une propriété *DataSource* qui garantit que les actions sont effectuées sur l'ensemble de données. Si *DataSource* vaut nil, le contrôle orienté données détenant la focalisation est utilisé.
Outils	Outils : Des outils supplémentaires comme *TCustomizeActionBars* qui affiche automatiquement la boîte de dialogue permettant de personnaliser les bandes d'action.

Tous les objets action sont décrits sous leur nom dans l'aide en ligne de référence. Reportez-vous à l'aide pour avoir des détails sur leur fonctionnement.

Conception de composants utilisant des actions

Vous pouvez également créer vos propres classes d'actions prédéfinies. Quand vous écrivez vos propres classes d'actions, vous pouvez leur attribuer la capacité à s'exécuter sur certaines classes d'objet cible. Vous pouvez ensuite utiliser vos actions personnalisées de la même façon que les classes d'actions prédéfinies. Ainsi, si l'action détermine qu'elle peut s'appliquer à une classe cible, il suffit d'affecter l'action au contrôle client et il agit sur la cible sans avoir besoin d'écrire un gestionnaire d'événement.

Les concepteurs de composants peuvent utiliser comme exemple les classes définies dans les unités StdActns et DBActns afin de dériver leurs propres classes d'actions qui implémentent des comportements spécifiques à leurs contrôles et leurs composants. Les classes de base de ces actions spécialisées (*TEditAction*, *TWindowAction*) surchargent généralement *HandlesTarget*, *UpdateTarget* et d'autres

méthodes afin de limiter la cible de l'action à une classe spécifique d'objets. Les classes dérivées surchargent habituellement *ExecuteTarget* pour réaliser une tâche spécifique. Voici ces méthodes :

Méthode	Description
HandlesTarget	Automatiquement appelée lorsque l'utilisateur invoque un objet (comme un bouton de barre d'outils ou un élément de menu) lié à l'action. La méthode *HandlesTarget* laisse l'objet action indiquer s'il convient de l'exécuter à ce moment avec comme "cible" l'objet spécifié par le paramètre *Target*. Voir "Comment les actions trouvent leurs cibles" à la page 6-29, pour plus de détails.
UpdateTarget	Automatiquement appelée lorsque l'application est inactive pour que les actions puissent se mettre à jour elles-mêmes en fonction des conditions en cours. Utilisez à la place de *OnUpdateAction*. Voir "Actualisation des actions" à la page 6-30, pour plus de détails.
ExecuteTarget	Automatiquement appelée lorsque l'action est déclenchée en réponse à une action de l'utilisateur à la place de *OnExecute* (par exemple, quand l'utilisateur sélectionne un élément de menu ou appuie sur un bouton de barre d'outils qui est lié à cette action). Voir "Que se passe-t-il lors du déclenchement d'une action ?" à la page 6-27, pour plus de détails.

Recensement d'actions

Si vous écrivez vos propres actions, il est possible de les recenser pour les faire apparaître dans l'éditeur de liste d'actions. Vous les recensez ou vous en annulez le recensement en utilisant les routines globales de l'unité *ActnList* :

```
procedure RegisterActions(const CategoryName: string; const AClasses: array of
TBasicActionClass; Resource: TComponentClass);

procedure UnRegisterActions(const AClasses: array of TBasicActionClass);
```

Quand vous appelez *RegisterActions*, les actions recensées apparaissent dans l'éditeur de liste d'actions afin d'être utilisées dans vos applications. Vous pouvez spécifier un nom de catégorie afin d'organiser vos actions, ainsi qu'un paramètre *Resource* permettant de fournir des valeurs de propriété par défaut.

Par exemple, le code suivant recense dans l'EDI les actions standard :

```
{ recensement des actions standard }

RegisterActions('', [TAction], nil);

RegisterActions('Edit', [TEditCut, TEditCopy, TEditPaste], TStandardActions);

RegisterActions('Window', [TWindowClose, TWindowCascade, TWindowTileHorizontal,
TWindowTileVertical, TWindowMinimizeAll, TWindowArrange], TStandardActions);
```

Quand vous appelez *UnRegisterActions*, les actions n'apparaissent plus dans l'éditeur de liste d'actions.

Création et gestion de menus

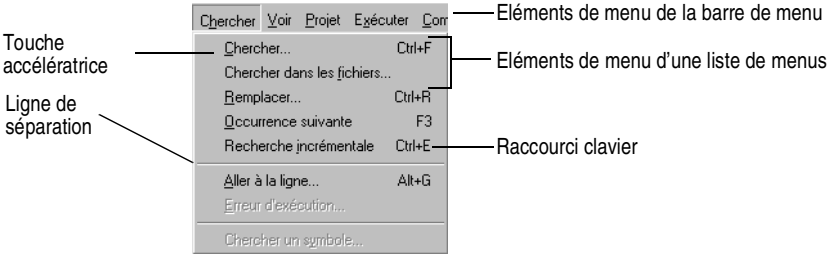
Les menus constituent pour les utilisateurs un moyen commode d'exécuter des commandes regroupées logiquement. Le concepteur de menus vous permet d'ajouter facilement à une fiche un menu prédéfini ou personnalisé. Ajoutez un composant menu à la fiche, ouvrez le concepteur de menus et saisissez directement les éléments de menu dans la fenêtre du concepteur de menus. Vous pouvez ajouter ou supprimer des éléments de menu et utiliser le glisser-déplacer pour réorganiser les éléments à la conception.

Vous n'avez même pas besoin d'exécuter votre programme pour voir le résultat : ce que vous concevez est immédiatement visible dans la fiche et apparaît exactement comme à l'exécution. En utilisant du code, vous pouvez également modifier les menus à l'exécution afin de proposer à l'utilisateur davantage d'informations ou d'options.

Ce chapitre décrit la manière d'utiliser le concepteur de menus pour concevoir des barres de menus et des menus surgissants (locaux). Les sujets suivants, concernant la manipulation des menus à la conception et à l'exécution, sont abordés :

- Ouverture du concepteur de menus
- Construction des menus
- Edition des éléments de menu dans l'inspecteur d'objets
- Utilisation du menu contextuel du concepteur de menus
- Utilisation des modèles de menu
- Enregistrement d'un menu comme modèle
- Ajout d'images à des éléments de menu

Figure 6.3 Terminologie des menus

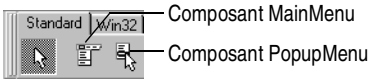


Pour des informations sur la façon de lier un élément de menu à du code qui s'exécute lorsque l'élément est sélectionné, voir "Association d'événements de menu à des gestionnaires d'événements" à la page 3-31.

Ouverture du concepteur de menus

Vous concevez les menus de votre application à l'aide du concepteur de menus. Avant de commencer à utiliser le concepteur de menus, ajoutez à votre fiche un composant *TMainMenu* ou *TPopupMenu*. Ces deux composants menu se trouvent dans la page Standard de la palette de composants.

Figure 6.4 Composants menu principal et menu surgissant



Un composant menu principal (MainMenu) crée un menu attaché à la barre de titre de la fiche. Un composant menu surgissant (PopupMenu) crée un menu qui apparaît quand l'utilisateur clique avec le bouton droit de la souris dans la fiche. Les menus surgissants n'ont pas de barre de menu.

Pour ouvrir le concepteur de menus, sélectionnez un composant menu de la fiche, puis :

- Double-cliquez sur le composant menu.

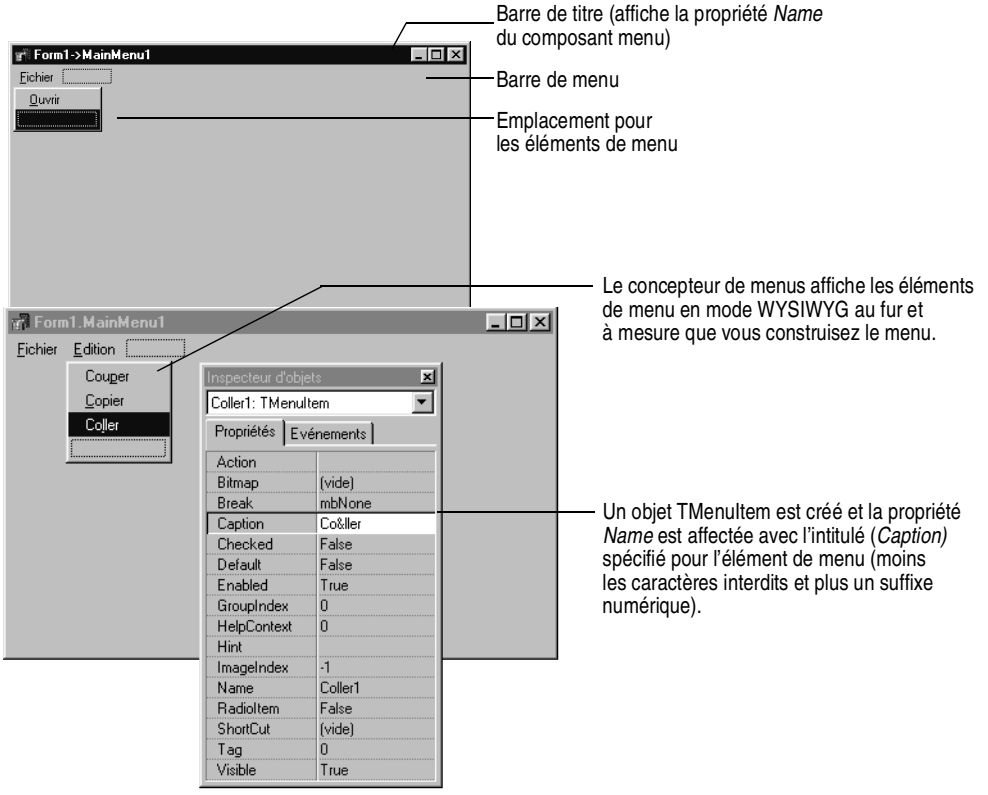
  ou

- Dans la page Propriétés de l'inspecteur d'objets, sélectionnez la propriété *Items* puis double-cliquez sur [Menu] dans la colonne des valeurs ou cliquez sur le bouton points de suspension (...).

  Le concepteur de menus apparaît, le premier élément de menu (vide) est sélectionné dans le concepteur et la propriété *Caption* est sélectionnée dans l'inspecteur d'objets.

Figure 6.5 Concepteur pour un menu surgissant

Figure 6.6 Concepteur pour un menu principal



Construction des menus

Vous devez ajouter un composant menu à vos fiches pour chaque menu devant apparaître dans l'application. Vous pouvez créer chaque structure de menu à partir de zéro ou partir de l'un des modèles de menu prédéfinis.

Cette section décrit les principes de base de la création de menus à la conception. Pour davantage d'informations sur les modèles de menu, voir "Utilisation des modèles de menu" à la page 6-43.

Nom des menus

Comme pour tous les composants, Delphi donne un nom par défaut au composant menu que vous ajoutez à une fiche, par exemple *MainMenu1*. Vous pouvez attribuer au menu un nom plus explicite respectant les conventions de nom du Pascal Objet.

Delphi ajoute le nom du menu à la déclaration de type de la fiche et le nom du menu apparaît dans la liste des composants.

Nom des éléments de menu

A la différence des composants menu, vous devez donner vous-même explicitement un nom aux éléments de menu quand vous les ajoutez à la fiche. Vous pouvez le faire de deux manières :

- En saisissant directement la valeur de la propriété *Name*.
- En saisissant d'abord la valeur de la propriété *Caption*, puis en laissant Delphi en dériver une valeur pour la propriété *Name*.

  Si, par exemple, vous spécifiez *Fichier* comme valeur de la propriété *Caption*, Delphi affecte la valeur *Fichier1* à la propriété *Name* de l'élément de menu. Si vous renseignez d'abord la propriété *Name* avant de renseigner la propriété *Caption*, Delphi laisse la propriété *Caption* vide jusqu'à ce que vous saisissiez une valeur.

Remarque Si vous saisissez dans la propriété *Caption* des caractères qui ne sont pas autorisés dans un identificateur Pascal Objet, Delphi modifie la propriété *Name* en conséquence. Si par exemple, l'intitulé commence par un chiffre, Delphi fait précéder le chiffre d'un caractère pour en dériver la valeur de la propriété *Name*.

Le tableau suivant donne des exemples de ce processus en supposant que tous ces éléments sont placés dans la même barre de menu.

Tableau 6.4 Exemples d'intitulés et des noms dérivés

Intitulé du composant	Nom dérivé	Explication
&Fichier	Fichier1	Retire le &
&Fichier (une deuxième fois)	Fichier2	Numérote les éléments répétés
1234	N12341	Ajoute une lettre au début et numérote en fin
1234 (une deuxième fois)	N12342	Ajoute un nombre pour que le nom dérivé ne soit plus ambigu.
$@@@#	N1	Supprime tous les caractères non standard, ajoute une lettre au début et un numéro d'ordre
- (signe moins)	N2	Numérote cette deuxième occurrence d'un intitulé sans aucun caractère standard

Comme pour le composant menu, Delphi ajoute le nom des éléments de menu à la déclaration de type de la fiche et leur nom apparaît dans la liste des composants.

Ajout, insertion et suppression d'éléments de menu

Les procédures suivantes décrivent la manière d'effectuer les opérations de base intervenant dans la conception d'une structure de menu. Chaque procédure suppose que la fenêtre du concepteur de menus est déjà ouverte.

Pour ajouter des éléments de menu à la conception :

1 Sélectionnez la position à laquelle vous voulez créer l'élément de menu.

Si vous venez juste d'ouvrir le concepteur de menus, la première position est déjà sélectionnée dans la barre de menu.

2 Commencez à saisir l'intitulé. Vous pouvez aussi commencer à saisir la propriété *Name* en plaçant le curseur dans l'inspecteur d'objets et en saisissant une valeur. Dans ce cas, vous devez resélectionner la propriété *Caption* et entrer une valeur.

3 Appuyez sur *Entrée*.

L'emplacement suivant d'élément de menu est sélectionné.

Si vous entrez d'abord la propriété *Caption*, utilisez les touches de déplacement pour revenir dans l'élément de menu que vous venez de saisir. Vous pouvez constater que Delphi a renseigné la propriété *Name* en partant de la valeur saisie dans l'intitulé. (Voir "Nom des éléments de menu" à la page 6-36.)

4 Continuez à saisir la valeur des propriétés *Name* et *Caption* pour chaque élément de menu à ajouter ou appuyez sur *Echap* pour revenir à la barre de menu.

Utilisez les touches de déplacement pour passer de la barre de menu au menu, puis pour vous déplacer dans les éléments de la liste ; appuyez sur *Entrée* pour achever une action. Pour revenir à la barre de menu, appuyez sur *Echap*.

Pour insérer un nouvel élément de menu vide :

1 Placez le curseur sur un élément de menu.
2 Appuyez sur *Inser*.

Dans la barre de menu, les éléments de menu sont insérés à gauche de l'élément sélectionné et au-dessus de l'élément sélectionné dans la liste de menus.

Pour supprimer un élément de menu :

1 Placez le curseur sur l'élément de menu à supprimer.
2 Appuyez sur *Suppr*.

Remarque Vous ne pouvez pas supprimer l'emplacement par défaut qui apparaît en dessous du dernier élément ajouté à la liste de menus ou à côté du dernier élément de la barre de menu. Cet emplacement n'apparaît pas dans le menu à l'exécution.

Ajout de lignes de séparation

Les lignes de séparation insèrent une ligne entre deux éléments de menu. Vous pouvez utiliser les lignes de séparation pour matérialiser des groupes de commandes dans une liste de menus ou simplement pour réaliser une séparation visuelle dans une liste.

Pour faire de l'élément de menu une ligne de séparation, entrez un tiret (-) comme libellé.

Spécification de touches accélératrices et de raccourcis clavier

Les touches accélératrices permettent à l'utilisateur d'accéder à une commande de menu à partir du clavier en appuyant sur *Alt*+ la lettre appropriée indiquée dans le code en la faisant précéder du symbole &. La lettre suivant le symbole & apparaît soulignée dans le menu.

Delphi vérifie automatiquement si des touches accélératrices sont dupliquées et les ajuste à l'exécution. Ainsi, les menus conçus dynamiquement à l'exécution ne contiennent aucune touche accélératrice dupliquée et tous les éléments de menu disposent d'une touche accélératrice. Vous pouvez désactiver cette vérification automatique en attribuant à la propriété *AutoHotkeys* d'un élément de menu la valeur *maManual*.

Pour spécifier une touche accélératrice

- Ajoutez un & devant la lettre appropriée de l'intitulé.

  Ainsi, pour ajouter une commande de menu Enregistrer dont la lettre *E* sert de touche accélératrice, entrez `&Enregistrer`.

Les raccourcis clavier permettent à l'utilisateur d'effectuer l'action directement sans accéder au menu, simplement en appuyant sur la combinaison de touches du raccourci clavier.

Pour spécifier un raccourci clavier :

- Utilisez l'inspecteur d'objets pour saisir une valeur dans la propriété *ShortCut* ou sélectionnez une combinaison de touches dans la liste déroulante.

  Cette liste ne propose qu'un sous-ensemble de toutes les combinaisons utilisables.

Quand vous ajoutez un raccourci, il apparaît à l'exécution à côté de l'intitulé de l'élément de menu.

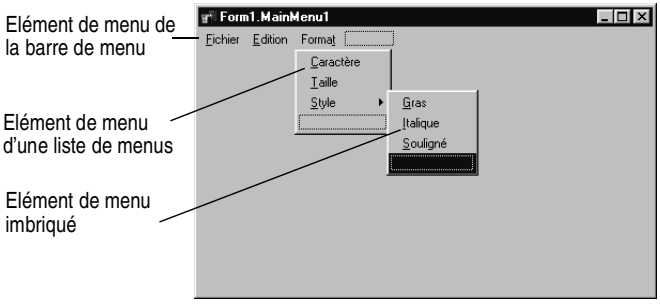
Attention A la différence des touches accélératrices, la présence de raccourcis clavier dupliqués n'est pas automatiquement vérifiée. Vous devez vous-même en assurer l'unicité.

Création de sous-menus

La plupart des menus d'application contiennent des listes déroulantes qui apparaissent à côté d'un élément de menu afin de proposer des commandes associées supplémentaires. De telles listes sont signalées par un pointeur à droite de l'élément de menu. Delphi gère les niveaux de sous-menus sans aucune limitation dans l'imbrication.

Une telle organisation de votre structure de menu permet d'économiser de la place verticalement. Néanmoins, pour optimiser la conception, il est préférable de ne pas utiliser plus de deux ou trois niveaux de menus dans la conception de votre interface. Dans le cas des menus surgissants, il est préférable de n'utiliser au maximum qu'un niveau de sous-menu.

Figure 6.7 Structure de menus imbriqués



Pour créer un sous-menu :

1. Sélectionnez l'élément de menu sous lequel vous voulez créer un sous-menu.
2. Appuyez sur *Ctrl*→ afin de créer le premier emplacement ou cliquez sur le bouton droit de la souris et choisissez Créer un sous-menu.
3. Entrez l'intitulé de l'élément de sous-menu ou déplacez un élément de menu existant sur cet emplacement.
4. Appuyez sur *Entrée* ou ↓ pour créer l'emplacement suivant.
5. Répétez les étapes 3 et 4 pour chaque élément du sous-menu.
6. Appuyez sur *Echap* pour revenir au niveau de menu précédent.

Création de sous-menus par déplacement de menus existants

Vous pouvez également créer un sous-menu en insérant un élément de la barre de menu (ou d'un modèle de menu) entre les éléments de menu d'une liste. Quand vous déplacez un menu dans une structure de menu existante, tous ses éléments de menu associés se déplacent, ce qui crée directement un sous-menu. Cela s'applique également aux sous-menus : le déplacement d'un élément de menu dans un sous-menu existant peut ainsi créer un niveau supplémentaire d'imbrication.

Déplacement d'éléments de menu

A la conception, vous pouvez déplacer les éléments de menu en utilisant le glisser-déplacer. Vous pouvez déplacer des éléments de menu dans la barre de menu, à un emplacement différent dans la liste de menus ou dans un autre composant menu.

La seule limitation à ces déplacements est de nature hiérarchique : vous ne pouvez pas déplacer un élément de menu de la barre de menu à l'intérieur de son propre menu ; de même, vous ne pouvez pas déplacer un élément de menu à l'intérieur de son propre sous-menu. Par contre, vous pouvez toujours déplacer un élément dans un menu *différent* quelle que soit sa position d'origine.

Quand vous faites glisser un élément, la forme du curseur change afin d'indiquer si vous pouvez déposer l'élément de menu à l'emplacement du

pointeur de la souris. Quand vous déplacez un élément de menu, tous les éléments de menu situés en dessous sont également déplacés.

Pour déplacer un élément de menu à l'intérieur de la barre de menu :

1. Faites glisser l'élément de menu jusqu'à ce que la pointe du curseur de déplacement désigne la nouvelle position.
2. Relâchez le bouton de la souris pour déposer l'élément de menu à la nouvelle position.

Pour déplacer un élément de menu dans une liste de menus :

1. Déplacez l'élément de menu le long de la barre de menu jusqu'à ce que le pointeur du curseur de déplacement désigne le nouveau menu.

   Cela force l'ouverture du menu, ce qui vous permet d'amener l'élément à son nouvel emplacement.
2. Faites glisser l'élément de menu dans la liste et relâchez le bouton de la souris pour déposer l'élément de menu à sa nouvelle position.

Ajout d'images à des éléments de menu

Des images peuvent aider les utilisateurs à naviguer dans les menus en associant des glyphes et des images à des actions d'élément de menu, comme les images des barres d'outils. Vous pouvez ajouter un par un des bitmaps uniques à des éléments de menu, ou organiser en liste les images de votre application et les ajouter à un menu au moyen de cette liste. Si, dans votre application, vous utilisez plusieurs bitmaps de même taille, il est préférable de les placer dans une liste d'images.

Pour ajouter une image unique à un menu ou à un élément de menu, définissez la propriété *Bitmap* de celui-ci pour qu'elle fasse référence au nom du bitmap que vous voulez utiliser.

Pour ajouter une image à un élément de menu à l'aide d'une liste d'images :

1. Déposez un composant *TMainMenu* ou *TPopupMenu* dans une fiche.
2. Déposez un objet *TImageList* dans la fiche.
3. Ouvrez l'éditeur de liste d'images en double-cliquant sur l'objet *TImageList*.
4. Choisissez Ajouter pour sélectionner le bitmap ou le groupe de bitmaps que vous voulez utiliser dans le menu. Choisissez OK.
5. Affectez l'objet liste d'images que vous venez de créer à la propriété *Images* du composant *TMainMenu* ou *TPopupMenu*.
6. Créez vos éléments de menu et vos sous-menus comme décrit plus haut.
7. Sélectionnez dans l'inspecteur d'objets l'élément de menu pour lequel vous voulez spécifier une image et affectez à sa propriété *ImageIndex* le numéro correspondant dans la liste d'images (la valeur par défaut de la propriété *ImageIndex* est -1 : pas d'image affichée).

Remarque Pour un affichage correct dans les menus, utilisez des images de 16 sur 16 pixels. Même si vous pouvez utiliser d'autres tailles pour les images placées dans les menus, il peut y avoir des problèmes d'alignement si vous utilisez des images plus grandes ou plus petites que 16 x 16 pixels.

Affichage du menu

Vous pouvez voir votre menu dans la fiche à la conception sans exécuter le code de votre programme. Les composants menu surgissant sont visibles dans la fiche à la conception, mais pas le menu surgissant lui-même. Utilisez le concepteur de menus pour visualiser un menu surgissant à la conception.

Pour visualiser le menu :

1. Si la fiche n'est pas visible, cliquez dans la fiche ou dans le menu Voir, choisissez la fiche que vous voulez voir.
2. Si la fiche contient plusieurs menus, sélectionnez le menu à visualiser dans la liste déroulante de la propriété *Menu* de la fiche.

   Le menu apparaît dans la fiche exactement tel qu'il apparaîtra à l'exécution du programme.

Edition des éléments de menu dans l'inspecteur d'objets

Cette section a décrit jusqu'à présent comment initialiser diverses propriétés des éléments de menu (par exemple, les propriétés *Name* ou *Caption*) en utilisant le concepteur de menus.

Cette section a également décrit comment initialiser certaines propriétés (par exemple, *ShortCut)* des éléments de menu directement dans l'inspecteur d'objets comme vous le faites pour les autres composants sélectionnés dans la fiche.

Quand vous modifiez un élément de menu en utilisant le concepteur de menus, ses propriétés restent affichées dans l'inspecteur d'objets. Vous pouvez donc faire passer la focalisation dans l'inspecteur d'objets et y poursuivre la modification de l'élément de menu. Vous pouvez également sélectionner l'élément de menu dans la liste des composants de l'inspecteur d'objets et modifier ses propriétés sans même ouvrir le concepteur de menus.

Pour fermer la fenêtre du concepteur de menus tout en poursuivant la modification des éléments de menu :

1. Faites passer la focalisation de la fenêtre du concepteur de menus à l'inspecteur d'objets en cliquant dans la page Propriétés de l'inspecteur d'objets.
2. Fermez normalement le concepteur de menus.

   La focalisation reste dans l'inspecteur d'objets où vous pouvez continuer à modifier les propriétés de l'élément de menu sélectionné. Pour éditer un autre élément de menu, sélectionnez-le dans la liste des composants.

Utilisation du menu contextuel du concepteur de menus

Le menu contextuel du concepteur de menus propose un accès rapide aux commandes les plus couramment utilisées du concepteur de menus et aux options des modèles de menu. (Pour davantage d'informations sur les modèles de menu, voir "Utilisation des modèles de menu" à la page 6-43.)

Pour afficher le menu contextuel, cliquez avec le bouton droit de la souris dans la fenêtre du concepteur de menus ou appuyez sur *Alt+F10* quand le curseur est dans la fenêtre du concepteur de menus.

Commandes du menu contextuel

Le tableau suivant résume les commandes du menu contextuel du concepteur de menus.

Tableau 6.5 Commandes du menu contextuel du concepteur de menus

Commande de menu	Action
Insérer	Insère un emplacement au-dessus ou à gauche du curseur.
Supprimer	Supprime l'élément de menu sélectionné (et tous ses éventuels sous-éléments).
Créer un sous-menu	Crée un emplacement à un niveau imbriqué et ajoute un pointeur à droite de l'élément de menu sélectionné.
Sélectionner un menu	Ouvre une liste des menus dans la fiche en cours. Double-cliquez sur un nom de menu pour l'ouvrir dans le concepteur de menus.
Enregistrer comme modèle	Ouvre la boîte de dialogue Enregistrement de modèle qui vous permet d'enregistrer un menu pour une réutilisation ultérieure.
Insérer depuis un modèle	Ouvre la boîte de dialogue Insertion de modèle qui vous permet de sélectionner le modèle à réutiliser.
Supprimer des modèles	Ouvre la boîte de dialogue Suppression de modèles qui vous permet de supprimer des modèles existants.
Insérer depuis une ressource	Ouvre la boîte de dialogue Insertion de menu depuis une ressource qui vous permet de choisir le fichier .mnu ouvert dans la fiche en cours.

Déplacement parmi les menus à la conception

Si vous concevez plusieurs menus pour votre fiche, vous pouvez utiliser le concepteur de menus ou l'inspecteur d'objets pour sélectionner un menu et passer d'un menu à l'autre.

Pour utiliser le menu contextuel afin de passer d'un menu de la fiche à un autre :

1 Cliquez avec le bouton droit de la souris dans le concepteur de menus et choisissez Sélectionnez un menu.

La boîte de dialogue Sélection de menu s'affiche.

Figure 6.8 Boîte de dialogue Sélection de menu

Cette boîte de dialogue énumère tous les menus associés à la fiche dont les menus sont ouverts dans le concepteur de menus.

2 Dans la liste de la boîte de dialogue Sélection de menu, choisissez le menu que vous voulez voir ou modifier.

Pour utiliser l'inspecteur d'objets afin de passer d'un menu de la fiche à un autre :

1 Sélectionnez la fiche dont vous voulez choisir un menu.

2 Dans la liste des composants, sélectionnez le menu que vous voulez modifier.

3 Dans la page Propriétés de l'inspecteur d'objets, sélectionnez la propriété *Items* de ce menu, puis cliquez sur le bouton points de suspension ou double-cliquez sur [Menu].

Utilisation des modèles de menu

Delphi propose plusieurs menus pré-conçus (des modèles de menu) qui contiennent des commandes d'utilisation courante. Vous pouvez utiliser ces menus dans vos applications sans les modifier (sauf pour ajouter le code) ou vous pouvez les utiliser comme point de départ en les personnalisant comme vous le feriez avec un menu que vous avez créé. Les modèles de menu ne contiennent pas de code de gestionnaire d'événement.

Les modèles de menu livrés avec Delphi sont stockés dans le sous-répertoire BIN de l'installation par défaut. Ces fichiers ont l'extension .DMT (extension des modèles de menu dans Delphi).

Vous pouvez enregistrer comme modèle tout menu que vous avez conçu dans le concepteur de menus. Après avoir enregistré un menu comme modèle, vous pouvez l'utiliser comme les autres modèles de menu. Si vous n'avez plus besoin d'un modèle de menu, vous pouvez le retirer de la liste.

Pour ajouter un modèle de menu à votre application :

1 Cliquez avec le bouton droit de la souris dans le concepteur de menus et choisissez Insérer depuis un modèle.

(S'il n'y a pas de modèle, l'option Insérer depuis un modèle apparaît estompée dans le menu contextuel.)

La boîte de dialogue Insertion de modèle est ouverte et affiche une liste des modèles de menu disponibles.

Figure 6.9 Boîte de dialogue Insertion de modèle des menus

2 Sélectionnez le modèle de menu que vous voulez insérer, puis appuyez sur *Entrée* ou choisissez OK.

Cela insère le menu dans votre fiche à l'emplacement du curseur. Si, par exemple, le curseur est positionné sur un élément de menu d'une liste, le modèle de menu est inséré en dessous de l'élément sélectionné. Si le curseur est dans la barre de menu, le modèle de menu est inséré à gauche du curseur.

Pour supprimer un modèle de menu :

1 Cliquez avec le bouton droit de la souris dans le concepteur de menus et choisissez Supprimer des modèles.

(S'il n'y a pas de modèles, l'option Supprimer des modèles apparaît estompée dans le menu contextuel.)

La boîte de dialogue Suppression de modèles est ouverte et affiche une liste des modèles existants.

2 Sélectionnez le modèle de menu à supprimer et appuyez sur *Suppr*.

Delphi supprime le modèle dans la liste des modèles et sur le disque dur.

Enregistrement d'un menu comme modèle

Il est possible d'enregistrer comme modèle tout menu que vous avez conçu afin de pouvoir le réutiliser. Vous pouvez employer les modèles de menu pour donner un aspect homogène à vos applications ou les utiliser comme points de départ que vous personnalisez.

Les modèles de menu que vous enregistrez sont stockés dans le sous-répertoire BIN dans des fichiers .DMT.

Pour enregistrer un menu comme modèle

1 Concevez le menu que vous voulez pouvoir réutiliser.

Ce menu peut contenir de nombreux éléments, commandes et sous-menus : tout dans le contenu de la fenêtre active du concepteur de menus est enregistré comme un seul menu réutilisable.

2 Cliquez avec le bouton droit de la souris dans le concepteur de menus et choisissez Enregistrer comme modèle.

La boîte de dialogue Enregistrement de modèle s'ouvre.

Figure 6.10 Boîte de dialogue Enregistrement de modèle pour les menus

3 Dans la boîte de saisie Description du modèle, entrez une brève description de ce menu, puis choisissez OK.

La boîte de dialogue Enregistrement du modèle se ferme en enregistrant votre menu et vous revenez dans le concepteur de menus.

Remarque La description que vous saisissez n'est affichée que dans les boîtes de dialogue Enregistrement de modèle, Insertion de modèle et Suppression de modèles. Elle n'a aucun rapport avec les propriétés *Name* ou *Caption* du menu.

Conventions de nom pour les éléments et les gestionnaires d'événements des modèles de menu

Quand vous enregistrez un menu comme modèle, Delphi n'enregistre pas sa propriété *Name* car chaque menu doit disposer d'un nom unique dans la portée de son propriétaire (la fiche). Cependant, quand vous insérez le menu comme modèle dans une nouvelle fiche en utilisant le concepteur de menus, Delphi génère alors de nouveaux noms pour lui et tous ses éléments.

Par exemple, vous enregistrez un menu Fichier comme modèle Dans le menu original, vous le nommez *MonFichier*. Si vous l'insérez comme modèle dans un nouveau menu, Delphi le nomme *Fichier1*. Si vous l'insérez dans un menu ayant déjà un élément de menu nommé *Fichier1*, Delphi le nomme *Fichier2*.

Delphi n'enregistre aucun gestionnaire d'événement *OnClick* associé au menu enregistré comme modèle car il n'y a pas moyen de savoir si le code est applicable dans la nouvelle fiche. Quand vous générez un nouveau gestionnaire d'événement pour un élément du modèle de menu, Delphi génère également le nom du gestionnaire d'événement.

Il est simple d'associer des éléments de menu à des gestionnaires d'événements *OnClick* existants de la fiche. Pour plus d'informations, voir "Association d'événements de menu à des gestionnaires d'événements" à la page 3-31.

Manipulation d'éléments de menu à l'exécution

A l'exécution, il est parfois nécessaire d'ajouter à une structure de menu existante des éléments de menu ; cela permet de proposer davantage d'informations ou d'options à l'utilisateur. Il est possible d'insérer un élément de menu en utilisant les méthodes *Add* ou *Insert* de l'élément de menu. Vous pouvez également masquer ou afficher les éléments d'un menu en jouant sur leur propriété *Visible*. La propriété *Visible* détermine si l'élément de menu est affiché dans le menu. Pour estomper un élément de menu sans le masquer, utilisez la propriété *Enabled*.

Pour des exemples utilisant les propriétés *Visible* et *Enabled* des éléments de menu, voir "Désactivation des éléments de menu" à la page 7-11.

Dans des applications MDI ou OLE, vous pouvez également fusionner des éléments de menu dans une barre de menu existante. Les sections suivantes décrivent ce processus plus en détail.

Fusion de menus

For Dans les applications MDI, comme l'application exemple éditeur de texte ou dans les applications client OLE, le menu principal de votre application doit être capable d'intégrer les éléments de menu d'une autre fiche ou d'un objet serveur OLE. Ce processus s'appelle la *fusion de menus*. Notez que la technologie OLE est limitée aux applications Windows et n'est pas utilisable en programmation multiplate-forme.

Pour préparer des menus à la fusion, vous devez spécifier les valeurs de deux propriétés :

- *Menu*, une propriété de la fiche.
- *GroupIndex*, une propriété des éléments du menu.

Spécification du menu actif : propriété Menu

La propriété *Menu* spécifie le menu actif de la fiche. Les opérations de fusion de menu portent uniquement sur le menu actif. Si la fiche contient plusieurs composants menu, vous pouvez changer le menu actif à l'exécution en modifiant la valeur de la propriété *Menu* dans le code. Par exemple,

```
Form1.Menu := SecondMenu;
```

Ordre des éléments de menu fusionnés : propriété GroupIndex

La propriété *GroupIndex* détermine l'ordre dans lequel les éléments des menus fusionnés apparaissent dans la barre de menu partagée. La fusion d'éléments de

menu peut remplacer des éléments existants de la barre de menu principale ou rajouter des éléments à celle-ci.

La valeur par défaut de *GroupIndex* est 0. Plusieurs règles s'appliquent à la spécification d'une valeur pour la propriété *GroupIndex* :

- Les nombres les plus bas apparaissent en premier (plus à gauche) dans le menu.

  Par exemple, affectez la valeur 0 (zéro) à la propriété *GroupIndex* d'un menu qui doit apparaître tout à gauche, comme le menu Fichier. De même, spécifiez une valeur élevée (elle n'a pas besoin d'être consécutive) à un menu qui doit apparaître tout à droite (comme le menu Aide).

- Pour remplacer des éléments du menu principal, attribuez la même valeur à la propriété *GroupIndex* des éléments du menu enfant.

  Cela peut s'appliquer à des groupes d'éléments ou à un seul élément. Si, par exemple, votre fiche principale contient un élément de menu Edition dont la propriété *GroupIndex* vaut 1, vous pouvez le remplacer par un ou plusieurs éléments du menu de la fiche enfant en attribuant également la valeur 1 à leur propriété *GroupIndex*.

  Si plusieurs éléments du menu enfant ont la même valeur pour *GroupIndex,* leur ordre n'est pas modifié quand ils sont fusionnés au menu principal.

- Pour insérer des éléments sans remplacer des éléments du menu principal, laissez des intervalles numériques entre les éléments du menu principal et intercalez les numéros de la fiche enfant.

  Vous pouvez par exemple, numéroter les éléments du menu principal 0 et 5, et insérer les éléments du menu enfant en les numérotant 1, 2, 3 et 4.

Importation de fichiers ressource

Delphi peut utiliser des menus conçus avec d'autres applications dans la mesure où ils utilisent le format standard de fichier ressource Windows (.RC). Vous pouvez importer ces menus directement dans votre projet Delphi, ce qui vous évite d'avoir à redéfinir des menus que vous avez conçu par ailleurs.

Pour charger un fichier menu .RC existant :

1. Dans le concepteur de menus, placez le curseur à l'endroit où le menu doit apparaître.

   Le menu importé peut faire partie du menu que vous concevez ou constituer la totalité d'un menu par lui-même.

2. Cliquez avec le bouton droit de la souris et choisissez Insérer depuis la ressource.

   La boîte de dialogue Insertion de menu depuis une ressource s'ouvre.

3. Dans la boîte de dialogue, sélectionnez le fichier ressource à charger, puis choisissez OK.

Le menu apparaît dans la fenêtre du concepteur de menus.

Remarque Si votre ficher ressource contient plusieurs menus, vous devez enregistrer chacun de ses menus dans un fichier ressource séparé avant de pouvoir importer.

Conception de barres d'outils et de barres multiples

Une *barre d'outils* est un volet, généralement placé en haut d'une fiche (sous la barre de menu) qui contient des boutons et d'autres contrôles. Une *barre multiple* est une sorte de barre d'outils qui affiche des contrôles dans des bandes déplaçables et redimensionnables. Si plusieurs volets sont alignés sur le haut de la fiche, ils s'empilent verticalement dans l'ordre de leur ajout.

Remarque Les barres multiples ne peuvent pas être utilisées dans CLX pour les applications multiplates-formes.

Vous pouvez placer toutes sortes de contrôles dans une barre d'outils. Outre les boutons, vous pouvez y placer des grilles de couleur, des barres de défilement, des libellés, etc.

Vous pouvez ajouter une barre d'outils à une fiche de plusieurs façons :

- Placez un composant volet (*TPanel*) dans la fiche et ajoutez-y des contrôles, en général des turboboutons.
- Utilisez un composant barre d'outils (*TToolBar*) à la place de *TPanel* et ajoutez-lui des contrôles. *TToolBar* gère les boutons et les autres contrôles en les disposant en lignes et en ajustant automatiquement leur taille et leur position. Si vous utilisez des contrôles bouton outil (*TToolButton*) dans la barre d'outils, *TToolBar* permet simplement de grouper les boutons de manière fonctionnelle et propose d'autres options d'affichage.
- Utilisez un composant barre multiple (*TCoolBar*) et ajoutez-lui des contrôles. La barre multiple affiche des contrôles dans des bandes qui peuvent être déplacées et redimensionnées de manière indépendante.

La méthode à employer pour implémenter une barre d'outils dépend de votre application. Le composant volet présente l'avantage de vous donner une maîtrise totale de l'aspect de la barre d'outils.

Si vous utilisez des composants barre d'outils ou bande multiple, vous êtes certain que votre application a bien le style d'une application Windows, car vous utilisez dans ce cas des contrôles natifs de Windows. Si ces contrôles du système d'exploitation changent à l'avenir, votre application changera également. Par ailleurs, comme les composants barre d'outils et barre multiple sont fondés sur des composants standard Windows, votre application nécessite la présence du fichier COMCTL32.DLL. Les barres d'outils et les barres multiples ne sont pas autorisés dans les applications WinNT 3.51.

Les sections suivantes expliquent comment :

- Ajouter une barre d'outils et les contrôles turbobouton correspondants en utilisant le composant volet

- Ajouter une barre d'outils et les contrôles bouton outil correspondants en utilisant le composant barre d'outils
- Ajouter un composant barre multiple
- Répondre aux clics
- Ajouter des barres d'outils et des barres multiples masquées
- Afficher et masquer des barres d'outils et des barres multiples

Ajout d'une barre d'outils en utilisant un composant volet

Pour ajouter à une fiche une barre d'outils en utilisant un composant volet :

1. Ajoutez à la fiche un composant volet (à partir de la page Standard de la palette de composants).
2. Affectez la valeur *alTop* à la propriété *Align* du volet. Quand il est aligné sur le haut de la fiche, le volet conserve sa hauteur mais ajuste sa largeur pour occuper toute la largeur de la zone client de la fiche, et ce même si la fenêtre change de taille.
3. Ajoutez des turboboutons ou d'autres contrôles dans le volet.

Les turboboutons sont conçus pour fonctionner dans des volets barre d'outils. Généralement, un turbobouton n'a pas d'intitulé mais seulement une petite image (appelée un *glyphe*) qui représente la fonction du bouton.

Les turboboutons ont trois modes de fonctionnement. Ils peuvent :

- Se comporter comme des boutons poussoir normaux.
- Se comporter comme des bascules.
- Se comporter comme un ensemble de boutons radio.

Pour implémenter des turboboutons dans des barres d'outils, vous pouvez :

- Ajouter un turbobouton dans un volet barre d'outils
- Affecter un glyphe au turbobouton
- Définir l'état initial du turbobouton
- Créer un groupe de turboboutons
- Utiliser des boutons bascule

Ajout d'un turbobouton à un volet

Pour ajouter un turbobouton à un volet barre d'outils, placez dans le volet un composant turbobouton (à partir de la page Supplément de la palette de composants.

C'est alors le volet et non la fiche qui est le "propriétaire" du turbobouton, donc déplacer ou masquer le volet déplace ou masque également le turbobouton.

La hauteur par défaut d'un volet est 41 et la hauteur par défaut d'un turbobouton est 25. Si vous affectez la valeur 8 à la propriété *Top* de chaque bouton, ils sont centrés verticalement. Le paramétrage par défaut de la grille aligne verticalement le turbobouton sur cette position.

Spécification du glyphe d'un turbobouton

Chaque turbobouton a besoin d'une image appelée un *glyphe* afin d'indiquer à l'utilisateur la fonction du bouton. Si vous ne spécifiez qu'une seule image pour le bouton, le bouton manipule l'image afin d'indiquer si le bouton est enfoncé, relâché, sélectionné ou désactivé. Vous pouvez également spécifier des images distinctes spécifiques à chaque état.

Normalement, les glyphes sont affectés à un turbobouton à la conception mais il est possible d'affecter d'autres glyphes à l'exécution.

Pour affecter un glyphe à un turbobouton à la conception :

1. Sélectionnez le turbobouton.
2. Dans l'inspecteur d'objets, sélectionnez la propriété *Glyph*.
3. Double-cliquez dans la colonne des valeurs à côté de *Glyph* pour afficher l'éditeur d'images et sélectionner le bitmap souhaité.

Définition de l'état initial d'un turbobouton

C'est l'aspect visuel d'un turbobouton qui donne à l'utilisateur des indications sur sa fonction et son état. N'ayant pas d'intitulé, il est indispensable d'utiliser des indications visuelles pour aider l'utilisateur.

Le tableau suivant décrit comment modifier l'aspect d'un turbobouton :

Tableau 6.6 Paramétrage de l'aspect d'un turbobouton

Pour que le turbobouton :	Initialisez :
Apparaisse enfoncé	Sa propriété *GroupIndex* à une valeur non nulle et sa propriété *Down* à *True*.
Apparaisse désactivé	Sa propriété *Enabled* à *False*.
Dispose d'une marge gauche	Sa propriété *Indent* à une valeur supérieure à 0.

Si, par exemple, votre application propose un outil de dessin activé par défaut, vérifiez que le bouton correspondant de la barre d'outils est enfoncé au démarrage de l'application. Pour ce faire, affectez une valeur non nulle à sa propriété *GroupIndex* et la valeur *True* à sa propriété *Down*.

Création d'un groupe de turboboutons

Une série de turboboutons représente souvent un ensemble de choix mutuellement exclusifs. Dans ce cas, vous devez associer les boutons dans un groupe, afin que l'enfoncement d'un bouton fasse remonter le bouton précédemment enfoncé du groupe.

Pour associer un nombre quelconque de turboboutons dans un groupe, affectez la même valeur à la propriété *GroupIndex* de chacun de ces turboboutons.

Le moyen le plus simple de procéder consiste à sélectionner tous les boutons à grouper puis à spécifier une même valeur pour leur propriété *GroupIndex*.

Utilisation de boutons bascule

Dans certains cas, vous voulez pouvoir cliquer sur un bouton déjà enfoncé d'un groupe afin de le faire remonter, ce qui laisse le groupe sans aucun bouton enfoncé. Un tel bouton est appelé une *bascule*. Utilisez la propriété *AllowAllUp* pour créer un groupe de boutons qui se comporte ainsi : cliquez une fois; il est enfoncé, cliquez à nouveau, il remonte.

Pour qu'un groupe de boutons radio se comporte comme une bascule, affectez à sa propriété *AllowAllUp* la valeur *True*.

L'affectation de la valeur *True* à la propriété *AllowAllUp* d'un des turboboutons du groupe l'affecte à tous ceux du groupe. Cela permet au groupe de se comporter comme un groupe normal, un seul bouton étant sélectionné à la fois mais, au même moment, tous les boutons peuvent être à l'état relâché.

Ajout d'une barre d'outils en utilisant le composant barre d'outils

Le composant barre d'outils (*TToolBar*) propose des caractéristiques de gestion des boutons et de l'affichage dont ne dispose pas le composant volet. Pour ajouter une barre d'outils à une fiche en utilisant le composant barre d'outils :

1. Ajoutez à la fiche un composant barre d'outils (à partir de la page Win32 de la palette de composants). La barre d'outils s'aligne automatiquement en haut de la fiche.
2. Ajoutez des boutons outil ou d'autres contrôles à la barre.

Les boutons outil sont conçus pour fonctionner dans des composants barre d'outils. Comme les turboboutons, les boutons outil peuvent :

- Se comporter comme des boutons poussoirs normaux.
- Se comporter comme des bascules.
- Se comporter comme un ensemble de boutons radio.

Pour implémenter des boutons outil dans une barre d'outils, vous pouvez :

- Ajouter un bouton outil
- Affecter des images à un bouton outil
- Définir l'aspect et les conditions initiales du bouton outil
- Créer un groupe de boutons outil
- Utiliser des boutons outil bascule

Ajout d'un bouton outil

Pour ajouter un bouton outil dans une barre d'outils, cliquez avec le bouton droit de la souris dans la barre d'outils et choisissez Nouveau bouton.

La barre d'outils est le "propriétaire" du bouton outil : déplacer ou masquer la barre d'outils déplace et masque également le bouton. De plus, tous les boutons outil de la barre d'outils ont automatiquement la même largeur et la même hauteur. Vous pouvez déposer dans la barre d'outils d'autres contrôles de la palette de composants, ils ont automatiquement une hauteur homogène. De plus,

les contrôles passent à la ligne et commencent une nouvelle ligne quand ils ne tiennent pas horizontalement sur une seule ligne de la barre d'outils.

Affectation d'images à des boutons outil

Chaque bouton outil dispose de la propriété *ImageIndex* qui détermine l'image apparaissant dedans à l'exécution. Si vous ne fournissez qu'une seule image au bouton outil, le bouton manipule cette image pour indiquer si le bouton est désactivé. Pour affecter une image à un bouton outil à la conception :

1. Sélectionnez la barre d'outils dans laquelle le bouton apparaît.
2. Dans l'inspecteur d'objet, attribuez un objet *TImageList* à la propriété *Images* de la barre d'outils. Une liste d'images est une collection d'icônes ou de bitmaps de même taille.
3. Sélectionnez un bouton outil.
4. Dans l'inspecteur d'objets, affectez à la propriété *ImageIndex* du bouton outil une valeur entière correspondant à l'image de la liste d'images qui doit être affectée au bouton.

Vous pouvez également spécifier des images distinctes apparaissant dans les boutons outil quand ils sont désactivés ou sous le pointeur de la souris. Pour ce faire, affectez des listes d'images distinctes aux propriétés *DisabledImages* et *HotImages* de la barre d'outils.

Définition de l'aspect et de l'état initial d'un bouton outil

Le tableau suivant indique comment modifier l'aspect des boutons d'une barre d'outils :

Tableau 6.7 Paramétrage de l'aspect des boutons d'une barre d'outils

Pour que le bouton outil :	Initialisez :
Apparaisse enfoncé	(Sur le bouton outil) sa propriété *Style* à *tbsCheck* et sa propriété *Down* à *True*.
Apparaisse désactivé	Sa propriété *Enabled* à *False*.
Dispose d'une marge gauche	Sa propriété *Indent* à une valeur supérieure à 0.
Semble avoir une bordure "pop-up", ce qui donne à la barre d'outils un aspect transparent	Sa propriété *Flat* à *True*

Remarque L'utilisation de la propriété *Flat* de *TToolBar* nécessite une version 4.70 ou ultérieure de COMCTL32.DLL.

Pour forcer le passage à la ligne après un bouton outil spécifique, sélectionnez le bouton outil devant apparaître en dernier sur la ligne et affectez la valeur *True* à sa propriété *Wrap*.

Pour désactiver le passage à la ligne automatique dans la barre d'outils, affectez la valeur *False* à la propriété *Wrapable* de la barre d'outils.

Création de groupes de boutons outil

Pour créer un groupe de boutons outils, sélectionnez les boutons à associer et affectez la valeur *tbsCheck* à leur propriété *Style* et la valeur *True* à leur propriété *Grouped*. La sélection d'un bouton outil du groupe désélectionne le choix précédent dans le groupe de boutons, ce qui permet de représenter un ensemble de choix mutuellement exclusifs.

Toute séquence non interrompue de boutons outils adjacents dont la propriété *Style* a la valeur *tbsCheck* et la propriété *Grouped* la valeur *True* forme un même groupe. Pour interrompre un groupe de boutons outils, séparez les boutons avec :

- Un bouton outil dont la propriété *Grouped* a la valeur *False*.
- Un bouton outil dont la propriété *Style* n'a pas la valeur *tbsCheck*. Pour créer des espaces ou des séparateurs dans la barre d'outils, ajoutez un bouton outil de *Style tbsSeparator* ou *tbsDivider*.
- Un contrôle d'un type autre que bouton outil.

Utilisation de boutons outil bascule

Utilisez la propriété *AllowAllUp* pour créer un groupe de boutons outils se comportant comme une bascule : cliquez une fois pour enfoncer le bouton et une seconde fois pour le faire remonter. Pour qu'un groupe de boutons outils se comporte comme une bascule, affectez la valeur *True* à sa propriété *AllowAllUp*.

Comme pour les turboboutons, l'affectation de la valeur *True* à la propriété *AllowAllUp* d'un des boutons du groupe affecte automatiquement la même valeur à tous les boutons du groupe.

Ajout d'un composant barre multiple

Remarque Le composant *TCoolBar* nécessite une version 4.70 ou ultérieure de COMCTL32.DLL et n'est pas disponible dans CLX.

Le composant barre multiple (*TCoolBar*) — également appelé *multibarre* — affiche des contrôles fenêtrés dans des bandes redimensionnables qui peuvent se déplacer indépendamment les unes des autres. L'utilisateur peut positionner les bandes faisant glisser des poignées de redimensionnement placées sur le côté de chaque bande.

Pour ajouter une barre multiple à une fiche dans une application Windows,

1. Ajoutez à la fiche un composant barre multiple (à partir de la page Win32 de la palette de composants). La barre multiple s'aligne automatiquement en haut de la fiche.
2. Ajoutez à la barre des contrôles fenêtrés de la palette de composants.

Seuls les composants de la VCL dérivant de *TWinControl* sont des contrôles fenêtrés. Vous pouvez ajouter à une barre multiple des contrôles graphiques

(comme les libellés ou les turboboutons), mais ils n'apparaissent pas dans des bandes distinctes.

Définition de l'aspect de la barre multiple

Le composant barre multiple dispose de plusieurs options de configuration. Le tableau suivant indique comment modifier l'aspect des boutons d'une barre multiple :

Tableau 6.8 Paramétrage de l'aspect des boutons d'une barre multiple

Pour que la barre multiple :	Initialisez :
Se redimensionne automatiquement pour s'adapter aux bandes qu'elle contient.	Sa propriété *AutoSize* à *True*.
Dispose de bandes d'une hauteur uniforme.	Sa propriété *FixedSize* à *True*.
Soit orientée à la verticale et pas à l'horizontale.	Sa propriété *Vertical* à *True*. Cela change l'effet de la propriété *FixedSize*.
Empêcher l'affichage à l'exécution de la propriété *Text* des bandes.	Sa propriété *ShowText* à *False*. Chaque bande d'une barre multiple a sa propre propriété *Text*.
Retirer la bordure autour de la barre.	Sa propriété *BandBorderStyle* à *bsNone*.
Empêcher les utilisateurs de modifier l'ordre des bandes à l'exécution. L'utilisateur peut toujours déplacer et redimensionner les bandes.	Sa propriété *FixedOrder* à *True*.
Créer une image de fond pour la barre multiple.	Sa propriété *Bitmap* avec un objet *TBitmap*.
Choisir une liste d'images apparaissant à gauche des bandes	Sa propriété *Images* avec un objet *TImageList*.

Pour affecter individuellement des images aux bandes, sélectionnez la barre multiple et, dans l'inspecteur d'objets, double-cliquez sur sa propriété *Bands*. Sélectionnez une bande et affectez une valeur à sa propriété *ImageIndex*.

Réponse aux clics

Quand l'utilisateur clique sur un contrôle, par exemple un bouton d'une barre d'outils, l'application génère un événement *OnClick* auquel vous pouvez répondre avec un gestionnaire d'événement. Etant donné que *OnClick* est l'événement par défaut des boutons, vous pouvez générer un gestionnaire squelette pour l'événement en double-cliquant sur le bouton à la conception. Pour plus d'informations, voir "Utilisation des événements et des gestionnaires d'événements" à la page 3-28 et "Génération du gestionnaire de l'événement par défaut d'un composant" à la page 3-29.

Affectation d'un menu à un bouton outil

Si vous utilisez une barre d'outils (*TToolBar*) contenant des boutons outils (*TToolButton*), vous pouvez associer un menu à un bouton spécifique :

1. Sélectionnez le bouton outil.
2. Dans l'inspecteur d'objets, affectez un menu surgissant (*TPopupMenu*) à la propriété *DropDownMenu* du bouton outil.

Si la propriété *AutoPopup* du menu a la valeur *True*, le menu apparaît automatiquement quand le bouton est enfoncé.

Ajout de barres d'outils masquées

Les barres d'outils n'ont pas besoin d'être visibles tout le temps. En fait, il est souvent commode de disposer de plusieurs barres d'outils, mais de n'afficher que celles dont l'utilisateur veut disposer. Souvent, les fiches que vous créez contiennent plusieurs barres d'outils, mais en masquent certaines ou même toutes.

Pour créer une barre d'outils masquée :

1. Ajoutez à la fiche un composant barre d'outils, barre multiple ou volet.
2. Affectez la valeur *False* à la propriété *Visible* du composant.

Bien que la barre d'outils reste visible à la conception afin que vous puissiez la modifier, elle reste cachée à l'exécution tant que l'application ne la rend pas explicitement visible.

Masquage et affichage d'une barre d'outils

Fréquemment, une application dispose de plusieurs barres d'outils mais vous ne voulez pas encombrer l'écran en les affichant toutes à la fois. Vous pouvez laisser l'utilisateur décider s'il veut afficher les barres d'outils. Comme tous les composants, les barres d'outils peuvent être masquées et affichées quand c'est nécessaire à l'exécution.

Pour masquer ou afficher une barre d'outils à l'exécution, affectez à sa propriété *Visible*, respectivement, la valeur *False* or *True*. Généralement vous faites ceci en réponse à un événement utilisateur particulier ou à un changement du mode de fonctionnement de l'application. Pour ce faire, chaque barre d'outils dispose généralement d'un bouton de fermeture. Quand l'utilisateur clique sur ce bouton, l'application masque la barre d'outils correspondante.

Vous pouvez également proposer un système pour inverser l'état de la barre d'outils. Dans l'exemple suivant, la visibilité d'une barre d'outils de crayons est inversée par un bouton de la barre d'outils principale. Comme chaque clic de la souris enfonce ou libère le bouton, un gestionnaire d'événement *OnClick* peut afficher ou masquer la barre d'outils des crayons selon que le bouton est relâché ou enfoncé.

```
procedure TForm1.PenButtonClick(Sender: TObject);
begin
  PenBar.Visible := PenButton.Down;
end;
```

Programmes exemple

Pour des exemples de applications Windows utilisant les actions et les listes d'actions, voir Demos\RichEdit. De plus, les démos de l'expert Application (Fichier|page Nouveau projet), Application MDI, Application SDI et Application Logo Winx peuvent utiliser les objets action et liste d'actions. Pour des exemples d'applications multiplates-formes, reportez-vous à Demos\CLX.

Chapitre

7

Manipulation des contrôles

Les contrôles sont des composants visuels avec lesquels l'utilisateur peut interagir à l'exécution. Ce chapitre décrit un ensemble de fonctionnalités communes à de nombreux contrôles.

Implémentation du glisser-déplacer dans les contrôles

Le glisser-déplacer est souvent une façon pratique de manipuler des objets. Les utilisateurs peuvent ainsi faire glisser des contrôles entiers, ou bien extraire des éléments de contrôles (tels que des boîtes liste ou des vues arborescentes) en les faisant glisser sur d'autres contrôles.

- Début de l'opération glisser-déplacer
- Acceptation des éléments à déplacer
- Déplacement des éléments
- Fin de l'opération glisser-déplacer
- Personnalisation du glisser-déplacer avec un objet déplacement
- Changement du pointeur de la souris

Début de l'opération glisser-déplacer

Chaque contrôle possède une propriété appelée *DragMode* qui détermine la façon dont les opérations glisser sont démarrées. Si la propriété *DragMode* est à *dmAutomatic*, l'opération glisser commence automatiquement quand l'utilisateur clique sur le bouton de la souris alors que le curseur se trouve au-dessus d'un contrôle. Le plus souvent, vous donnerez à *DragMode* la valeur *dmManual* (la valeur par défaut) et lancerez l'opération glisser en gérant les événements bouton de souris enfoncé.

Pour faire glisser un contrôle manuellement, appelez la méthode *BeginDrag* du contrôle. *BeginDrag* requiert un paramètre booléen appelé *Immediate* et, de manière facultative, un paramètre entier appelé *Threshold*. Si vous transmettez

True pour *Immediate*, l'opération glisser commence immédiatement. Si vous transmettez *False*, l'opération glisser ne commence pas avant que l'utilisateur ne déplace la souris du nombre de pixels spécifié par *Threshold*. L'appel de

```
BeginDrag False)
```

permet au contrôle d'accepter les clics de la souris sans lancer une opération glisser.

Vous pouvez imposer des conditions pour commencer l'opération glisser, par exemple vérifier le bouton de souris enfoncé par l'utilisateur, en testant les paramètres du gestionnaire de l'événement bouton de souris enfoncé, avant l'appel à *BeginDrag*. Le code qui suit, par exemple, gère l'événement bouton de souris enfoncé dans une boîte liste de fichiers en ne lançant l'opération glisser que si le bouton gauche de la souris a été enfoncé.

```
procedure TFMForm.FileListBox1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then  { ne glisser que si le bouton gauche est enfoncé }
    with Sender as TFileListBox do  { traite Sender comme TFileListBox }
      begin
        if ItemAtPos(Point(X, Y), True) >= 0 then  { y a-t-il un élément ici ? }
          BeginDrag(False);  { si oui, le faire glisser }
    end;
end;
```

Acceptation des éléments à déplacer

Quand l'utilisateur fait glisser quelque chose sur un contrôle, celui-ci reçoit un événement *OnDragOver*. Il doit alors indiquer s'il peut accepter l'élément dans le cas où l'utilisateur le lâcherait à cet emplacement. L'aspect du curseur change pour indiquer si le contrôle peut accepter l'élément que l'utilisateur fait glisser. Pour accepter les éléments que l'utilisateur fait glisser sur un contrôle, attachez un gestionnaire à l'événement *OnDragOver* du contrôle.

L'événement "glisser-dessus" a un paramètre appelé *Accept* que le gestionnaire d'événement peut définir à *True* pour indiquer qu'il accepte l'élément. Si *Accept* vaut *True*, l'application renvoie un événement "glisser-déplacer" au contrôle.

L'événement "glisser-dessus" présente d'autres paramètres, dont la source de l'opération glisser et l'emplacement actuel du curseur de la souris, que le gestionnaire d'événement peut utiliser pour déterminer s'il doit accepter le déplacement. Dans l'exemple ci-dessous, une arborescence de répertoires accepte les objets déplacés seulement s'ils viennent d'une boîte liste de fichiers.

```
procedure TFMForm.DirectoryOutline1DragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if Source is TFileListBox then
    Accept := True
  else
    Accept := False;
end;
```

Déplacement des éléments

Si un contrôle indique qu'il peut accepter un élément déplacé, il doit le traiter s'il est effectivement lâché. Pour gérer les éléments lâchés, attachez un gestionnaire à l'événement *OnDragDrop* du contrôle qui accepte l'opération lâcher. Comme l'événement "glisser-dessus", l'événement "glisser-déplacer" indique la source de l'élément déplacé et les coordonnées du curseur de la souris lorsqu'il est au-dessus du contrôle acceptant l'élément. Le dernier paramètre vous permet de contrôler le chemin emprunté par un élément au cours de l'opération glisser ; vous pouvez, par exemple, utiliser ces informations pour modifier la couleur affichée par les composants au cours de leur transfert.

Dans l'exemple suivant, une arborescence de répertoires, qui accepte les éléments déplacés depuis une boîte liste de fichiers, répond en déplaçant les fichiers vers le répertoire sur lequel ils sont lâchés :

```
procedure TFMForm.DirectoryOutline1DragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  if Source is TFileListBox then
    with DirectoryOutline1 do
      ConfirmChange('Move', FileListBox1.FileName, Items[GetItem(X, Y)].FullPath);
end;
```

Fin de l'opération glisser-déplacer

Une opération glisser se termine lorsque l'élément est déplacé avec succès ou qu'il est relâché au-dessus d'un contrôle qui ne peut pas l'acccepter. A ce stade un événement "fin-glisser" est envoyé au contrôle à partir duquel l'élément a été déplacé. Pour permettre à un contrôle de répondre quand des éléments en sont extraits, attachez un gestionnaire à l'événement *OnEndDrag* du contrôle.

Le paramètre le plus important dans un événement *OnEndDrag* est appelé *Target*, il indique quel contrôle, le cas échéant, accepte l'élément déplacé. Si *Target* est **nil**, cela signifie qu'aucun contrôle ne l'accepte. L'événement *OnEndDrag* comprend aussi les coordonnées du contrôle de réception.

Dans cet exemple, une boîte liste de fichiers gère un événement "fin-glisser" en mettant à jour sa liste de fichiers.

```
procedure TFMForm.FileListBox1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
  if Target <> nil then FileListBox1.Update;
end;
```

Personnalisation du glisser-déplacer avec un objet déplacement

Vous pouvez utiliser un descendant de *TDragObject* pour personnaliser le comportement glisser-déplacer d'un objet. Les événements "glisser-dessus" et "glisser-déplacer" standard indiquent la source de l'élément glissé et les coordonnées du curseur de souris au-dessus du contrôle qui l'accepte. Pour

obtenir des informations supplémentaires sur l'état en cours, dérivez un objet glissé de *TDragObject* ou *TDragObjectEx* et surchargez ses méthodes virtuelles. L'objet glissé doit être créé dans l'événement *OnStartDrag*.

Normalement, le paramètre source des événements "glisser-dessus" et "glisser-déplacer" est le contrôle qui commence l'opération glisser. Si plusieurs sortes de contrôles peuvent commencer une opération impliquant le même type de données, la source doit gérer chaque sorte de contrôle. Lorsque vous utilisez un descendant de *TDragObject*, toutefois, la source est l'objet glissé lui-même ; si chaque contrôle crée le même type d'objet glissé dans son événement *OnStartDrag*, la cible doit gérer uniquement une sorte d'objet. Les événements "glisser-dessus" et "glisser-déplacer" peuvent indiquer si la source est un objet glissé, en opposition au contrôle, en appelant la fonction *IsDragObject*.

Les descendants de *TDragObjectEx* sont libérés automatiquement alors que les descendants de *TDragObject* ne le sont pas. Si vous avez des descendants de *TDragObject* qui ne sont pas explicitement libérés, vous pouvez les modifier de façon à ce qu'ils dérivent de *TDragObjectEx* au lieu de surveiller les perte de mémoire.

Les objets glissés vous permettent de déplacer des éléments entre une fiche implémentée dans le fichier exécutable principal de l'application et une fiche implémentée en utilisant une DLL, ou entre des fiches implémentées en utilisant différentes DLL.

Changement du pointeur de la souris

Il est possible de personnaliser l'aspect du pointeur de la souris lors d'opérations glisser en définissant la propriété *DragCursor* du composant source (VCL seulement).

Implémentation du glisser-ancrer dans les contrôles

Remarque Les propriétés de glisser-ancrer sont disponibles dans la VCL, mais non dans CLX.

Les descendants de *TWinControl* peuvent faire office de sites ancrés et les descendants de *TControl* peuvent faire office de fenêtres enfant ancrées dans les sites d'ancrage. Par exemple, pour fournir un site d'ancrage sur le bord gauche de la fenêtre d'une fiche, alignez un volet sur le bord gauche de la fiche et faites-en un site d'ancrage. Lorsque des contrôles ancrables sont déplacés vers le volet puis lâchés, ils deviennent des contrôles enfant du volet.

- Transformation d'un contrôle fenêtré en un site d'ancrage
- Transformation d'un contrôle en un enfant ancrable
- Contrôle de l'ancrage des contrôles enfant
- Contrôle du désancrage des contrôles enfant
- Contrôle de la réponse des contrôles enfant aux opérations glisser-ancrer

Transformation d'un contrôle fenêtré en un site d'ancrage

Remarque Les propriétés de glisser-ancrer sont disponibles dans la VCL, mais non dans CLX.

Pour transformer un contrôle fenêtré en un site d'ancrage :

1. Mettez la propriété *DockSite* à *True*.
2. Si l'objet site ancré ne doit apparaître que lorsqu'il contient un client ancré, mettez sa propriété *AutoSize* à *True*. Lorsque *AutoSize* est à *True*, le site ancré a pour taille 0 jusqu'à ce qu'il accepte d'ancrer un contrôle enfant, après quoi il est redimensionné de sorte qu'il englobe le contrôle enfant.

Transformation d'un contrôle en un enfant ancrable

Remarque Les propriétés de glisser-ancrer sont disponibles dans la VCL, mais non dans CLX.

Pour transformer un contrôle en un enfant ancrable :

1. Mettez sa propriété *DragKind* à *dkDock*. Lorsque *DragKind* est à *dkDock*, le fait de faire glisser le contrôle déplace ce dernier vers un nouveau site d'ancrage ou désancre le contrôle qui devient une fenêtre flottante. Lorsque *DragKind* est à *dkDrag* (valeur par défaut), le fait de faire glisser le contrôle démarre une opération glisser-déplacer qui doit être implémentée à l'aide des événements *OnDragOver*, *OnEndDrag* et *OnDragDrop*.
2. Mettez sa propriété *DragMode* à *dmAutomatic*. Lorsque *DragMode* est à *dmAutomatic*, le glissement (glisser-déplacer ou ancrage, suivant *DragKind*) est automatiquement lancé lorsque l'utilisateur commence à faire glisser le contrôle avec la souris. Lorsque *DragMode* est à *dmManual*, vous pouvez commencer une opération glisser-ancrer (ou glisser-déplacer) en appelant la méthode *BeginDrag*.
3. Définissez sa propriété *FloatingDockSiteClass* pour indiquer le descendant *TWinControl* qui doit héberger le contrôle lorsqu'il est désancré et devient une fenêtre flottante. Lorsque le contrôle est libéré et hors d'un site d'ancrage, un contrôle fenêtré de cette classe est dynamiquement créé et devient le parent de l'enfant ancrable. Si le contrôle enfant ancrable est un descendant de *TWinControl*, il n'est pas nécessaire de créer un site ancré flottant séparé pour héberger le contrôle, bien qu'il soit possible de spécifier une fiche pour obtenir une bordure et une barre de titre. Pour ignorer une fenêtre conteneur dynamique, attribuez à *FloatingDockSiteClass* la même classe que le contrôle et elle deviendra une fenêtre flottante sans parent.

Contrôle de l'ancrage des contrôles enfant

Remarque Les propriétés de glisser-ancrer sont disponibles dans la VCL, mais non dans CLX.

Un site d'ancrage accepte automatiquement les contrôles enfant lorsqu'ils sont libérés au-dessus de lui. Pour la plupart des contrôles, le premier enfant est ancré pour remplir la zone client, le deuxième divise cette dernière en différentes régions, et ainsi de suite. Les contrôles de page ancrent les enfants dans de nouvelles feuilles à onglets (ou fusionnent dans des feuilles à onglets si l'enfant est un autre contrôle de page).

Trois événements permettent aux sites d'influer sur l'ancrage des contrôles enfant :

```
property OnGetSiteInfo: TGetSiteInfoEvent;
TGetSiteInfoEvent = procedure(Sender: TObject; DockClient: TControl; var   InfluenceRect:
TRect; var CanDock: Boolean) of object;
```

OnGetSiteInfo intervient sur le site d'ancrage lorsque l'utilisateur fait glisser un enfant ancrable sur le contrôle. Il permet au site d'indiquer s'il accepte en tant qu'enfant le contrôle spécifié par le paramètre *DockClient* et, si tel est le cas, où l'enfant doit se trouver en vue de son ancrage. Lorsque *OnGetSiteInfo* intervient, *InfluenceRect* est initialisée selon les coordonnées d'écran du site d'ancrage et *CanDock* est intialisée à *True*. Une région d'ancrage plus limitée peut être créée en changeant *InfluenceRect* et l'enfant peut être rejeté en mettant *CanDock* à *False*.

```
property OnDockOver: TDockOverEvent;
TDockOverEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y:  Integer; State:
TDragState; var Accept: Boolean) of object;
```

OnDockOver intervient sur le site d'ancrage lorsque l'utilisateur fait glisser un enfant ancrable sur le contrôle. Il est analogue à l'événement *OnDragOver* au cours d'une opération normale de glisser-déposer. Utilisez-le pour indiquer que l'enfant peut être relâché en vue de son ancrage, en initialisant la propriété *Accept*. Si le contrôle ancrable est rejeté par le gestionnaire d'événement *OnGetSiteInfo* (par exemple, si le type de contrôle est incorrect), *OnDockOver* ne se produit pas.

```
property OnDockDrop: TDockDropEvent;
TDockDropEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y:  Integer) of
object;
```

OnDockDrop intervient sur le site d'ancrage lorsque l'utilisateur relâche l'enfant ancrable sur le contrôle. Il est analogue à l'événement *OnDragDrop* au cours d'une opération normale de glisser-déposer. Utilisez-le pour faire en sorte d'accepter le contrôle en tant que contrôle enfant. L'accès au contrôle enfant peut être obtenu à l'aide de la propriété *Control* de *TDockObject* spécifié par le paramètre *Source*.

Contrôle du désancrage des contrôles enfant

Remarque Les propriétés de glisser-ancrer sont disponibles dans la VCL, mais non dans CLX.

Un site d'ancrage permet de désancrer automatiquement les contrôles enfant lorsqu'ils sont déplacés et que leur propriété *DragMode* vaut *dmAutomatic*. Les sites d'ancrage peuvent réagir lorsque les contrôles enfant sont retirés, et même empêcher le désancrage, dans un gestionnaire d'événement *OnUnDock* :

```
property OnUnDock: TUnDockEvent;
TUnDockEvent = procedure(Sender: TObject; Client: TControl; var Allow: Boolean)   of
object;
```

Le paramètre *Client* indique le contrôle enfant qui tente un désancrage et le paramètre *Allow* permet au site d'ancrage (*Sender*) de rejeter le désancrage. Lorsque vous implémentez un gestionnaire d'événement *OnUnDock*, il peut être utile de connaître les autres enfants éventuellement ancrés. Ces informations figurent dans la propriété en lecture seule *DockClients*, qui est un tableau indexé de *TControl*. Le nombre de clients ancrés est donné par la propriété en lecture seule *DockClientCount*.

Contrôle de la réponse des contrôles enfant aux opérations glisser-ancrer

Remarque Les propriétés de glisser-ancrer sont disponibles dans la VCL, mais non dans CLX.

Les contrôles enfant ancrables disposent de deux événements qui interviennent au cours des opérations glisser-ancrer. *OnStartDock*, analogue à l'événement *OnStartDrag* d'une opération glisser-déposer, permet au contrôle enfant ancrable de créer un objet glisser personnalisé. *OnEndDock*, comme *OnEndDrag*, se produit lorsque l'opération glisser s'achève.

Manipulation du texte dans les contrôles

Les sections suivantes expliquent comment utiliser les différentes fonctions des contrôles éditeur de texte formaté et des contrôles mémo. Certaines de ces fonctions peuvent aussi être utilisées avec les contrôles éditeur.

- Définition de l'alignement du texte
- Ajout de barres de défilement en mode exécution
- Ajout de l'objet presse-papiers
- Sélection de texte
- Sélection de la totalité d'un texte
- Opérations couper, copier et coller
- Suppression du texte sélectionné
- Désactivation des éléments de menu
- Ajout d'un menu surgissant
- Gestion de l'événement OnPopup

Définition de l'alignement du texte

Dans un composant mémo ou éditeur de texte formaté, le texte peut être aligné à gauche, à droite ou centré. Pour modifier l'alignement du texte, spécifiez la propriété *Alignment* du composant. L'alignement n'est appliqué que si la propriété *WordWrap* est à *True* ; si le retour à la ligne automatique est désactivé, il n'existe pas de marge sur laquelle s'aligner.

Par exemple, le code suivant attache un gestionnaire d'événement OnClick à l'élément de menu Caractère | Gauche, puis attache le même gestionnaire d'événement aux deux éléments de Droit et Centré du menu Caractère.

```
procedure TEditForm.AlignClick(Sender: TObject);
begin
  Left1.Checked := False;  { efface les trois coches }
  Right1.Checked := False;
  Center1.Checked := False;
  with Sender as TMenuItem do Checked := True;  { coche l'élément cliqué }
  with Editor do  { puis initialise Alignment pour faire correspondre }
    if Left1.Checked then
      Alignment := taLeftJustify
    else if Right1.Checked then
      Alignment := taRightJustify
    else if Center1.Checked then
      Alignment := taCenter;
end;
```

Ajout de barres de défilement en mode exécution

Les composants mémo ou éditeur de texte formaté peuvent contenir des barres de défilement horizontales ou verticales ou les deux, selon les besoins. Lorsque le retour à la ligne automatique est actif, le composant n'a besoin que d'une barre de défilement vertical. Si l'utilisateur désactive le retour à la ligne automatique, le composant a besoin aussi d'une barre de défilement horizontal, puisque le texte n'est plus limité par le bord droit de l'éditeur.

Pour ajouter des barres de défilement en mode exécution :

1. Déterminez si le texte peut dépasser la marge droite. Dans la majorité des cas, cela implique de tester si le retour à la ligne automatique est activé. Vous devrez aussi vérifier qu'il existe réellement des lignes dépassant la largeur du contrôle.
2. Définissez la propriété *ScrollBars* du composant mémo ou éditeur de texte formaté de façon à inclure ou à exclure les barres de défilement.

L'exemple suivant attache le gestionnaire de l'événement *OnClick* à l'élément de menu Caractères | Retour à la ligne.

```
procedure TEditForm.WordWrap1Click(Sender: TObject);
begin
  with Editor do
  begin
```

```
    WordWrap := not WordWrap;  { active ou désactive le retour à la ligne }
    if WordWrap then
      ScrollBars := ssVertical  { seule la barre verticale est nécessaire }
    else
      ScrollBars := ssBoth;  { deux barres peuvent être nécessaires }
      WordWrap1.Checked := WordWrap;  { coche ou désactive l'élément de menu }
  end;
end;
```

Les composants mémo ou éditeur de texte formaté ne gèrent pas les barres de défilement exactement de la même manière. Le composant éditeur de texte formaté peut dissimuler ses barres de défilement si le texte ne sort pas des limites du composant. Le composant Mémo affiche toujours les barres de défilement lorsqu'elles ont été activées.

Ajout de l'objet Clipboard

La plupart des applications manipulant du texte permettent aux utilisateurs de déplacer un texte sélectionné d'un document vers un autre, même s'il s'agit d'une autre application. L'objet *Clipboard* de Delphi encapsule un presse-papiers (comme le Presse-papiers de Windows) et inclut les méthodes permettant de couper, de copier et de coller du texte (ainsi que d'autres formats, par exemple les graphiques). L'objet *Clipboard* est déclaré dans l'unité *Clipbrd*.

Pour ajouter l'objet *Clipboard* à une application,

1. Sélectionnez l'unité qui utilisera le presse-papiers.
2. Recherchez le mot réservé **implementation**.
3. Ajoutez *Clipbrd* à la clause **uses** sous **implementation**.
   - Si une clause **uses** existe déjà dans la partie **implementation**, ajoutez *Clipbrd* à la fin de celle-ci.
   - S'il n'y a pas de clause **uses**, ajoutez-en une rédigée ainsi :

     ```
     uses Clipbrd;
     ```

Par exemple, dans une application contenant une fenêtre enfant, la clause uses dans la partie implémentation de l'unité peut ressembler à ceci :

```
uses
  MDIFrame, Clipbrd;
```

Sélection de texte

Pour transférer du texte dans le presse-papiers, il faut d'abord sélectionner ce texte. La possibilité de mettre en surbrillance le texte sélectionné est intégrée aux composants éditeur. Lorsque l'utilisateur sélectionne un texte, celui-ci apparaît en surbrillance.

Le tableau suivant dresse la liste des propriétés fréquemment utilisées pour la manipulation du texte sélectionné.

Tableau 7.1 Propriétés du texte sélectionné

Propriété	Description
SelText	Contient une chaîne représentant le texte sélectionné dans le composant.
SelLength	Contient la longueur d'une chaîne sélectionnée.
SelStart	Contient la position de départ d'une chaîne.

Sélection de la totalité d'un texte

La méthode *SelectAll* sélectionne la totalité du texte présent dans le composant mémo ou éditeur de texte formaté. C'est particulièrement utile quand le contenu de l'éditeur dépasse la zone visible du composant. Dans les autres cas, les utilisateurs peuvent sélectionner du texte à l'aide du clavier ou de la souris.

Pour sélectionner la totalité du contenu d'un composant mémo ou éditeur de texte formaté, appelez la méthode *SelectAll* du contrôle *RichEdit1*.

Par exemple,

```
procedure TMainForm.SelectAll(Sender: TObject);
begin
  RichEdit1.SelectAll;  { sélectionne tout le texte du composant RichEdit }
end;
```

Couper, copier et coller du texte

Les applications utilisant l'unité *Clipbrd* peuvent couper, copier et coller du texte, des graphiques et des objets, dans le presse-papiers. Les composants éditeur qui encapsulent les contrôles de manipulation de texte standard disposent tous de méthodes intégrées autorisant les interactions avec le presse-papiers. Pour plus d'informations sur l'utilisation des graphiques et du presse-papiers, voir "Utilisation du presse-papiers avec les graphiques" à la page 8-22.

Pour couper, copier ou coller du texte avec le presse-papiers, appelez respectivement les méthodes *CutToClipboard*, *CopyToClipboard* et *PasteFromClipboard* du composant.

Par exemple, le code suivant attache des gestionnaires aux événements *OnClick* des commandes Edition | Couper, Edition | Copier et Edition | Coller :

```
procedure TEditForm.CutToClipboard(Sender: TObject);
begin
  Editor.CutToClipboard;
end;
procedure TEditForm.CopyToClipboard(Sender: TObject);
begin
  Editor.CopyToClipboard;
end;
procedure TEditForm.PasteFromClipboard(Sender: TObject);
```

```
begin
  Editor.PasteFromClipboard;
end;
```

Effacement du texte sélectionné

Vous pouvez effacer le texte sélectionné dans un éditeur sans le placer dans le presse-papiers. Pour ce faire, appelez la méthode *ClearSelection* de l'éditeur. Par exemple, s'il existe un élément Supprimer dans le menu Edition, votre code peut ressembler à :

```
procedure TEditForm.Delete(Sender: TObject);
begin
  RichEdit1.ClearSelection;
end;
```

Désactivation des éléments de menu

Il est souvent utile de désactiver des commandes de menus sans pour autant les retirer du menu. Dans un éditeur de texte, par exemple, si aucun texte n'est sélectionné, les commandes Couper, Copier et Supprimer du menu Edition sont inapplicables. L'activation ou la désactivation des éléments de menu peut être déclenchée lorsque l'utilisateur sélectionne le menu. Pour désactiver un élément de menu, donnez la valeur *False* à sa propriété *Enabled*.

Dans l'exemple suivant, un gestionnaire est attaché à l'événement *OnClick* d'un élément Edition appartenant à la barre de menu d'une fiche enfant. Il définit la propriété *Enabled* des éléments Couper, Copier et Supprimer dans le menu Edition, selon que du texte est sélectionné ou non dans le composant *RichEdit1*. La commande Coller sera activée ou désactivée selon que le presse-papiers contient ou non du texte.

```
procedure TEditForm.Edit1Click(Sender: TObject);
var
  HasSelection: Boolean;  { déclare une variable temporaire }
begin
  Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);  {active ou désactive l'élément Coller }
  HasSelection := Editor.SelLength > 0;  { True si du texte est sélectionné }
  Cut1.Enabled := HasSelection;  { activation des éléments si HasSelection vaut True }
  Copy1.Enabled := HasSelection;
  Delete1.Enabled := HasSelection;
end;
```

La méthode *HasFormat* du presse-papiers renvoie une valeur booléenne indiquant si le presse-papiers contient des objets, du texte ou des images d'un format particulier. En appelant *HasFormat* avec le paramètre *CF_TEXT*, vous pouvez déterminer si le presse-papiers contient du texte, et activer ou désactiver l'élément Coller selon le cas.

Pour plus d'informations sur l'utilisation du presse-Papiers avec des graphiques, voir chapitre 8, "Utilisation des graphiques et du multimédia".

Ajout d'un menu surgissant

Les menus surgissants (ou locaux) sont d'un usage courant et faciles à mettre en œuvre dans toute sorte d'application. Ils réduisent le nombre d'opérations nécessaires à la réalisation des tâches : en cliquant avec le bouton droit de la souris sur l'espace de travail de l'application, l'utilisateur accède à une liste regroupant les commandes les plus fréquemment utilisées.

Dans une application éditeur de texte, par exemple, vous pouvez ajouter un menu surgissant qui comporte les commandes d'édition Couper, Copier et Coller. Ces éléments de menu surgissant peuvent utiliser les mêmes gestionnaires d'événements que les éléments correspondants du menu Edition. Il n'est pas nécessaire de créer des raccourcis clavier, ni des touches raccourci pour les menus surgissants, car les éléments des menus qui leur correspondent en possèdent généralement.

La propriété *PopupMenu* d'une fiche indique quel menu surgissant doit s'afficher lorsque l'utilisateur clique avec le bouton droit de la souris sur la fiche. Les différents contrôles possèdent aussi leurs propriétés *PopupMenu* qui ont priorité sur la propriété de la fiche, permettant de définir des menus personnalisés pour des contrôles particuliers.

Pour ajouter un menu surgissant à une fiche,

1. Placez un composant menu surgissant sur la fiche.
2. Utilisez le concepteur de menus pour définir les éléments du menu surgissant.
3. Définissez par le nom du composant menu surgissant la propriété *PopupMenu* de la fiche ou du contrôle devant faire apparaître le menu.
4. Attachez les gestionnaires aux événements *OnClick* des éléments du menu surgissant.

Gestion de l'événement OnPopup

Il peut être nécessaire de préparer certains éléments d'un menu surgissant avant d'afficher celui-ci, comme vous devez spécifier les éléments activés ou désactivés d'un menu normal. Avec un menu normal, l'événement *OnClick* correspondant à l'affichage du menu est généralement associé au titre de ce menu, comme décrit dans la section "Désactivation des éléments de menu" à la page 7-11.

Comme les menus surgissants n'ont pas de barre de menu, vous devez gérer l'événement dans le composant lui-même. Le composant menu surgissant offre pour cela un événement particulier appelé *OnPopup*.

Pour préparer des éléments d'un menu surgissant avant de les afficher,

1. Sélectionnez le composant menu surgissant.
2. Attachez un gestionnaire à son événement *OnPopup*.
3. Ecrivez dans le gestionnaire d'événement le code activant, désactivant, dissimulant ou affichant les éléments du menu.

Dans le code suivant, un gestionnaire existant pour l'événement *Edit1Click* décrit précédemment dans la section "Désactivation des éléments de menu" à la page 7-11 est attaché à l'événement *OnPopup* du composant menu surgissant. Une ligne de code est ajoutée à *Edit1Click* pour chaque élément du menu surgissant.

```
procedure TEditForm.Edit1Click(Sender: TObject);
var
  HasSelection: Boolean;
begin
  Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);
  Paste2.Enabled := Paste1.Enabled;{Ajoute cette ligne}
  HasSelection := Editor.SelLength <> 0;
  Cut1.Enabled := HasSelection;
  Cut2.Enabled := HasSelection;{Ajoute cette ligne}
  Copy1.Enabled := HasSelection;
  Copy2.Enabled := HasSelection;{Ajoute cette ligne}
  Delete1.Enabled := HasSelection;
end;
```

Ajout de graphiques à des contrôles

Plusieurs contrôles permettent de personnaliser la manière dont le contrôle est restitué. Ce sont les boîtes liste, boîtes à options, menus, en-têtes, contrôles onglets, vues liste, barres d'état, vues arborescentes et barres d'état. Au lieu d'utiliser la méthode standard dessinant un contrôle ou chacun de ses éléments, le propriétaire du contrôle (généralement la fiche) dessine ces éléments en mode exécution. L'utilisation la plus courante de ces contrôles dessinés par le propriétaire est de remplacer le texte par des dessins ou d'ajouter des dessins au texte des éléments. Pour des informations sur l'utilisation du style "dessiné par le propriétaire" pour ajouter des images aux menus, voir "Ajout d'images à des éléments de menu" à la page 6-40.

Les contrôles dessinés par le propriétaire ont un point commun : ils contiennent tous des listes d'éléments. Généralement, il s'agit de listes de chaînes qui sont affichées sous forme de texte ou de liste d'objets contenant des chaînes qui sont affichées sous forme de texte. Il est possible d'associer un objet à chaque élément de ces listes et d'utiliser l'objet lorsque vous dessinez un élément.

Dans Delphi, la création d'un contrôle dessiné par le propriétaire se fait généralement en trois étapes :

1. Spécification du style dessiné par le propriétaire
2. Ajout d'objets graphiques à une liste de chaînes
3. Dessiner des éléments dessinés par le propriétaire

Spécification du style dessiné par le propriétaire

Pour personnaliser le dessin d'un contrôle, vous devez spécifier des gestionnaires d'événements qui restituent l'image du contrôle quand il doit être dessiné. Certains contrôles reçoivent automatiquement ces événements. Par exemple, les vues liste ou arborescentes et les barres d'outils reçoivent les événements aux diverses étapes du processus de dessin sans avoir à définir la moindre propriété. Ces événements ont des noms de la forme "OnCustomDraw" ou "OnAdvancedCustomDraw".

D'autres contrôles nécessitent l'initialisation d'une propriété avant de recevoir les événements de dessin personnalisé. Les boîtes liste, les boîtes à options, les en-têtes et les barres d'état ont une propriété appelée *Style*. La propriété *Style* détermine si le contrôle utilise le dessin par défaut (appelé style "standard") ou bien le dessin effectué par le propriétaire. Les grilles utilisent une propriété appelée *DefaultDrawing* qui permet d'activer ou de désactiver le dessin par défaut. Les vues listes et les contrôles onglets ont une propriété appelée *OwnerDraw* qui active ou désactive le dessin par défaut.

Pour les boîtes liste et les boîtes à options, il y a plusieurs styles dessinés par le propriétaire, appelés *fixed* et *variable*, comme décrit dans le tableau suivant. Les autres contrôles sont toujours "fixes" : bien que la taille de l'élément contenant du texte soit variable, la taille de chaque élément est fixée avant le dessin du contrôle.

Tableau 7.2 Comparaison entre les styles "fixed" et "variable"

Styles dessinés par le propriétaire	Signification	Exemples
Fixed	Chaque élément est de la même hauteur, déterminée par la propriété *ItemHeight*.	*lbOwnerDrawFixed*, *csOwnerDrawFixed*
Variable	Chaque élément peut avoir une hauteur différente qui dépend des données au moment de l'exécution.	*lbOwnerDrawVariable*, *csOwnerDrawVariable*

Ajout d'objets graphiques à une liste de chaînes

Toute liste de chaînes est capable de contenir une liste d'objets en plus de sa liste de chaînes.

Dans une application de gestion de fichiers, par exemple, vous devez ajouter un bitmap indiquant le type du lecteur à la lettre le désignant. Pour cela, vous devez ajouter les images bitmap à l'application, puis les copier à l'endroit approprié dans la liste de chaînes, comme le décrivent les sections suivantes.

Ajout d'images à une application

Un contrôle image est un contrôle non visuel qui contient une image graphique (un bitmap, par exemple). Les contrôles image servent à afficher des images graphiques sur une fiche, mais vous pouvez aussi les utiliser pour stocker des

images cachées que vous utiliserez dans votre application. Par exemple, il est possible de stocker des images bitmap pour les contrôles dessinés par le propriétaire dans des contrôles image cachés, comme décrit ci-dessous :

1. Ajoutez des contrôles image à la fiche principale.
2. Définissez leurs propriétés *Name*.
3. Donnez la valeur *False* à la propriété *Visible* de chaque contrôle image.
4. Définissez la propriété *Picture* de chaque contrôle image par le bitmap souhaité en utilisant l'éditeur d'image depuis l'inspecteur d'objets.

Les contrôles image seront invisibles lorsque vous exécuterez l'application.

Ajout d'images à une liste de chaînes

Une fois que vous avez des images graphiques dans une application, vous pouvez les associer aux chaînes de la liste. Vous pouvez soit ajouter les objets en même temps que les chaînes, soit les associer à des chaînes qui ont déjà été ajoutées. Si vous disposez de toutes les données dont vous avez besoin, vous ajouterez sans doute les chaînes et les objets en même temps.

L'exemple suivant montre comment ajouter des images à une liste de chaînes. Ce code est extrait d'une application de gestion de fichiers dans laquelle chaque lecteur correct est représenté par une lettre et est associé à un bitmap indiquant le type du lecteur. L'événement *OnCreate* se présente comme suit :

```
procedure TFMForm.FormCreate(Sender: TObject);
var
  Drive: Char;
  AddedIndex: Integer;
begin
  for Drive := 'A' to 'Z' do  { passe par tous les lecteurs possibles }
  begin
    case GetDriveType(Drive +':/') of { valeurs positives signifiant des lecteurs valides }
     DRIVE_REMOVABLE:  { ajoute un onglet }
       AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Floppy.Picture.Graphic);
     DRIVE_FIXED:  { ajoute un onglet }
       AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Fixed.Picture.Graphic);
     DRIVE_REMOTE:  { ajoute un onglet }
       AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Network.Picture.Graphic);
    end;
    if UpCase(Drive) = UpCase(DirectoryOutline.Drive) then  { lecteur actif ? }
     DriveTabSet.TabIndex := AddedIndex;  { puis en fait l'onglet en cours }
  end;
end;
```

Dessiner des éléments dessinés par le propriétaire

Lorsque vous avez spécifié qu'un contrôle est dessiné par le propriétaire (en initialisant une propriété ou en définissant un gestionnaire d'événement), le contrôle n'est plus dessiné à l'écran. Au lieu de cela, le système d'exploitation génère un événement pour chaque élément visible du contrôle. C'est votre application qui gère ces événements et dessine les éléments.

Pour dessiner les éléments d'un contrôle dessiné par le propriétaire, suivez les étapes indiquées ci-après. Ces étapes se répètent pour chaque élément visible du contrôle, mais vous utiliserez le même gestionnaire d'événement pour tous.

1 Le cas échéant, dimensionnez l'élément.

   Si les éléments sont tous de même taille (par exemple, avec un style de boîte liste *lsOwnerDrawFixed*), cette opération n'est pas nécessaire.

2 Dessinez l'élément.

Dimensionnement des éléments dessinés par le propriétaire

Avant de laisser votre application dessiner chaque élément d'un contrôle de taille variable lorsqu'il est dessiné par le propriétaire, le système d'exploitation génère un événement de type *measure-item*. Cet événement indique à l'application l'endroit où l'élément apparaîtra sur le contrôle.

Delphi détermine la taille probable de l'élément (généralement juste assez grand pour afficher le texte de l'élément dans la police de caractères active). Votre application peut gérer l'événement et modifier la zone rectangle choisie. Par exemple, si vous comptez remplacer le texte de l'élément par une image bitmap, vous modifierez le rectangle pour qu'il soit de la taille du bitmap. Si vous voulez avoir à la fois l'image et le texte, vous ajusterez la taille du rectangle pour qu'il puisse contenir les deux.

Pour changer la taille d'un élément dessiné par le propriétaire, attachez un gestionnaire à l'événement measure-item dans le contrôle dessiné par le propriétaire. Le nom de l'événement peut varier en fonction du contrôle. Les boîtes liste et les boîtes à options utilisent *OnMeasureItem.* Les grilles n'ont pas ce type d'événement.

L'événement définissant la taille utilise deux paramètres importants : l'indice et la taille de l'élément. Cette taille est variable : l'application peut l'augmenter ou la diminuer. La position des éléments suivants dépend de la taille des éléments précédents.

Par exemple, dans une boîte liste variable dessinée par le propriétaire, si l'application définit la hauteur du premier élément à cinq pixels, le second élément commence au sixième pixel depuis le haut, et ainsi de suite. Dans les boîtes liste et dans les boîtes à options, le seul aspect des éléments que l'application puisse changer est la hauteur. La largeur de l'élément est toujours celle du contrôle.

Les grilles dessinées par le propriétaire ne peuvent pas modifier la taille des cellules au fur et à mesure qu'elles sont dessinées. En effet, la taille des lignes et des colonnes est définie avant le dessin par les propriétés *ColWidths* et *RowHeights*.

Le code suivant, attaché à l'événement *OnMeasureItem* du composant boîte liste dessinée par le propriétaire, augmente la hauteur de chaque élément de liste pour permettre de placer l'image bitmap associée.

```
procedure TFMForm.DriveTabSetMeasureTab(Sender: TObject; Index: Integer;
  var TabWidth: Integer);  { notez que TabWidth définit un paramètre var}
var
  BitmapWidth: Integer;
begin
  BitmapWidth := TBitmap(DriveTabSet.Tabs.Objects[Index]).Width;
  { augmente la largeur de l'onglet de celle du bitmap associé plus deux }
  Inc(TabWidth, 2 + BitmapWidth);
end;
```

Remarque Vous devez transtyper les éléments à partir de la propriété *Objects* dans la liste de chaînes. *Objects* est une propriété de type *TObject*, aussi peut-elle contenir n'importe quel type d'objet. Lorsque vous extrayez un objet d'un tableau, vous devez le transtyper afin qu'il reprenne le type des éléments.

Dessin des éléments par le propriétaire

Lorsqu'une application doit dessiner ou redessiner un contrôle dessiné par le propriétaire, le système d'exploitation génère un événement de type *draw-item* pour chaque élément visible du contrôle. Selon le contrôle, l'élément peut également recevoir les événements de dessin pour l'élément pris comme un tout ou comme sous-éléments.

Pour dessiner chaque élément d'un contrôle dessiné par le propriétaire, attachez un gestionnaire à l'événement *draw-item* de ce contrôle.

Les noms des événements relatifs aux objets dessinés par le propriétaire commencent généralement par :

- *OnDraw*, comme *OnDrawItem* ou *OnDrawCell*
- *OnCustomDraw*, comme *OnCustomDrawItem*
- *OnAdvancedCustomDraw*, comme *OnAdvancedCustomDrawItem*

L'événement *draw-item* contient des paramètres identifiant l'élément à dessiner, le rectangle dans lequel il s'inscrit et, habituellement, des informations sur son état (actif, par exemple). L'application gère chaque événement en plaçant l'élément approprié dans le rectangle transmis.

Par exemple, le code suivant montre comment dessiner des éléments dans une boîte liste ayant un bitmap associé à chaque chaîne. Il attache ce gestionnaire à l'événement *OnDrawItem* :

```
procedure TFMForm.DriveTabSetDrawTab(Sender: TObject; TabCanvas: TCanvas;
  R: TRect; Index: Integer; Selected: Boolean);
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap(DriveTabSet.Tabs.Objects[Index]);
  with TabCanvas do
  begin
  Draw(R.Left, R.Top + 4, Bitmap);  { dessine le bitmap }
  TextOut(R.Left + 2 + Bitmap.Width,  { positionne le texte }
    R.Top + 2, DriveTabSet.Tabs[Index]);  { et le dessine à droite du bitmap }
  end;
end;
```

Chapitre

8

Utilisation des graphiques et du multimédia

Les éléments graphiques et multimédia permettent d'améliorer la présentation de vos applications. Delphi propose divers moyens d'introduire ces caractéristiques dans votre application. Pour ajouter des éléments graphiques, vous pouvez insérer des images pré-dessinées à la conception, les créer en utilisant des contrôles graphiques à la conception ou les dessiner dynamiquement à l'exécution. Pour ajouter des fonctions multimédia, Delphi propose des composants spéciaux qui peuvent jouer des séquences audio et vidéo. Notez que les composants multimédia ne sont disponibles en programmation multiplate-forme.

Présentation de la programmation relative aux graphiques

Les composants graphiques de la VCL définis dans l'unité Graphics encapsulent la GDI (Graphics Device Interface) de Windows pour faciliter l'ajout de graphiques aux applications Windows. Les composants graphiques CLX définis dans l'unité QGraphics encapsulent les widgets graphiques Qt pour permettre d'ajouter des graphiques aux applications multiplates-formes.

Lorsque vous dessinez des graphiques dans une application Delphi, vous travaillez sur le *canevas*, de l'objet, plutôt que directement sur l'objet. Le mot *Canvas* désigne une propriété de l'objet, mais c'est aussi un objet. Le principal avantage de l'objet canevas est qu'il gère efficacement des ressources et prend en compte le contexte de périphérique. Que vous dessiniez sur des bitmaps, sur l'écran, sur l'imprimante ou sur des métafichiers (dessins dans CLX), vos programmes peuvent utiliser les mêmes méthodes. Les canevas sont uniquement disponibles en phase d'exécution. Tout le travail relatif aux canevas se fait donc en écrivant du code. Les sections suivantes expliquent comment utiliser les composants graphiques de la VCL pour simplifier les opérations de programmation.

Remarque VCL Puisque *TCanvas* est un gestionnaire de ressources qui enveloppe le contexte de périphérique Windows, vous pouvez aussi utiliser toutes les fonctions GDI de Windows sur le canevas. La propriété *Handle* du canevas *est* le handle du contexte de périphérique.

Remarque CLX *TCanvas* est un gestionnaire de ressources qui enveloppe un dispositif de dessin Qt. La propriété *Handle* du canevas *est* un pointeur typé sur l'instance d'un objet dispositif de dessin Qt. Le fait que ce pointeur d'instance soit exposé vous permet d'utiliser les fonctions de bas niveau de la bibliothèque graphique Qt qui nécessitent un pointeur d'instance sur un objet dispositif de dessin.

La façon dont les images graphiques apparaissent dans votre application dépend de la façon dont elles sont dessinées. Si vous dessinez directement dans le canevas d'un contrôle, l'image est affichée directement. Toutefois, si vous dessinez sur une image hors écran comme un canevas *Tbitmap*, l'image n'apparaît que lorsqu'un contrôle effectue la copie d'un bitmap sur le canevas du contrôle. Ce qui signifie que lorsqu'on dessine des bitmaps et qu'on les affecte à un contrôle image, l'image n'apparaît que si le contrôle a la possibilité de traiter son message (VCL) ou son événement (CLX) *OnPaint*.

Lorsqu'on travaille avec des graphiques, on rencontre fréquemment les termes *dessin* et *peinture* :

- Lorsque vous dessinez, vous créez avec du code un seul élément graphique spécifique tel qu'une ligne ou une forme. Dans votre code, vous indiquez à un objet de dessiner un graphique particulier à un emplacement particulier sur son canevas en appelant une méthode de dessin du canevas.
- Lorsque vous peignez, vous créez l'apparence entière d'un objet. La peinture implique généralement le dessin. En effet, en réponse à des événements *OnPaint*, un objet dessinera des graphiques. Une boîte de saisie, par exemple, se peint elle-même en dessinant un rectangle, puis en dessinant du texte à l'intérieur. Un contrôle forme, en revanche, se peint lui-même en dessinant un seul graphique.

Les exemples donnés au début de ce chapitre démontrent comment dessiner divers graphiques en réponse à des événements *OnPaint*. Les sections ultérieures montrent comment faire le même type de dessin en réponse à d'autres événements.

Rafraîchissement de l'écran

A certains moments, le système d'exploitation détermine que l'apparence des objets affichés à l'écran doit être rafraîchie. Windows génère des messages WM_PAINT que la VCL redirige vers des gestionnaires d'événements *OnPaint.* (Si vous utilisez CLX dans le cadre du développement multiplate-forme, un événement de dessin est généré que CLX redirige vers des gestionnaires d'événements *OnPaint*.) Si vous avez écrit un gestionnaire de l'événement *OnPaint* pour cet objet, il est appelé lorsque vous utilisez la méthode *Refresh*. Le nom généré par défaut pour le gestionnaire d'événement *OnPaint* dans un fiche est *FormPaint*. La méthode *Refresh* est parfois utilisée pour rafraîchir un composant ou une fiche. Par exemple, la méthode *Refresh* peut être appelée dans

le gestionnaire d'événement *OnResize* de la fiche afin de réafficher des graphiques ou, si vous utilisez la VCL, pour dessiner un fond sur la fiche.

Bien que certains systèmes d'exploitation gèrent automatiquement l'affichage des zones clientes d'une fenêtre qui ne sont plus valides, Windows ne le fait pas. Pour Windows, tout dessin est considéré comme permanent. Lorsqu'une fiche ou un contrôle est temporairement masqué, par exemple lors d'un glisser-déplacer, la fiche ou le contrôle doivent repeindre la zone masquée lorsqu'elle ne l'est plus. Pour plus d'informations sur le message WM_PAINT, voir l'aide en ligne de Windows.

Lors de l'utilisation du contrôle *TImage* pour afficher une image graphique sur une fiche, le dessin et le rafraîchissement du graphique contenu dans le *TImage* sont gérés automtiquement. La propriété *Picture* spécifie le bitmap, le dessin ou tout autre objet raphique affiché par *TImage*. Vous pouvez aussi définir la propriété *Proportional* pour que l'image l'image soir affichée sans distorsion dans le contrôle image. Dessiner sur un *TImage* crée une image persistante. Par conséquent, il n'est pas nécessaire de redessiner l'image contenue. Au contraire, le canevas d'un *TPaintBox* écrit directement sur le pilote de l'écran (VCL) ou le dispositif de dessin (CLX), et de ce fait, tout ce qui est dessiné sur le canevas du *PaintBox* est transitoire. Cela est vrai pour les contrôles similaires, y compris la fiche elle-même. De plus, si vous dessinez ou peignez à partir du constructeur d'un *TPaintBox*, vous devrez ajouter ce code dans le gestionnaire *OnPaint* afin que l'image soit repeinte à chaque fois que la zone cliente est invalidée.

Types des objets graphiques

La VCL et CLX proposent les objets graphiques énumérés dans le tableau suivant. Ces objets disposent de méthodes pour dessiner dans le canevas, décrites dans la section "Utilisation des méthodes du canevas pour dessiner des objets graphiques" à la page 8-10, et pour charger et enregistrer des fichiers graphiques (voir "Chargement et enregistrement de fichiers graphiques" à la page 8-20).

Tableau 8.1 Types d'objets graphiques

Objet	Description
Picture	Utilisé pour contenir une image graphique. Pour ajouter d'autres formats de fichiers graphiques, utilisez la méthode *Register* de l'objet Picture. Elle permet de gérer des fichiers arbitraires comme l'affichage d'images dans un contrôle image.
Bitmap	Objet graphique utilisé pour créer des images, les manipuler (mise à l'échelle, défilement, rotation et peinture) et les stocker sur disque sous forme de fichiers. Il est très facile de créer la copie d'un bitmap, puisque c'est le *handle* qui est copié et non l'image.
Clipboard	Représente le conteneur d'un texte ou d'un graphique qui est coupé, copié ou collé depuis ou vers une application. Grâce au presse-papiers, vous pouvez extraire des données en fonction d'un format donné ; comptage des références d'handles, et l'ouverture et la fermeture du presse-papiers ; gérer et manipuler des formats pour les objets du presse-papiers.

Tableau 8.1 Types d'objets graphiques (suite)

Objet	Description
Icon	Représente la valeur chargée depuis un fichier icône (fichier .ICO).
Metafile (VCL seulement) Drawing (CLX seulement)	Contient un fichier, qui enregistre les opérations nécessaires à la construction d'une image, au lieu de contenir les pixels du bitmap de l'image. Les métafichiers ou les dessins sont extrêmement réductibles sans perte de détail de l'image et nécessitent souvent moins de mémoire que les bitmaps, particulièrement pour les pilotes haute résolution comme les imprimantes. Mais les métafichiers et les dessins ne sont pas aussi rapides que les bitmaps. Utilisez les métafichiers ou les dessins lorsque vous recherchez souplesse et précison plutôt que les performances.

Propriétés et méthodes communes du canevas

Le tableau suivant énumère les principales propriétés de l'objet canevas. Pour une liste complète des propriétés et des méthodes, voir la rubrique traitant du composant *TCanvas* dans l'aide en ligne.

Tableau 8.2 Propriétés communes de l'objet canevas

Propriété	Description
Font	Spécifie la police devant être utilisée pour écrire du texte sur l'image. Définit les propriétés de l'objet *TFont* afin de spécifier le type de police, sa couleur, sa taille, et son style.
Brush	Détermine la couleur et le modèle utilisés par le canevas pour remplir les fonds et les formes graphiques. Définissez les propriétés de l'objet *TBrush* pour spécifier la couleur et le modèle ou le bitmap à utiliser lors du remplissage des espaces sur le canevas.
Pen	Spécifie le type de crayon utilisé par le canevas pour dessiner des lignes et des formes. Définissez les propriétés de l'objet *TPen* de façon à spécifier la couleur, le style, la largeur et le mode du crayon.
PenPos	Spécifie la position de dessin en cours du crayon.
Pixels	Spécifie la couleur des pixels à l'intérieur du *ClipRect* en cours.

Pour davantage d'informations sur ces propriétés, voir "Utilisation des propriétés de l'objet canevas" à la page 8-5.

Le tableau suivant liste les différentes méthodes pouvant être utilisées :

Tableau 8.3 Méthodes communes de l'objet canevas

Méthode	Description
Arc	Dessine un arc sur l'image ainsi que le périmètre de l'ellipse délimitée par le rectangle spécifié.
Chord	Dessine une figure fermée représentée par l'intersection d'une ligne et d'une ellipse.
CopyRect	Copie une partie de l'image d'un autre canevas sur le canevas.
Draw	Dessine sur le canevas à l'emplacement donné par les coordonnées (X, Y) l'objet graphique spécifié par le paramètre Graphic.

Tableau 8.3 Méthodes communes de l'objet canevas (suite)

Méthode	Description
Ellipse	Dessine sur le canevas l'ellipse définie par un rectangle délimité.
FillRect	Remplit sur le canevas le rectangle spécifié en utilisant le pinceau en cours.
FloodFill (VCL seulement)	Remplit une zone du canevas en utilisant le pinceau en cours.
FrameRect	Dessine un rectangle en utilisant le pinceau du canevas pour dessiner sa bordure.
LineTo	Dessine une ligne sur le canevas en partant de la position de *PenPos* au point spécifié par X et Y, et définit la position du crayon à (X, Y).
MoveTo	Change la position de dessin en cours par le point (X,Y).
Pie	Dessine sur le canevas un secteur de l'ellipse délimitée par le rectangle (X1, Y1) et (X2, Y2).
Polygon	Dessine sur le canevas une série de lignes connectant les points transmis et fermant la forme par une ligne allant du dernier point au premier point.
Polyline	Dessine sur le canevas une série de lignes à la position en cours du crayon et connectant chacun des points transmis dans *Points*.
Rectangle	Dessine sur le canevas un rectangle dont le coin supérieur gauche apparaît au point (X1, Y1) et le coin inférieur droit au point (X2, Y2). Utilisez *Rectangle* pour dessiner un cadre utilisant *Pen* et remplissez-le avec *Brush*.
RoundRect	Dessine sur le canevas un rectangle à coins arrondis.
StretchDraw	Dessine sur le canevas un graphique afin que l'image tienne dans le rectangle spécifié. Le facteur d'amplification de l'image devra sans doute être modifié pour que l'image tienne dans le rectangle.
TextHeight, TextWidth	Renvoie respectivement la hauteur et la largeur d'une chaîne dans la police en cours. La hauteur inclut l'intervalle entre les lignes.
TextOut	Ecrit sur le canevas une chaîne commençant au point (X,Y), puis modifie *PenPos* par rapport à la fin de la chaîne.
TextRect	Ecrit une chaîne à l'intérieur d'une région ; toute partie de la chaîne se trouvant à l'extérieur de la région ne sera pas visible.

Pour davantage d'informations sur ces méthodes, voir "Utilisation des méthodes du canevas pour dessiner des objets graphiques" à la page 8-10.

Utilisation des propriétés de l'objet canevas

A l'objet canevas, il est possible de définir les propriétés d'un crayon afin qu'il dessine des lignes, celles d'un pinceau pour qu'il remplisse des formes, celles d'une fonte pour écrire du texte et celles d'un tableau de pixels pour représenter une image.

Cette section traite des sujets suivants :

- Utilisation des crayons
- Utilisation des pinceaux
- Lecture et définition des pixels

Utilisation des crayons

La propriété *Pen* d'un canevas contrôle la façon dont les lignes apparaissent, y compris les lignes dessinées pour définir le pourtour d'une forme. Dessiner une ligne droite revient à changer un groupe de pixels alignés entre deux points.

Le crayon lui-même possède quatre propriétés qu'il est possible de changer : *Color*, *Width*, *Style*, et *Mode*.

- Propriété Color: modifie la couleur du crayon
- Propriété Width: modifie la largeur du crayon
- Propriété Style: modifie le style du crayon
- Propriété Mode: modifie le mode du crayon

Les valeurs de ces propriétés déterminent la façon dont le crayon change les pixels de la ligne. Par défaut, chaque crayon est noir, a une largeur de 1 pixel, est de style uni, et a un mode appelé copie qui écrase tout ce qui se trouve déjà sur le canevas.

Vous pouvez utiliser *TPenRecall* pour enregistrer et restaurer rapidement les propriétés des crayons.

Changement de la couleur du crayon

La couleur du crayon est définie en mode exécution comme toute autre propriété *Color*. La couleur du crayon détermine la couleur des lignes qu'il dessine : lignes, polylignes et contour des formes, ainsi que d'autres types de lignes et polylignes. Pour modifier la couleur du crayon, donnez une valeur à la propriété *Color* du crayon.

Pour permettre à l'utilisateur de choisir une nouvelle couleur de crayon, vous devez placer une grille de couleurs dans la barre d'outils du crayon. Une grille de couleurs permet de spécifier une couleur de premier plan et une couleur d'arrière-plan. Si vous n'utilisez pas de grille, vous devez penser à fournir une couleur d'arrière-plan pour dessiner les intervalles entre les segments de lignes. La couleur d'arrière-plan provient de la propriété *Color* du pinceau.

Quand l'utilisateur choisit une nouvelle couleur en cliquant dans la grille, ce code modifie la couleur du crayon en réponse à l'événement *OnClick* :

```
procedure TForm1.PenColorClick(Sender: TObject);
begin
  Canvas.Pen.Color := PenColor.ForegroundColor;
end;
```

Changement de l'épaisseur du crayon

L'épaisseur du crayon détermine la taille, exprimée en pixels, de la ligne qu'il dessine.

Remarque N'oubliez pas que lorsque l'épaisseur est supérieure à un pixel, Windows 95/98 dessine toujours une ligne continue, sans tenir compte de la valeur de la propriété *Style* du crayon.

Pour modifier l'épaisseur du crayon, affectez une valeur numérique à la propriété *Width* du crayon.

Supposons que la barre d'outils du crayon contienne une barre de défilement permettant de définir la largeur de celui-ci, et que vous vouliez mettre à jour le libellé attenant à la barre de défilement pour que l'utilisateur voit ce qu'il fait. Pour utiliser la position de la barre de défilement afin de déterminer l'épaisseur du crayon, il est nécessaire de changer l'épaisseur du crayon chaque fois que la position change.

Voici comment traiter l'événement *OnChange* de la barre de défilement :

```
procedure TForm1.PenWidthChange(Sender: TObject);
begin
  Canvas.Pen.Width := PenWidth.Position;{ définit la largeur de crayon directement }
 PenSize.Caption := IntToStr(PenWidth.Position);{ conversion en chaîne pour le libellé }
end;
```

Changement du style du crayon

La propriété *Style* d'un crayon permet de créer des lignes continues, pointillées ou à tirets.

Remarque VCL

Dans le développement d'une application multiplate-forme devant être déployée sous Windows, Windows 95/98 ne permet pas de dessiner des lignes pointillées ou à tirets lorsque le crayon a une largeur supérieure à un pixel. Il dessine à la place une ligne continue, quel que soit le style spécifié.

La définition des propriétés d'un crayon est une opération qui se prête parfaitement au partage d'événements entre plusieurs contrôles. Pour déterminer le contrôle qui reçoit l'événement, il suffit de tester le paramètre *Sender*.

Pour créer un gestionnaire pour l'événement clic de chacun des six boutons de style de la barre d'outils d'un crayon, procédez comme suit :

1. Sélectionnez les six boutons de style de crayon et choisissez Inspecteur d'objets | Evénements | événement *OnClick* et tapez `SetPenStyle` dans la colonne des gestionnaires.

   Delphi génère un gestionnaire vide appelé *SetPenStyle* et l'attache à l'événement *OnClick* de chacun des six boutons.

2. Remplissez le gestionnaire de l'événement *Onclick* en définissant le style du crayon selon la valeur du paramètre *Sender* qui désigne le contrôle ayant envoyé l'événement clic :

```
procedure TForm1.SetPenStyle(Sender: TObject);
begin
  with Canvas.Pen do
  begin
    if Sender = SolidPen then Style := psSolid
   else if Sender = DashPen then Style := psDash
   else if Sender = DotPen then Style := psDot
   else if Sender = DashDotPen then Style := psDashDot
   else if Sender = DashDotDotPen then Style := psDashDotDot
   else if Sender = ClearPen then Style := psClear;
  end;
end;
```

Changement du mode du crayon

La propriété *Mode* d'un crayon vous permet de spécifier les différentes façons de combiner la couleur du crayon à celle du canevas. Par exemple, le crayon peut toujours être noir, être de la couleur inverse à l'arrière-plan du canevas, etc. Pour plus de détails, voir la rubrique traitant de *TPen* dans l'aide en ligne.

Renvoi de la position du crayon

La position de dessin en cours, position à partir de laquelle le crayon va dessiner la prochaine ligne, est appelée la position du crayon. Le canevas stocke la position du crayon dans sa propriété *PenPos*. Cette position n'affecte que le dessin des lignes ; pour les formes et le texte, vous devez spécifier toutes les coordonnées dont vous avez besoin.

Pour définir la position du crayon, appelez la méthode *MoveTo* du canevas. Par exemple, le code suivant déplace la position du crayon sur le coin supérieur gauche du canevas :

```
Canvas.MoveTo(0, 0);
```

Remarque Lorsqu'une ligne est dessinée avec la méthode *LineTo*, la position en cours est déplacée sur le point d'arrivée de la ligne.

Utilisation des pinceaux

La propriété *Brush* d'un canevas détermine la façon dont les zones sont remplies, y compris l'intérieur des formes. Remplir une zone avec le pinceau revient à changer d'une certaine façon un grand nombre de pixels adjacents.

Le pinceau a trois propriétés que vous pouvez manipuler :

- Propriété Color : modifie la couleur de remplissage
- Propriété Style : modifie le style du pinceau
- Propriété Bitmap : utilise un bitmap comme modèle de pinceau

Les valeurs de ces propriétés déterminent la façon dont le canevas remplit les formes et d'autres zones. Par défaut, chaque pinceau est blanc, a un style uni et n'a pas de motif de remplissage.

Vous pouvez utiliser *TBrushRecall* pour enregistrer et restaurer rapidement les propriétés des pinceaux.

Changement de la couleur du pinceau

La couleur du pinceau détermine la couleur utilisée par le canevas pour remplir les formes. Pour modifier la couleur de remplissage, affectez une valeur à la propriété *Color* du pinceau. Le pinceau est utilisé pour la couleur d'arrière-plan dans le dessin de lignes et de texte.

Il est possible de définir la couleur du pinceau de la même manière que celle du crayon, en réponse à un clic dans la grille de couleurs présentée dans la barre d'outils du pinceau (voir la section "Changement de la couleur du crayon" à la page 8-6) :

```
procedure TForm1.BrushColorClick(Sender: TObject);
begin
```

```
  Canvas.Brush.Color := BrushColor.ForegroundColor;
end;
```

Changement du style du pinceau

Le style d'un pinceau détermine le motif utilisé pour remplir les formes. Il vous permet de spécifier différentes façons de combiner la couleur du pinceau à des couleurs déjà présentes sur le canevas. Les styles prédéfinis comprennent des couleurs unies, pas de couleur et divers motifs de lignes et de hachurages.

Pour modifier le style d'un pinceau, définissez sa propriété *Style* par l'une des valeurs prédéfinies suivantes : *bsSolid*, *bsClear*, *bsHorizontal*, *bsVertical*, *bsFDiagonal*, *bsBDiagonal*, *bsCross* ou *bsDiagCross*.

Cet exemple définit le style du pinceau en faisant partager le même gestionnaire d'événement *OnClick* aux huit boutons de style de pinceau. Tous les boutons sont sélectionnés, *OnClick* est sélectionné dans Inspecteur d'objets | Evénements et le gestionnaire *OnClick* porte le nom *SetBrushStyle*.

```
procedure TForm1.SetBrushStyle(Sender: TObject);
begin
  with Canvas.Brush do
  begin
    if Sender = SolidBrush then Style := bsSolid
   else if Sender = ClearBrush then Style := bsClear
   else if Sender = HorizontalBrush then Style := bsHorizontal
   else if Sender = VerticalBrush then Style := bsVertical
   else if Sender = FDiagonalBrush then Style := bsFDiagonal
   else if Sender = BDiagonalBrush then Style := bsBDiagonal
   else if Sender = CrossBrush then Style := bsCross
   else if Sender = DiagCrossBrush then Style := bsDiagCross;
  end;
end;
```

Définition de la propriété Bitmap du pinceau

La propriété *Bitmap* du pinceau vous permet de spécifier une image bitmap qui sera utilisée comme motif de remplissage des formes et des autres zones.

L'exemple suivant charge un bitmap d'un fichier et l'affecte au pinceau du canevas de la fiche Form1 :

```
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create;
  try
    Bitmap.LoadFromFile('MyBitmap.bmp');
    Form1.Canvas.Brush.Bitmap := Bitmap;
    Form1.Canvas.FillRect(Rect(0,0,100,100));
  finally
    Form1.Canvas.Brush.Bitmap := nil;
    Bitmap.Free;
  end;
end;
```

Remarque Le pinceau n'assume pas la possession d'un objet bitmap affecté à sa propriété *Bitmap*. Vous devez vous assurer que l'objet Bitmap reste valide pendant la durée de vie du pinceau, après quoi vous devez vous-même libérer l'objet Bitmap.

Lecture et définition de pixels

Chaque canevas a une propriété *Pixels* indexée qui représente les points de couleur constituant l'image sur le canevas. Vous devrez rarement accéder directement à la propriété *Pixels*, sauf si vous voulez connaître ou modifier la couleur d'un pixel particulier.

Remarque La définition de pixels individuels prend beaucoup plus de temps que les opérations graphiques sur des zones particulières. N'utilisez pas la propriété tableau *Pixel* pour accéder aux pixels d'une image dans un tableau général. Pour un accès performant aux pixels d'une image, voir la propriété *TBitmap.ScanLine*.

Utilisation des méthodes du canevas pour dessiner des objets graphiques

Cette section montre comment utiliser certaines méthodes pour dessiner des objets graphiques. Elle traite des sujets suivants :

- Dessin de lignes et de polylignes
- Dessin de formes
- Dessin de rectangles arrondis
- Dessin de polygones

Dessin de lignes et de polylignes

Un canevas peut dessiner des lignes droites et des polylignes (ou lignes brisées). Une ligne droite est une ligne de pixels reliant deux points. Une polyligne est une chaîne de lignes droites, reliées bout à bout. Le canevas dessine toutes les lignes en utilisant son crayon.

Dessin de lignes

Pour dessiner une ligne droite sur un canevas, utilisez la méthode *LineTo* du canevas.

La méthode *LineTo* dessine une ligne partant de la position en cours du crayon et allant au point spécifié, et fait du point d'arrivée de la ligne la position en cours. Le canevas dessine la ligne en utilisant son crayon.

Par exemple, la méthode suivante dessine des lignes diagonales qui se croisent sur une fiche, chaque fois que la fiche est peinte :

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 with Canvas do
 begin
   MoveTo(0, 0);
   LineTo(ClientWidth, ClientHeight);
```

```
    MoveTo(0, ClientHeight);
    LineTo(ClientWidth, 0);
  end;
end;
```

Dessin de polylignes

En plus des lignes individuelles, le canevas peut dessiner des polylignes, qui sont des groupes composés d'un nombre quelconque de segments de ligne reliés entre eux.

Pour dessiner une polyligne sur un canevas, appelez la méthode *Polyline* du canevas.

Le paramètre passé à la méthode *Polyline est un tableau de* points. Imaginez qu'une polyligne réalise une méthode *MoveTo* sur le premier point et une méthode *LineTo* sur chaque point successif. Si vous voulez dessiner plusieurs lignes, vous devez savoir que *Polyline* est plus rapide que la méthode *MoveTo* et que la méthode *LineTo*, car elle élimine un certain nombre d'appels supplémentaires.

La méthode suivante, par exemple, dessine un losange dans une fiche :

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  with Canvas do
    Polyline([Point(0, 0), Point(50, 0), Point(75, 50), Point(25, 50), Point(0, 0)]);
end;
```

Cet exemple montre bien les possibilités de Delphi de créer un paramètre tableau ouvert à la volée. Il est possible de passer n'importe quel tableau de points, mais une manière simple de construire un tableau facilement consiste à mettre ses éléments entre crochets et de passer le tout en paramètre. Pour plus d'informations, voir l'aide en ligne.

Dessin de formes

Les canevas disposent de méthodes vous permettant de dessiner différents types de formes. Le canevas dessine le pourtour d'une forme avec son crayon, puis remplit l'intérieur avec son pinceau. La ligne qui définit la bordure de la forme est déterminée par l'objet *Pen* en cours.

Cette section couvre :

- Dessin de rectangle et d'ellipses
- Dessin de rectangles à coins arrondis
- Dessin de polygones

Dessin de rectangles et d'ellipses

Pour dessiner un rectangle ou une ellipse sur un canevas, appelez la méthode *Rectangle* ou la méthode *Ellipse* du canevas, en transmettant les coordonnées des limites d'un rectangle.

La méthode *Rectangle* dessine le rectangle ; *Ellipse* dessine une ellipse qui touche tous les côtés du rectangle.

La méthode suivante dessine un rectangle remplissant le quart supérieur gauche d'une fiche, puis dessine une ellipse sur la même zone :

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Rectangle(0, 0, ClientWidth div 2, ClientHeight div 2);
 Canvas.Ellipse(0, 0, ClientWidth div 2, ClientHeight div 2);
end;
```

Dessin de rectangles à coins arrondis

Pour dessiner un rectangle à coins arrondis sur un canevas, appelez la méthode *RoundRect* du canevas.

Les quatre premiers paramètres transmis à *RoundRect* sont les limites d'un rectangle, comme pour la méthode *Rectangle* ou la méthode *Ellipse. RoundRect* prend deux paramètres supplémentaires qui indiquent comment dessiner les coins arrondis.

La méthode suivante, par exemple, dessine un rectangle à coins arrondis dans le quart supérieur de la fiche, en arrondissant les coins en arcs d'un cercle de 10 pixels de diamètre :

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.RoundRect(0, 0, ClientWidth div 2, ClientHeight div 2, 10, 10);
end;
```

Dessin de polygones

Pour dessiner, sur un canevas, un polygone ayant un nombre quelconque de côtés, appelez la méthode *Polygon* du canevas.

Polygon prend un tableau de points comme seul paramètre et relie les points avec le crayon, puis relie le dernier point au premier de façon à fermer le polygone. Après avoir dessiné les lignes, *Polygon* utilise le pinceau pour remplir la zone interne au polygone.

Le code suivant dessine un triangle rectangle dans la moitié inférieure gauche de la fiche :

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Polygon([Point(0, 0), Point(0, ClientHeight),
   Point(ClientWidth, ClientHeight)]);
end;
```

Gestion de plusieurs objets de dessin dans votre application

Différentes méthodes de dessin (rectangle, forme, ligne, etc.) sont typiquement disponibles sur la barre d'outils et le volet de boutons. Les applications peuvent répondre à des clics sur des turboboutons de façon à définir les objets de dessin voulu. Cette section décrit comment :

- Faire le suivi de l'outil de dessin à utiliser

- Changer d'outil de dessin en utilisant des turboboutons
- Utiliser des outils de dessin

Faire le suivi de l'outil de dessin à utiliser

Une application graphique doit pouvoir connaître à tout moment le type d'outil de dessin (une ligne, un rectangle, une ellipse ou un rectangle arrondi, par exemple) que l'utilisateur veut utiliser. Vous pouvez affecter des nombres à chaque type d'outil, mais vous devrez alors vous rappeler de la signification de chaque nombre. Vous pouvez rendre cette technique plus simple en affectant un nom de constante mnémonique à chaque nombre, mais le code sera alors incapable de distinguer les nombres se trouvant dans la bonne plage et ceux du bon type. Par chance, le Pascal Objet fournit un moyen de gérer ces deux points faibles. Vous pouvez déclarer un type énuméré.

Un type énuméré est juste un moyen rapide pour affecter des valeurs séquentielles à des constantes. Depuis qu'il s'agit aussi d'une déclaration de type, vous pouvez utiliser la vérification de type du Pascal Objet pour vous assurer que vous n'affectez que ces valeurs spécifiques.

Pour déclarer un type énuméré, utilisez le mot réservé type, suivi par un identificateur de type, du signe égal et des identificateurs pour les valeurs mis entre parenthèses et séparés par des virgules.

Par exemple, le code suivant déclare un type énuméré pour tous les outils de dessin de l'application graphique :

```
type
  TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
```

Par convention, les identificateurs de type commencent par la lettre *T*, et les constantes similaires (celles constituant le type énuméré) commencent par un même préfixe de deux caractères (comme ici *dt* pour "drawing tool").La déclaration du type TDrawingTool est équivalente à la déclaration d'un groupe de constantes :

```
const
  dtLine = 0;
  dtRectangle = 1;
  dtEllipse = 2;
  dtRoundRect = 3;
```

La principale différence est qu'en déclarant un type énuméré, vous affectez des constantes et pas seulement des valeurs, mais aussi un type qui permet d'utiliser la vérification de type du Pascal Objet pour vous prémunir de nombreuses erreurs. Une variable de type TDrawingTool peut être affectée seulement par une des constantes dtLine..dtRoundRect. Toute tentative d'affectation d'un autre nombre (même de la portée 0..3) génèrera une erreur de compilation.

Dans le code suivant, un champ ajouté à une fiche fera le suivi de l'outil de dessin de la fiche :

```
type
  TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
 TForm1 = class(TForm)
    ...{ déclarations de méthode}
 public
    Drawing: Boolean;
   Origin, MovePt: TPoint;
   DrawingTool: TDrawingTool;{ champ pour l'outil en cours }
 end;
```

Changement d'outil en utilisant un turbobouton

Chaque outil de dessin de votre application doit avoir un gestionnaire pour son événement *OnClick*. Supposons que votre application ait une barre d'outils comportant un bouton pour chacun des quatre outils de dessin : ligne, rectangle, ellipse et rectangle arrondi. Vous attacherez les gestionnaires suivants aux événements *OnClick* des quatre boutons, en affectant à *DrawingTool* la valeur correspondant à chaque outil :

```
procedure TForm1.LineButtonClick(Sender: TObject);{ LineButton }
begin
  DrawingTool := dtLine;
end;

procedure TForm1.RectangleButtonClick(Sender: TObject);{ RectangleButton }
begin
  DrawingTool := dtRectangle;
end;

procedure TForm1.EllipseButtonClick(Sender: TObject);{ EllipseButton }
begin
  DrawingTool := dtEllipse;
end;

procedure TForm1.RoundedRectButtonClick(Sender: TObject);{ RoundRectButton }
begin
  DrawingTool := dtRoundRect;
end;
```

Utilisation des outils de dessin

Vous savez maintenant spécifier l'outil à utiliser. Il vous reste à indiquer comment dessiner les différentes formes. Les seules méthodes réalisant des dessins sont les gestionnaires de souris (déplacement de souris et relâchement de bouton de souris), et le seul code de dessin dessine des lignes, quel que soit l'outil sélectionné.

Pour utiliser les différents outils de dessin, votre code doit spécifier comment dessiner selon l'outil sélectionné. Vous devez ajouter l'instruction au gestionnaire d'événement de chaque outil.

Cette section explique comment :

- dessiner des formes
- partager du code entre les gestionnaires d'événements

Dessiner des formes

Dessiner des formes est aussi simple que dessiner des lignes. Une seule instruction suffit. Vous n'avez besoin que des coordonnées.

Voici réécrit le gestionnaire de l'événement *OnMouseUp* qui dessine des formes pour les quatre outils :

```
procedure TForm1.FormMouseUp(Sender: TObject; Button TMouseButton; Shift: TShiftState;
                             X,Y: Integer);
begin
case DrawingTool of
    dtLine:
      begin
        Canvas.MoveTo(Origin.X, Origin.Y);
       Canvas.LineTo(X, Y)
      end;
   dtRectangle: Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
   dtEllipse: Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
   dtRoundRect: Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                  (Origin.X - X) div 2, (Origin.Y - Y) div 2);
  end;
 Drawing := False;
end;
```

Il est également nécessaire de modifier le gestionnaire de *OnMouseMove* pour dessiner des formes :

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.Pen.Mode := pmNotXor;
   case DrawingTool of
      dtLine: begin
                Canvas.MoveTo(Origin.X, Origin.Y);
               Canvas.LineTo(MovePt.X, MovePt.Y);
               Canvas.MoveTo(Origin.X, Origin.Y);
               Canvas.LineTo(X, Y);
              end;
     dtRectangle: begin
                    Canvas.Rectangle(Origin.X, Origin.Y, MovePt.X, MovePt.Y);
                   Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
                  end;
     dtEllipse: begin
                   Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
                  Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
                 end;
     dtRoundRect: begin
                    Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                      (Origin.X - X) div 2, (Origin.Y - Y) div 2);
```

```
                    Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                       (Origin.X - X) div 2, (Origin.Y - Y) div 2);
                  end;
    end;
   MovePt := Point(X, Y);
  end;
 Canvas.Pen.Mode := pmCopy;
end;
```

En principe, tout le code répétitif de l'exemple précédent devrait être dans une routine séparée. La section suivante présente le code relatif au dessin des formes dans une seule routine pouvant être appelée par tous les gestionnaires d'événements de souris.

Partage de code entre plusieurs gestionnaires d'événements

Chaque fois que plusieurs gestionnaires d'événements utilisent le même code, vous rendez l'application plus efficace en plaçant le code répété dans une méthode partagée par les gestionnaires d'événements.

Pour ajouter une méthode à une fiche,

1. Ajoutez la déclaration de la méthode à l'objet fiche.

   Il est possible d'ajouter la déclaration dans les sections **public** ou **private**, à la fin des déclarations de l'objet fiche. Si le code partage uniquement les détails de la manipulation de certains événements, il est préférable de créer une méthode partagée **private**.

2. Ecrivez l'implémentation de la méthode dans la partie implementation de l'unité de la fiche.

L'en-tête de l'implémentation de la méthode doit correspondre exactement à la déclaration, les mêmes paramètres apparaissant dans le même ordre.

Le code suivant ajoute à la fiche une méthode appelée *DrawShape* et l'appelle depuis chacun des gestionnaires. D'abord, la déclaration de *DrawShape* est ajoutée à la déclaration de l'objet fiche :

```
type
  TForm1 = class(TForm)
    ...{ champs et méthodes déclarés ici}
 public
    { déclarations publiques }
   procedure DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
  end;
```

Ensuite l'implémentation de *DrawShape* est écrite dans la partie implementation de l'unité :

```
implementation
{$R *.FRM}
...{ autres implémentations de méthode omises pour plus de clarté }
procedure TForm1.DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
begin
  with Canvas do
  begin
```

```
    Pen.Mode := AMode;
   case DrawingTool of
      dtLine:
        begin
          MoveTo(TopLeft.X, TopLeft.Y);
         LineTo(BottomRight.X, BottomRight.Y);
        end;
     dtRectangle: Rectangle(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
     dtEllipse: Ellipse(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
     dtRoundRect: RoundRect(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y,
        (TopLeft.X - BottomRight.X) div 2, (TopLeft.Y - BottomRight.Y) div 2);
    end;
  end;
end;
```

Les autres gestionnaires d'événements sont modifiés pour appeler *DrawShape*.

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  DrawShape(Origin, Point(X, Y), pmCopy);{ dessine la forme finale }
 Drawing := False;
end;
procedure TForm1.FormMouseMove(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    DrawShape(Origin, MovePt, pmNotXor);{ efface la forme précédente }
   MovePt := Point(X, Y);{ enregistre le point en cours }
   DrawShape(Origin, MovePt, pmNotXor);{ dessine la forme en cours }
  end;
end;
```

Dessiner sur un graphique

Vous n'avez pas besoin de composant pour manipuler les objets graphiques de votre application. Vous pouvez construire des objets graphiques, dessiner sur eux, les sauvegarder et les détruire sans même dessiner sur l'écran. En fait, il est rare qu'une application dessine directement sur une fiche. Le plus souvent, une application doit dessiner sur un graphique. Elle utilise ensuite un composant contrôle image pour afficher le graphique sur une fiche.

Une fois les opérations de dessin de l'application reportées sur le graphique du contrôle image, il est facile d'y ajouter les fonctionnalités relatives à l'impression, aux opérations sur le presse-papiers, à l'ouverture et à l'enregistrement des objets graphiques. Les objets graphiques peuvent être des fichiers bitmap, des dessins, des icônes ou toute autre classe graphique installée en tant que graphique jpeg.

Remarque Etant donné que vous dessinez sur une image hors écran, comme un canevas *Tbitmap*, l'image n'apparaît pas tant qu'un contrôle effectue la copie d'un bitmap sur le canevas du contrôle. En d'autres mots, lorsque vous dessinez des bitmaps et les affectez à un contrôle image, l'image n'apparaît que si le contrôle a la

possibilité de traiter son message. En revanche, si vous dessinez directement sur la propriété *Canvas* d'un contrôle, l'objet image apparaît immédiatement.

Création de graphiques défilables

Le graphique ne doit pas être de la même taille que la fiche : il doit être soit plus petit, soit plus grand. En ajoutant un contrôle boîte de défilement sur la fiche et en plaçant une image graphique à l'intérieur, vous pouvez afficher des graphiques beaucoup plus grands que la fiche et même plus grands que l'écran. Pour ajouter un graphique défilable, vous devez commencer par ajouter un composant *TScrollbox* puis ajouter ensuite le contrôle image.

Ajout d'un contrôle image

Un contrôle image est un composant conteneur qui vous permet d'afficher vos objets bitmap. Un contrôle image peut être utilisé pour contenir un bitmap qui n'est pas nécessairement affiché en permanence, ou un bitmap dont l'application a besoin pour générer d'autres images.

Remarque Pour des informations sur l'utilisation des graphiques dans des contrôles, voir "Ajout de graphiques à des contrôles" à la page 7-13.

Positionnement du contrôle

Un contrôle image peut être placé n'importe où dans une fiche. Pour tirer le meilleur parti de la capacité d'un contrôle image à ajuster sa taille sur celle de son image, le seul point à définir est le coin supérieur gauche du contrôle. Si le contrôle image sert d'emplacement non visible pour un bitmap, il peut être placé n'importe où dans la fiche, comme un composant non visuel.

Si vous placez le contrôle image en le mettant à l'intérieur de la boîte de défilement déjà installée dans la zone client de la fiche, vous serez sûr que la boîte de défilement affichera des barres de défilement pour permettre l'accès aux parties du dessin n'apparaissant pas à l'écran. Définissez ensuite les propriétés du contrôle image.

Définition de la taille initiale du bitmap

Lorsqu'un contrôle image est ajouté, il n'existe qu'en tant que conteneur. La propriété *Picture* du contrôle image peut être définie en mode conception de façon à contenir un graphique statique. Mais, le contrôle peut également charger l'image depuis un fichier pendant l'exécution, comme décrit dans la section "Chargement et enregistrement de fichiers graphiques" à la page 8-20.

Pour créer un bitmap vide au démarrage de l'application,

1. Attachez un gestionnaire à l'événement *OnCreate* de la fiche contenant l'image.
2. Créez un objet bitmap, et affectez-le à la propriété *Picture.Graphic* du contrôle image.

Dans cet exemple, l'image est dans *Form1*, la fiche principale de l'application. Le code attache donc un gestionnaire à l'événement *OnCreate* de *Form1* :

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Bitmap: TBitmap;{ variable temporaire pour contenir le bitmap }
begin
  Bitmap := TBitmap.Create;{ construire l'objet bitmap }
  Bitmap.Width := 200;{ affecter la largeur initiale... }
  Bitmap.Height := 200;{ ...et la hauteur initiale }
  Image.Picture.Graphic := Bitmap;{ affecter le bitmap au contrôle image }
  Bitmap.Free; {Nous en avons fini avec le bitmap, libérons-le }
end;
```

L'affectation du bitmap à la propriété *Graphic* de l'image copie le bitmap dans l'objet *Picture*. Mais, l'objet *Picture* ne devient pas propriétaire du bitmap, c'est pourquoi, après avoir effectué cette affectation, vous devez le libérer.

Si vous exécutez l'application maintenant, la zone client de la fiche apparaît comme une zone blanche représentant le bitmap. Si vous redimensionnez la fenêtre de sorte que la zone client ne puisse afficher toute l'image, la boîte de défilement affiche automatiquement des barres de défilement pour permettre la visualisation du reste de l'image. Mais si vous essayez de dessiner dans l'image, rien n'apparaît : l'application dessine toujours dans la fiche qui est derrière l'image et la boîte de défilement.

Dessiner sur un bitmap

Pour dessiner sur un bitmap, utilisez le canevas du contrôle image et attachez les gestionnaires d'événements de souris aux événements appropriés du contrôle image. Typiquement, vous devriez utiliser des opérations sur des régions (rectangles, polylignes et ainsi de suite). Ce sont des méthodes rapides et efficaces pour dessiner.

Un moyen efficace de dessiner des images lorsque vous avez besoin d'accéder de manière individuelle aux pixels est d'utiliser la propriété *ScanLine* du bitmap. Pour une utilisation plus générale, vous pouvez définir un format de pixels de 24 bits et traiter le pointeur renvoyé par *ScanLine* comme un tableau de couleurs RVB. Vous devrez sinon connaître le format natif de la propriété *ScanLine*.

Cet exemple explique comment utiliser *ScanLine* pour extraire des pixels ligne par ligne.

```
procedure TForm1.Button1Click(Sender: TObject);
// Cet exemple montre un dessin effectué directement sur le bitmap
var
  x,y : integer;
  Bitmap : TBitmap;
  P : PByteArray;
begin
  Bitmap := TBitmap.create;
  try
    Bitmap.LoadFromFile('C:\Program Files\Borland\Delphi 4\Images\Splash\256color\
factory.bmp');
    for y := 0 to Bitmap.height -1 do
```

```
    begin
      P := Bitmap.ScanLine[y];
      for x := 0 to Bitmap.width -1 do
        P[x] := y;
    end;
  canvas.draw(0,0,Bitmap);
  finally
    Bitmap.free;
  end;
end;
```

Chargement et enregistrement de fichiers graphiques

Des images graphiques n'existant que pour la durée de l'exécution d'une application sont d'un intérêt limité. Le plus souvent, la même image est utilisée à chaque exécution, ou bien l'image créée est enregistrée pour une utilisation ultérieure. Le composant image facilite le chargement d'une image depuis un fichier et son enregistrement.

Les composants que vous utilisez pour charger, sauvegarder et remplacer des images graphiques supportent la plupart des formats de graphiques dont les fichiers bitmap, les métafichiers, les glyphes, et ainsi de suite. Ils supportent aussi les classes graphiques installables.

Le mécanisme d'ouverture et d'enregistrement des fichiers graphiques est semblable à celui utilisé pour les autres fichiers et est décrit dans les sections suivantes :

- Charger une image depuis un fichier
- Enregistrer une image dans un fichier
- Remplacer l'image

Chargement d'une image depuis un fichier

Votre application doit fournir la possibilité de charger une image depuis un fichier si votre application a besoin de modifier l'image ou si vous voulez la stocker à l'extérieur de l'application afin qu'un autre utilisateur ou une autre application puisse la modifier.

Pour charger un fichier graphique dans un contrôle image, appelez la méthode *LoadFromFile* de l'objet *Picture* du contrôle image.

Le code suivant extrait un nom de fichier dans une boîte de dialogue d'ouverture de fichiers graphiques, et charge ensuite ce fichier dans un contrôle image nommé *Image* :

```
procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then
  begin
    CurrentFile := OpenPictureDialog1.FileName;
   Image.Picture.LoadFromFile(CurrentFile);
  end;
end;
```

Enregistrement d'une image dans un fichier

L'objet *Picture* peut charger et enregistrer des graphiques sous divers formats. Vous pouvez créer et recenser vos propres formats de fichiers graphiques afin que les objets image puissent également les enregistrer et les stocker.

Pour enregistrer le contenu d'un contrôle image dans un fichier, appelez la méthode *SaveToFile* de l'objet *Picture* du contrôle image.

La méthode *SaveToFile* nécessite de spécifier le nom du fichier de sauvegarde. L'image est nouvellement créée et n'a pas encore de nom de fichier, ou bien l'image existe déjà mais l'utilisateur veut l'enregistrer dans un fichier différent. Dans l'un ou l'autre cas, l'application doit demander à l'utilisateur un nom de fichier avant l'enregistrement, comme le montre la section suivante.

Les deux gestionnaires d'événements suivants, attachés respectivement aux éléments de menu Fichier | Enregistrer et Fichier | Enregistrer sous, gèrent l'enregistrement des fichiers ayant déjà un nom, l'enregistrement des fichiers n'ayant pas de nom et l'enregistrement des fichiers sous un nouveau nom.

```
procedure TForm1.Save1Click(Sender: TObject);
begin
  if CurrentFile <> '' then
    Image.Picture.SaveToFile(CurrentFile){ enregistrer si déjà nommé }
 else SaveAs1Click(Sender);{ sinon, obtenir un nom }
end;
procedure TForm1.Saveas1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then{ obtenir un nom de fichier }
  begin
    CurrentFile := SaveDialog1.FileName;{ enregistrer le nom spécifié par l'utilisateur }
   Save1Click(Sender);{ puis enregistrer normalement }
  end;
end;
```

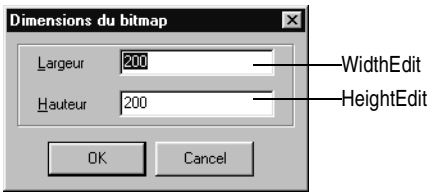
Remplacement de l'image

Il est possible à tout moment de remplacer l'image d'un contrôle image. Si un nouveau graphique est affecté à l'image ayant déjà un graphique, le nouveau graphique remplace l'ancien.

Pour remplacer l'image contenue dans un contrôle image, affectez un nouveau graphique à l'objet *Picture* du contrôle image.

La création d'un nouveau graphique passe par le même processus que la création du premier graphique (voir la section "Définition de la taille initiale du bitmap" à la page 8-18), mais il faut également permettre à l'utilisateur de choisir une taille différente de celle utilisée par défaut pour le graphique initial. Un moyen simple de proposer une telle option est de présenter une boîte de dialogue comme celle de la figure suivante.

Figure 8.1 Boîte de dialogue Dimension bitmap de l'unité BMPDlg



Cette boîte de dialogue particulière est créée dans l'unité *BMPDlg* incluse dans le projet *GraphEx* (répertoire EXAMPLES\DOC\GRAPHEX).

Cette boîte de dialogue étant dans votre projet, ajoutez-la à la clause uses de l'unité de votre fiche principale. Vous pouvez ensuite attacher un gestionnaire à l'événement *OnClick de* l'élément de menu Fichier|Nouveau. Voici un exemple :

```
procedure TForm1.New1Click(Sender: TObject);
var
  Bitmap: TBitmap;{ variable temporaire pour le nouveau bitmap }
begin
  with NewBMPForm do
  begin
    ActiveControl := WidthEdit;{ focalisation sur le champ largeur }
   WidthEdit.Text := IntToStr(Image.Picture.Graphic.Width);{ dimensions en cours... }
   HeightEdit.Text := IntToStr(Image.Picture.Graphic.Height);{ ...par défaut }
   if ShowModal <> idCancel then{ continuer si l'utilisateur n'annule pas la boîte de
                                  dialogue }
    begin
      Bitmap := TBitmap.Create;{ créer un objet bitmap }
     Bitmap.Width := StrToInt(WidthEdit.Text);{ utiliser la largeur spécifiée }
     Bitmap.Height := StrToInt(HeightEdit.Text);{ utiliser la hauteur spécifiée }
     Image.Picture.Graphic := Bitmap;{ remplacer le graphique par le nouveau bitmap }
     CurrentFile := '';{ indique le fichier non nommé }
     Bitmap.Free;
    end;
  end;
end;
```

Remarque L'affectation d'un nouveau bitmap à la propriété *Graphic* de l'objet *Picture* oblige l'objet *Picture* à copier le nouveau graphique, mais il ne devient pas son propriétaire. L'objet *Picture* maintient son propre objet graphique interne. A cause de cela, le code précédent libère l'objet bitmap après l'affectation.

Utilisation du presse-papiers avec les graphiques

Vous pouvez utiliser le presse-papiers de Windows pour copier et coller des graphiques dans les applications ou pour échanger des graphiques avec d'autres applications. L'objet *Clipboard* de la VCL facilite la gestion de différents types d'informations, y compris les graphiques.

Avant d'utiliser l'objet *Clipboard* dans une application, il faut ajouter l'unité Clipbrd (QClipbrd dans CLX) à la clause **uses** de toute unité devant accéder aux données du presse-papiers.

Pour les applications multiplates-formes, les données, qui sont stockées dans le presse-papiers lorsque vous utilisez CLX, sont stockées en tant que type mime en association avec un objet *TStream*. CLX fournit le source mime prédéfini suivant et des constantes chaîne de type mime pour les objets CLX ci-dessous :

- TBitmap = 'image/delphi.bitmap'
- TComponent = 'application/delphi.component'
- TPicture = 'image/delphi.picture'
- TDrawing = 'image/delphi.drawing'

Copier des graphiques dans le presse-papiers

Toute image graphique, y compris le contenu d'un contrôle image, peut être copiée dans le presse-papiers. Une fois placée dans le presse-papiers, l'image est disponible pour toutes les applications.

Pour copier une image dans le presse-papiers, affectez l'image à l'objet *Clipboard* en utilisant la méthode *Assign*.

Le code suivant montre comment copier dans le presse-papiers l'image d'un contrôle image nommé *Image* en réponse au choix d'un élément de menu Edition | Copier :

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
  Clipboard.Assign(Image.Picture)
end.
```

Couper des graphiques dans le presse-papiers

L'opération qui consiste à couper un graphique dans le presse-papiers est semblable à la copie, sauf que le graphique est supprimé de la source.

Pour couper un graphique dans le presse-papiers, commencez par le copier dans le presse-papiers, puis supprimez l'original.

Lorsque vous coupez un graphique, la seule question est de savoir comment montrer que l'image originale a été effacée. La solution classique consiste à mettre la région à blanc, comme dans le code suivant qui attache un gestionnaire à l'événement *OnClick* d'un élément de menu Edition | Couper :

```
procedure TForm1.Cut1Click(Sender: TObject);
var
  ARect: TRect;
begin
  Copy1Click(Sender);{ copier l'image dans le presse-papiers }
 with Image.Canvas do
  begin
    CopyMode := cmWhiteness;
   ARect := Rect(0, 0, Image.Width, Image.Height);{ obtenir le rectangle bitmap}
   CopyRect(ARect, Image.Canvas, ARect);{ copier le bitmap sur lui-même }
   CopyMode := cmSrcCopy;{ restaurer le mode normal }
  end;
end;
```

Coller des graphiques depuis le presse-papiers

Si le presse-papiers contient un graphique bitmap, il est possible de le coller dans tout objet image, y compris les contrôles image et la surface d'une fiche.

Pour coller un graphique depuis le presse-papiers,

1 Appelez la méthode *HasFormat* (si vous utilisez la VCL) ou la méthode *Provides* (si vous utilisez CLX) du presse-papiers pour savoir s'il contient un graphique.

   HasFormat (ou *Provides* dans CLX) est une fonction booléenne. Elle renvoie *True* si le presse-papiers contient un élément du type spécifié par le paramètre. Pour tester la présence d'un graphique, passez le paramètre *CF_BITMAP* sur la plate-forme Windows. Passez *SDelphiBitmap* dans les applications multiplates-formes.

2 Affectez le presse-papiers à la destination.

Ce code montre comment coller une image depuis le presse-papiers dans un contrôle image en réponse à un clic sur un élément de menu Edition | Coller :

```
procedure TForm1.PasteButtonClick(Sender: TObject);
var
   Bitmap: TBitmap;
begin
   if Clipboard.HasFormat(CF_BITMAP) then { y a-t-il un bitmap dans le presse-papiers
                                            Windows ? )
   begin
      Image.Picture.Bitmap.Assign(Clipboard);
   end;
end;
```

Le même exemple dans CLX pour le développement multiplate-forme doit ressembler à ceci :

```
procedure TForm1.PasteButtonClick(Sender: TObject);
var
  Bitmap: TBitmap;
begin
 if Clipboard.Provides(SDelphiBitmap) then { y a-t-il un bitmap dans le presse-papiers ? )
 begin
   Image1.Picture.Bitmap.Assign(Clipboard);
 end;
end;
```

Le graphique du presse-papiers peut provenir de cette application ou y avoir été copié par une autre application, comme Microsoft Paint. Dans ce cas, il n'est pas nécessaire de vérifier le format du presse-papiers, car le menu Coller serait indisponible si le presse-papiers contenait un format non supporté.

Techniques de dessin dans une application

Cet exemple explique les détails relatifs à l'implémentation de l'effet "rubber banding" dans une application graphique qui suit les mouvements de la souris au fur et à mesure que l'utilisateur dessine un graphique en mode exécution. Le code qui suit provient d'une application exemple située dans le répertoire Demos\DOC\Graphex. Cette application dessine des lignes et des formes sur le canevas d'une fenêtre en réponse à des cliquer-glisser : l'appui sur le bouton de la souris commence le dessin, le relâchement du bouton termine le dessin.

Pour commencer, cet exemple de code montre comment dessiner sur la surface d'une fiche principale. Les exemples ultérieurs expliquent comment dessiner sur un bitmap.

Les rubriques suivantes décrivent l'exemple :

- Comment répondre à la souris
- Ajout d'un champ à un objet fiche pour faire le suivi des actions de la souris
- Amélioration du dessin de ligne

Répondre à la souris

Votre application peut répondre aux actions de la souris : enfoncement du bouton de la souris, déplacement de la souris et relâchement du bouton de la souris. Elle peut aussi répondre à un clic (un appui suivi d'un relâchement, sans déplacer la souris) qui peut être généré par certaines frappes de touches (comme l'appui sur la touche *Entrée* dans une boîte de dialogue modale).

Cette section traite des sujets suivants :

- Qu'y a-t'il dans un événement de souris ?
- Réponse à l'action bouton de souris enfoncé
- Réponse à l'action bouton de souris relâché
- Réponse au déplacement de la souris

Qu'y a-t'il dans un événement de souris ?

La VCL possède trois événements de souris : l'événement *OnMouseDown*, l'événement *OnMouseMove* et l'événement *OnMouseUp*.

Lorsqu'une application détecte une action de la souris, elle appelle le gestionnaire que vous avez défini pour l'événement correspondant, en lui transmettant cinq paramètres. Les informations contenues dans ces paramètres permettent de personnaliser la réponse aux événements. Il s'agit des paramètres suivants :

Tableau 8.4 Paramètres des événements de souris

Paramètre	Signification
Sender	L'objet ayant détecté l'action de souris.
Button	Indique le bouton de la souris impliqué : *mbLeft*, *mbMiddle* ou *mbRight*.
Shift	Indique l'état des touches *Alt*, *Ctrl* et *Maj* au moment de l'action de souris.
X, *Y*	Les coordonnées de l'endroit où l'événement a eu lieu.

La plupart du temps, les informations essentielles pour le gestionnaire d'un événement souris sont les coordonnées mais, dans certains cas, il est utile de tester le paramètre *Button* pour déterminer quel bouton de la souris a provoqué l'événement.

Remarque Delphi utilise le même critère que Microsoft Windows pour déterminer le bouton enfoncé. Aussi, si vous avez interverti les boutons "primaire" et "secondaire" par défaut de la souris (pour que le bouton droit de la souris soit le bouton primaire), un clic du bouton primaire (droit) entraînera une valeur *mbLeft* pour le paramètre *Button*.

Réponse à l'action bouton de souris enfoncé

Lorsque l'utilisateur appuie sur un bouton de la souris, un événement *OnMouseDown* est adressé à l'objet situé en dessous du pointeur de la souris. L'objet peut alors répondre à l'événement.

Pour répondre à une action bouton de souris enfoncé, attachez un gestionnaire à l'événement *OnMouseDown*.

La VCL génère un gestionnaire vide pour l'événement souris enfoncée se produisant sur la fiche :

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
   Shift: TShiftState; X, Y: Integer);
begin
end;
```

Exemple : réponse à l'action bouton de souris enfoncé

Le code suivant affiche la chaîne 'Ici!' sur une fiche à l'emplacement du clic de la souris :

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
   Shift: TShiftState; X, Y: Integer);
begin
 Canvas.TextOut(X, Y, 'Ici!');{ écrire du texte aux coordonnées (X, Y) }
end;
```

Lorsque l'application est exécutée, le curseur étant sur la fiche, tout appui sur le bouton de la souris fera apparaître la chaîne "Ici!" au point cliqué. Ce code définit la position de dessin en cours par les coordonnées du point où l'utilisateur a enfoncé le bouton de la souris :

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
   Shift: TShiftState; X, Y: Integer);
begin
   Canvas.MoveTo(X, Y);{ définir la position du crayon }
end;
```

Désormais, l'enfoncement du bouton de la souris définit la position du crayon et initialise ainsi le point de départ de la ligne. Pour dessiner une ligne jusqu'au point où l'utilisateur a relâché le bouton, vous devez répondre à l'événement bouton de souris relâché.

Réponse à l'action bouton de souris relâché

Un événement *OnMouseUp* se produit dès que l'utilisateur relâche le bouton de la souris. L'événement est, en général, adressé à l'objet au-dessus duquel la souris se trouvait lorsque le bouton a été enfoncé, qui n'est pas nécessairement celui au-dessus duquel le bouton de la souris est relâché. Cela permet, par exemple, de dessiner une ligne comme si elle s'étendait au-delà des bords de la fiche.

Pour répondre à une action bouton de souris relâché, définissez le gestionnaire de l'événement *OnMouseUp*.

Voici un gestionnaire *OnMouseUp* qui dessine une ligne jusqu'au point où le bouton de la souris a été relâché :

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
   Shift: TShiftState; X, Y: Integer);
begin
   Canvas.LineTo(X, Y);{ dessiner une ligne de PenPos à (X, Y) }
end;
```

Le code permet à l'utilisateur de dessiner des lignes en cliquant, en déplaçant la souris, puis en relâchant le bouton. Dans ce cas, l'utilisateur ne voit pas la ligne tant qu'il ne relâche pas le bouton de la souris.

Réponse au déplacement de la souris

Un événement *OnMouseMove* se produit périodiquement lorsque l'utilisateur déplace la souris. L'événement est adressé à l'objet qui était sous le pointeur de la souris lorsque l'utilisateur a enfoncé le bouton. Cela vous permet de fournir un retour d'informations à l'utilisateur en dessinant des lignes temporaires au fur et à mesure que la souris est déplacée.

Pour répondre aux déplacements de la souris, définissez un gestionnaire pour l'événement *OnMouseMove* de la fiche. Cet exemple utilise les événements déplacement de la souris pour dessiner sur la fiche des formes intermédiaires pendant que l'utilisateur maintient enfoncé le bouton de la souris, offrant ainsi à l'utilisateur un aperçu de ce qu'il obtiendra. Le gestionnaire de l'événement

OnMouseMove dessine une ligne dans la fiche à l'emplacement de l'événement *OnMouseMove* :

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
   Shift: TShiftState; X, Y: Integer);
begin
   Canvas.LineTo(X, Y);{ dessiner la ligne à la position en cours }
end;
```

Avec ce code, le dessin suit le déplacement de la souris sur la fiche, avant même que le bouton de la souris ne soit enfoncé.

Les événements déplacement de la souris se produisent, même lorsque le bouton de la souris n'a pas été enfoncé.

Pour déterminer si un bouton de la souris est enfoncé, il est nécessaire d'ajouter un objet champ à l'objet fiche.

Ajout d'un champ à un objet fiche

Lorsque vous ajoutez un composant à une fiche, Delphi ajoute également un champ représentant ce composant dans l'objet fiche. Il est ensuite possible de faire référence à ce composant par le nom de son champ. Vous pouvez également ajouter vos propres champs en modifiant la déclaration de type dans le fichier d'en-tête de l'unité de la fiche.

Dans l'exemple suivant, il faut que la fiche détermine si l'utilisateur a enfoncé le bouton de la souris. Pour cela, un champ booléen a été ajouté dont la valeur est définie lorsque l'utilisateur enfonce le bouton de la souris.

Pour ajouter un champ à un objet, modifiez la définition de type de l'objet, en spécifiant l'identificateur du champ et son type après la directive **public** à la fin de la déclaration.

Delphi est "propriétaire" de toutes les déclarations placées avant la directive **public** : c'est là qu'il place tous les champs qui représentent les contrôles et les méthodes répondant aux événements.

Le code suivant ajoute dans la déclaration de l'objet fiche un champ de type Boolean, appelé *Drawing*. Il ajoute également deux champs afin de stocker les points de type TPoint : *Origin* et *MovePt*.

```
type
   TForm1 = class(TForm)
      procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
         Shift: TShiftState; X, Y: Integer);
      procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
         Shift: TShiftState; X, Y: Integer);
      procedure FormMouseMove(Sender: TObject; Button: TMouseButton;
         Shift: TShiftState; X, Y: Integer);
   public
      Drawing: Boolean;
      Origin, MovePt: TPoint;{ champs pour stocker les points }
   end;
```

Nous disposons maintenant d'un champ *Drawing* permettant de déterminer s'il faut dessiner. Il faut donc l'initialiser à *True* lorsque l'utilisateur enfonce le bouton de la souris et à *False* lorsqu'il le relâche :

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
   Shift: TShiftState; X, Y: Integer);
begin
   Drawing := True;{ définir l'indicateur de dessin }
   Canvas.MoveTo(X, Y);
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
   Shift: TShiftState; X, Y: Integer);
begin
   Canvas.LineTo(X, Y);
   Drawing := False;{ effacer l'indicateur de dessin }
end;
```

Ensuite, vous pouvez modifier le gestionnaire de l'événement *OnMouseMove* de façon à ne dessiner que si *Drawing* est à *True* :

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
   Shift: TShiftState; X, Y: Integer);
begin
   if Drawing then{ dessiner seulement si l'indicateur de dessin est défini }
      Canvas.LineTo(X, Y);
end;
```

Désormais, le dessin n'est effectué qu'entre les événements bouton de souris enfoncé et bouton de souris relâché, mais il y a toujours cette ligne brisée qui suit le déplacement de la souris au lieu d'une belle ligne droite.

Le problème tient à ce qu'à chaque déplacement de la souris, le gestionnaire de l'événement déplacement de souris appelle *LineTo* qui déplace la position du crayon. Par conséquent, le point d'origine de la ligne droite est perdu lorsque le bouton de la souris est relâché.

Amélioration du dessin des lignes

Maintenant que nous disposons de champs pour garder la trace des divers points, il est possible d'améliorer le dessin des lignes dans l'application.

Suivi du point d'origine

Lors du dessin d'une ligne, le point de départ de la ligne doit être suivi avec le champ *Origin*.

Il faut initialiser *Origin* avec le point où l'événement bouton de souris enfoncé se produit. De cette manière, le gestionnaire de l'événement bouton de souris relâché peut utiliser *Origin* pour tracer le début de la ligne, comme dans le code :

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
 Canvas.MoveTo(X, Y);
 Origin := Point(X, Y);{ enregistrer le début de ligne }
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.MoveTo(Origin.X, Origin.Y);{ déplacer le crayon au point de départ }
 Canvas.LineTo(X, Y);
 Drawing := False;
end;
```

Cette modification permet de redessiner la ligne finale. Mais qu'en est-il des dessins intermédiaires ?

Suivi des déplacements

Tel qu'est écrit le gestionnaire de l'événement *OnMouseMove*, il présente l'inconvénient de dessiner une ligne, non pas depuis sa position d'origine, mais depuis la position actuelle de la souris. Pour y remédier, il suffit de placer la position de dessin au point d'origine et de tirer la ligne jusqu'au point en cours :

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.MoveTo(Origin.X, Origin.Y);{ déplacer le crayon au point de départ }
   Canvas.LineTo(X, Y);
  end;
end;
```

Le code précédent fait le suivi de la position en cours de la souris, mais les lignes intermédiaires restent affichées et la ligne finale se voit à peine. L'astuce consiste à effacer chaque ligne avant de tracer la ligne suivante. Cela implique de garder la trace de son emplacement. C'est la fonction du champ *MovePt* ajouté préalablement.

Vous devez définir *MovePt* par le point d'arrivée de chaque ligne intermédiaire, et utiliser *MovePt* et *Origin* pour effacer cette ligne avant de dessiner la suivante :

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
 Canvas.MoveTo(X, Y);
 Origin := Point(X, Y);
 MovePt := Point(X, Y);
end;
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.Pen.Mode := pmNotXor;{ utiliser le mode XOR pour dessiner/effacer }
   Canvas.MoveTo(Origin.X, Origin.Y);{ déplacer le crayon à l'origine }
   Canvas.LineTo(MovePt.X, MovePt.Y);{ effacer l'ancienne ligne }
   Canvas.MoveTo(Origin.X, Origin.Y);{ commencer de nouveau à l'origine }
   Canvas.LineTo(X, Y);{ dessiner la nouvelle ligne }
  end;
 MovePt := Point(X, Y);{ enregistrer le point pour le prochain déplacement }
 Canvas.Pen.Mode := pmCopy;
end;
```

Maintenant, un effet satisfaisant est obtenu lorsque la ligne est dessinée. En modifiant le mode du crayon en *pmNotXor*, il combine la ligne avec les pixels de l'arrière-plan. Lorsque la ligne est effacée, les pixels sont en fait ramenés à leur état antérieur. En remettant le mode du crayon à *pmCopy* (sa valeur par défaut) après avoir dessiné les lignes, le crayon est prêt pour le dessin final lorsque le bouton de la souris est relâché.

Utilisation du multimédia

Delphi vous permet d'ajouter des composants multimédia à vos applications. Vous pouvez le faire en ajoutant le composant *TAnimate* de la page Win32 ou le composant *TMediaPlayer* de la page Système de la palette des composants. Utilisez le composant animation pour jouer des séquences vidéo silencieuses dans votre application. Utilisez le composant lecteur multimédia pour jouer des séquences audio ou vidéo dans une application.

Pour davantage d'informations sur les composants *TAnimate* et *TMediaPlayer*, voir l'aide en ligne de la VCL.

Cette section aborde les sujets suivants :

- Ajout de séquences vidéo silencieuses à une application
- Ajout de séquences audio et/ou vidéo à une application

Ajout de séquences vidéo silencieuses à une application

Le contrôle animation de Delphi vous permet d'ajouter des séquences vidéo silencieuses à votre application.

Pour ajouter une séquence vidéo silencieuse à une application :

1. Double-cliquez sur l'icône animation dans la page Win32 de la palette des composants. Cela place automatiquement un contrôle animation dans la fiche dans laquelle vous voulez afficher la séquence vidéo.
2. En utilisant l'inspecteur d'objets, sélectionnez la propriété *Name* et entrez un nouveau nom pour votre contrôle animation. Vous utiliserez ce nom pour appeler le contrôle animation (respectez les conventions standard de nomenclature des identificateurs Delphi).

   Travaillez toujours directement dans l'inspecteur d'objets pour initialiser des propriétés de conception ou pour créer des gestionnaires d'événements
3. Effectuez l'une des opérations suivantes :
   - Sélectionnez la propriété *Common AVI* et choisissez l'un des AVI proposés dans la liste déroulante.
   - Ou sélectionnez la propriété *FileName*, cliquez sur le bouton points de suspension et choisissez un fichier AVI dans l'un des répertoires disponibles localement ou sur le réseau, puis choisissez Ouvrir dans la boîte de dialogue Ouvrir AVI.
   - Ou sélectionnez une ressource AVI en utilisant les propriétés *ResName* ou *ResID*. Utilisez la propriété *ResHandle* pour indiquer le module contenant la ressource identifiée par *ResName* ou *ResID*.

   Cela charge le fichier AVI en mémoire. Pour afficher à l'écran le premier plan de la séquence AVI, utilisez la propriété *Active* ou la méthode *Play*, puis affectez la valeur *True* à la propriété *Open*.

4. Affectez à la propriété *Repetitions* le nombre spécifiant combien de fois la séquence AVI doit être jouée. Si cette valeur est nulle, la séquence est répétée jusqu'à l'appel de la méthode *Stop*.
5. Faites les autres modifications des propriétés du contrôle animation. Si, par exemple, vous voulez modifier le premier plan affiché à l'ouverture du contrôle, affectez le numéro de plan voulu à la propriété *StartFrame*.
6. Affectez la valeur *True* à la propriété *Active* en utilisant la liste déroulante ou écrivez un gestionnaire d'événement pour exécuter la séquence AVI quand un événement spécifique a lieu à l'exécution. Par exemple, pour activer la séquence AVI quand un objet bouton est choisi, écrivez en conséquence le gestionnaire d'événement *OnClick*. Vous pouvez également appeler la méthode *Play* pour faire jouer la séquence AVI.

Remarque Si vous faites des modifications à la fiche ou à l'un des composants de la fiche après avoir affecter la valeur *True* à *Active*, la propriété *Active* revient à *False* et vous devez la remettre à *True*. Vous devez donc faire ceci juste avant la compilation ou à l'exécution.

Exemple d'ajout de séquences vidéo silencieuses

Vous pouvez, par exemple, afficher un logo animé dans le premier écran apparaissant au démarrage de votre application. Une fois l'affichage du logo terminé, l'écran disparaît.

Pour exécuter cet exemple, créez un nouveau projet et enregistrez le fichier Unit1.pas sous le nom Frmlogo.pas et le fichier Project1.dpr sous le nom Logo.dpr. Ensuite :

1. Double-cliquez sur l'icône animation dans la page Win32 de la palette des composants.
2. En utilisant l'inspecteur d'objets, affectez à sa propriété Name la valeur *Logo1*.
3. Sélectionnez sa propriété FileName, cliquez sur le bouton points de suspension (...), choisissez le fichier cool.avi dans le répertoire ..\Demos\Coolstuf. Cliquez ensuite sur le bouton Ouvrir dans la boîte de dialogue Ouvrir AVI.

   Cela charge le fichier cool.avi en mémoire.
4. Positionnez le contrôle animation dans la fiche en cliquant dessus et en le faisant glisser sur le coin supérieur droit de la fiche.
5. Affectez la valeur 5 à sa propriété Repetitions.
6. Cliquez sur la fiche pour lui attribuer la focalisation et affectez à sa propriété Name la valeur LogoForm1 et à sa propriété Caption la valeur *Fenêtre Logo*. Diminuez la hauteur de la fiche pour y centrer à droite le contrôle animation.
7. Double-cliquez sur l'événement OnActivate et entrez le code suivant qui exécute la séquence AVI quand la fiche obtient la focalisation à l'exécution :

```
Logo1.Active := True;
```

8 Double-cliquez sur l'icône de libellé dans la page Standard de la palette des composants. Sélectionnez la propriété Caption du composant et entrez *Bienvenue à Cool Images 4.0*. Sélectionnez ensuite sa propriété Font, cliquez sur le bouton points de suspension (...) et choisissez dans la boîte de dialogue Fonte, Style : gras, Taille : 18, Couleur : Navy, puis choisissez OK. Cliquez sur le contrôle libellé et faites-le glisser pour le centrer dans la fiche.

9 Cliquez sur le contrôle animation pour lui donner la focalisation. Double-cliquez sur son événement OnStop et écrivez le code suivant pour fermer la fiche à l'arrêt du fichier AVI :

```
LogoForm1.Close;
```

10 Sélectionnez Exécuter | Exécuter pour exécuter la fenêtre au logo animé.

Ajout de séquences audio et/ou vidéo à une application

Le composant lecteur multimédia de Delphi vous permet d'ajouter des séquences audio et/ou vidéo à votre application. Il ouvre un périphérique de média et peut jouer, arrêter, faire une pause, enregistrer, etc., les séquences audio et/ou vidéo utilisées par le périphérique de média. Le périphérique de média peut être matériel ou logiciel.

Remarque Les séquences audio et/ou vidéo ne sont pas prises en charge en programmation multiplate-forme.

Pour ajouter une séquence audio et/ou vidéo à une application :

1 Double-cliquez sur l'icône du lecteur multimédia dans la page Système de la palette des composants. Cela place automatiquement un contrôle lecteur multimédia dans la fiche à laquelle vous voulez jouer les caractéristiques multimédia.

2 En utilisant l'inspecteur d'objets, sélectionnez la propriété *Name* et entrez le nouveau nom du contrôle lecteur multimédia. Vous utiliserez ce nom pour désigner le contrôle lecteur multimédia. Respectez les conventions standard des identificateurs de nom Delphi).

   Travaillez toujours directement dans l'inspecteur d'objets pour initialiser des propriétés de conception ou pour créer des gestionnaires d'événements.

3 Sélectionnez la propriété *DeviceType* et choisissez le type de périphérique approprié ouvert par la propriété *AutoOpen* ou la méthode *Open*. Si *DeviceType* a la valeur dtAutoSelect, le type de périphérique est sélectionné en fonction de l'extension du fichier média spécifié par la propriété *FileName*. Pour davantage d'informations sur les types de périphériques et leurs fonctions, voir le tableau suivant.

4 Si le périphérique stocke son média dans un fichier, spécifiez le nom du fichier média en utilisant la propriété *FileName*. Sélectionnez la propriété *FileName*. Cliquez sur le bouton points de suspension et choisissez un fichier média dans un répertoire disponible localement ou sur le réseau, puis choisissez Ouvrir dans la boîte de dialogue Ouvrir. Sinon à l'exécution, insérez

dans le lecteur matériel le support contenant le média (disque, cassette, etc) pour le périphérique de média sélectionné.

5 Affectez la valeur *True* à la propriété *AutoOpen*. Ainsi, le lecteur multimédia ouvre automatiquement le périphérique spécifié quand la fiche contenant le lecteur est créée à l'exécution. Si *AutoOpen* a la valeur *False*, le périphérique doit être ouvert par un appel de la méthode *Open*.

6 Affectez la valeur *True* à la propriété *AutoEnable* pour activer ou désactiver automatiquement à l'exécution les boutons nécessaires du lecteur multimédia. Sinon, double-cliquez sur la propriété *EnabledButtons* pour affecter la valeur *True* ou *False* à chaque bouton selon que vous souhaitez l'activer ou pas.

Le périphérique multimédia est exécuté, mis en pause ou arrêté quand l'utilisateur clique sur les boutons correspondants du composant lecteur multimédia. Il est également possible de contrôler le périphérique via les méthodes correspondant aux boutons (Play, Pause, Stop, Next, Previous, etc).

7 Positionnez la barre de contrôle du lecteur multimédia dans la fiche. Vous pouvez le faire en cliquant dessus et en la faisant glisser à la position de votre choix ou en sélectionnant la propriété *Align* et en choisissant l'alignement souhaité dans sa liste déroulante.

Si vous voulez que le lecteur multimédia soit invisible à l'exécution, affectez la valeur *False* à sa propriété *Visible* et contrôlez le périphérique en appelant les méthodes appropriées (*Play*, *Pause*, *Stop*, *Next*, *Previous*, *Step*, *Back*, *Start Recording*, *Eject*).

8 Effectuez les autres paramétrages du contrôle lecteur multimédia. Si, par exemple, le média nécessite une fenêtre d'affichage, affectez à la propriété *Display* le contrôle affichant le média. Si le périphérique utilise plusieurs pistes, affectez à la propriété *Tracks* la piste souhaitée.

Tableau 8.5 Types de périphériques multimédia et leurs fonctions

Type de périphérique	Logiciel/matériel utilisé	Joue	Utilise des pistes	Utilise une fenêtre d'affichage
dtAVIVideo	Lecteur AVI Vidéo pour Window	fichiers AVI Video	Non	Oui
dtCDAudio	Lecteur CD Audio pour Windows ou un lecteur CD Audio	Disques CD Audio	Oui	Non
dtDAT	Lecteur de cassettes audio-numériques	Cassettes audio-numériques	Oui	Non
dtDigitalVideo	Lecteur vidéo-numérique pour Windows	fichiers AVI, MPG, MOV	Non	Oui
dtMMMovie	Lecteur de films MM	films MM	Non	Oui
dtOverlay	Périphérique overlay	Vidéo analogique	Non	Oui
dtScanner	Scanner d'image	N/d pour Play (scanne des images avec Record)	Non	Non

Tableau 8.5 Types de périphériques multimédia et leurs fonctions (suite)

Type de périphérique	Logiciel/matériel utilisé	Joue	Utilise des pistes	Utilise une fenêtre d'affichage
dtSequencer	Séquenceur MIDI pour Windows	fichiers MIDI	Oui	Non
dtVCR	Enregistreur de cassettes vidéo	Cassettes vidéo	Non	Oui
dtWaveAudio	Lecteur audio Wav pour Windows	fichiers WAV	Non	Non

Exemple d'ajout de séquences audio et/ou vidéo (VCL seulement)

Cet exemple exécute une séquence vidéo AVI pour une publicité multimédia de Delphi. Pour exécuter cet exemple, créez un nouveau projet et enregistrez le fichier Unit1.pas sous le nom FrmAd.pas et le fichier Project1.dpr sous le nom DelphiAd.dpr. Puis :

1. Double-cliquez sur l'icône lecteur multimédia dans la page Système de la palette des composants.
2. En utilisant l'inspecteur d'objets, affectez à la propriété Name du lecteur multimédia la valeur *VideoPlayer1*.
3. Sélectionnez sa propriété DeviceType et choisissez dtAVIVideo dans la liste déroulante.
4. Sélectionnez sa propriété FileName, cliquez sur le bouton points de suspension (...) et choisissez le fichier speedis.avi dans le répertoire ..\Demos\Coolstuf. Choisissez le bouton Ouvrir dans la boîte de dialogue Ouvrir.
5. Affectez la valeur *True* à sa propriété AutoOpen et la valeur *False* à sa propriété Visible.
6. Double-cliquez sur l'icône animation dans la page Win32 de la palette des composants. Affectez la valeur *False* à sa propriété AutoSize, *175* à sa propriété Height et *200* à sa propriété Width. Cliquez sur le contrôle animation et faites-le glisser dans le coin supérieur gauche de la fiche.
7. Cliquez sur le contrôle lecteur multimédia pour lui donner la focalisation. Sélectionnez sa propriété Display et choisissez Animate1 dans la liste déroulante.
8. Cliquez sur la fiche pour lui attribuer la focalisation et sélectionnez sa propriété Name et affectez-lui la valeur Delphi_Ad. Redimensionnez la fiche pour lui donner la taille du contrôle animation.
9. Double-cliquez sur l'événement OnActivate et écrivez le code suivant pour exécuter la séquence vidéo AVI quand la fiche a la focalisation :

```
VideoPlayer1.Play;
```

10. Choisissez Exécuter | Exécuter pour exécuter la vidéo AVI.

Chapitre

9

Ecriture d'applications multithreads

Delphi dispose de plusieurs objets facilitant la conception d'applications multithreads. Les applications multithreads sont des applications qui contiennent plusieurs chemins d'exécution simultanés. Même si l'utilisation de plusieurs threads doit être mûrement réfléchie, elle peut améliorer un programme :

- **En évitant les engorgements.** Avec un seul thread, un programme doit arrêter complètement l'exécution pour attendre les processus lents comme les accès disque, la communication avec d'autres machines ou l'affichage de données multimédia. La CPU est inactive jusqu'à l'achèvement du processus. Avec plusieurs threads, l'application peut poursuivre l'exécution dans des threads séparés pendant qu'un thread attend le résultat du processus lent.
- **En organisant le comportement d'un programme.** Le comportement d'un programme peut souvent être décomposé en plusieurs processus fonctionnant de manière indépendante. L'utilisation de threads permet d'exécuter simultanément une section de code pour chacun de ces processus. Utilisez les threads pour assigner des priorités aux diverses tâches du programme afin d'attribuer davantage de temps machine aux tâches critiques.
- **En gérant plusieurs processeurs.** Si le système exécutant le programme dispose de plusieurs processeurs, il est possible d'améliorer les performances en décomposant le travail en plusieurs threads s'exécutant simultanément sur des processeurs distincts.

Remarque Tous les systèmes d'exploitation ne gèrent pas réellement l'utilisation de plusieurs processeurs, même si cela est proposé par le matériel sous-jacent. Par exemple, Windows 9x ne fait que simuler l'utilisation de processeurs multiples, même si le matériel sous-jacent le gère.

Définition d'objets thread

Dans la plupart des applications, vous pouvez utiliser un objet thread pour représenter un thread d'exécution de l'application. Les objets thread simplifient l'écriture d'applications multithreads en encapsulant les utilisations les plus fréquentes des threads.

Remarque Les objets thread ne permettent pas de contrôler les attributs de sécurité ou la taille de la pile des threads. Si vous souhaitez contrôler ces paramètres, vous devez utiliser la fonction *BeginThread*. Même si vous utilisez *BeginThread*, vous pouvez néanmoins utiliser les objets de synchronisation de threads et les méthodes décrites dans la section "Coordination de threads" à la page 9-7. Pour davantage d'informations sur l'utilisation de *BeginThread*, voir l'aide en ligne.

Pour utiliser un objet thread dans une application, créez un nouveau descendant de *TThread*. Pour créer un descendant de TThread, choisissez Fichier | Nouveau dans le menu principal. Dans la boîte de dialogue des nouveaux objets, sélectionnez Objet thread. Vous devez spécifier le nom de classe du nouvel objet thread. Une fois le nom spécifié, Delphi crée un nouveau fichier unité pour implémenter le thread.

Remarque A la différence de la plupart des boîtes de dialogue de l'EDI demandant un nom de classe, la boîte de dialogue Nouvel objet thread ne préfixe pas automatiquement le nom de classe avec un 'T'.

Le fichier unité automatiquement généré contient le squelette du code du nouvel objet thread. Si vous avez nommé ce thread *TMyThread*, le code doit avoir l'aspect suivant :

```
unit Unit2;
interface
uses
  Classes;
type
  TMyThread = class(TThread)
  private
    { Déclarations privées }
  protected
    procedure Execute; override;
  end;
implementation
{ TMyThread }
procedure TMyThread.Execute;
begin
  { Code du thread ici }
end;
end.
```

Vous devez spécifier le code de la méthode *Execute*. Ces étapes sont décrites dans les sections suivantes.

Initialisation du thread

Si vous souhaitez écrire du code d'initialisation pour la nouvelle classe de thread, vous devez écraser la méthode Create. Ajoutez un nouveau constructeur à la déclaration de la classe de thread et écrivez le code d'initialisation pour l'implémenter. C'est là que vous pouvez affecter une priorité par défaut au thread et indiquer s'il doit être libéré automatiquement à la fin de son exécution.

Affectation d'une priorité par défaut

La priorité indique la préférence accordée au thread quand le système d'exploitation répartit le temps machine entre les différents threads de l'application. Utilisez un thread de priorité élevée pour gérer les tâches critiques et un thread de priorité basse pour les autres tâches. Pour indiquer la priorité de l'objet thread, affectez la propriété *Priority*.

Si vous écrivez une application Windows, les valeurs de *Priority* se répartissent sur une échelle comportant sept niveaux, comme décrit dans le tableau suivant :

Tableau 9.1 Priorités des threads

Valeur	Priorité
tpIdle	Le thread s'exécute uniquement quand le système est inoccupé. Windows n'interrompt pas d'autres threads pour exécuter un thread de priorité *tpIdle.*
tpLowest	La priorité du thread est deux points en dessous de la normale.
tpLower	La priorité du thread est un point en dessous de la normale.
tpNormal	Le thread a une priorité normale.
tpHigher	La priorité du thread est un point au-dessus de la normale.
tpHighest	La priorité du thread est deux points au-dessus de la normale.
tpTimeCritical	Le thread a la priorité la plus élevée.

Remarque Si vous écrivez une application multiplate-forme, vous devez séparer le code destiné à attribuer les priorités pour Windows du code pour Linux. Sous Linux, *Priority* est une valeur numérique qui dépend de la politique choisie pour les threads, politique qui ne peut être modifiée que par l'utilisateur root. Voir la version CLX de *TThread* et *Priority* dans l'aide en ligne pour plus de détails.

Attention "Gonfler" la priorité du thread pour une opération utilisant intensivement la CPU peut "sous-alimenter" les autres threads de l'application. Il ne faut accorder une priorité élevée qu'à des threads qui passent l'essentiel du temps à attendre des événements extérieurs.

Le code suivant illustre le constructeur d'un thread de priorité basse qui effectue des tâches d'arrière-plan ne devant pas interférer avec les performances du reste de l'application :

```
constructor TMyThread.Create(CreateSuspended: Boolean);
begin
  inherited Create(CreateSuspended);
  Priority := tpIdle;
end;
```

Libération des threads

Généralement, lorsque les threads ont fini d'être exécutés, ils peuvent être simplement libérés. Dans ce cas, le plus simple consiste à laisser l'objet thread se libérer lui-même. Pour ce faire, affectez la valeur *True* à la propriété *FreeOnTerminate*.

Il y a cependant des cas où la fin d'un thread doit être coordonnée avec les autres threads. Par exemple, il se peut que vous deviez attendre qu'un thread renvoie une valeur avant d'effectuer une action dans un autre thread. Pour ce faire, vous ne souhaitez pas libérer le premier thread avant que le second n'ait reçu la valeur renvoyée. Vous pouvez traiter ce type de situation en affectant la valeur *False* à *FreeOnTerminate* et en libérant explicitement le premier thread à partir du second.

Ecriture de la fonction thread

La méthode *Execute* constitue la fonction thread. Vous pouvez la concevoir comme un programme qui est exécuté par l'application, à cette différence près qu'il partage le même espace de processus. L'écriture d'une fonction thread est plus délicate que celle d'un programme distinct car il faut prendre garde à ne pas écraser la mémoire utilisée par d'autres threads de l'application. D'un autre côté, comme le thread partage le même espace de processus que les autres threads, il est possible d'utiliser la mémoire partagée pour faire communiquer les threads.

Utilisation du thread principal VCL/CLX

Quand vous utilisez des objets appartenant aux hiérarchies d'objets VCL et CLX, leurs propriétés et méthodes ne sont pas nécessairement adaptées à l'utilisation de threads. C'est-à-dire que l'accès aux propriétés et méthodes peut effectuer des actions utilisant de la mémoire qui n'est pas protégée de l'action d'autres threads. De ce fait, un thread principal est placé à part pour l'accès aux objets VCL et CLX. C'est ce thread qui gère tous les messages Windows reçus par les composants d'une application.

Si tous les objets accèdent à leurs propriétés et exécutent leurs méthodes dans ce seul thread, il n'est pas nécessaire de se préoccuper d'éventuelles interférences entre les objets. Pour utiliser le thread principal, créez une routine séparée effectuant les actions nécessaires. Appelez cette routine séparée depuis la méthode *Synchronize* de votre thread. Par exemple :

```
procedure TMyThread.PushTheButton;
begin
  Button1.Click;
end;
⋮
procedure TMyThread.Execute;
begin
⋮
  Synchronize(PushTheButton);
⋮
end;
```

Synchronize attend le thread principal pour entrer dans la boucle des messages puis exécute la méthode qui lui est transmise.

Remarque Comme Synchronize utilise la boucle des messages, elle ne fonctionne pas dans les applications console. Vous devez utiliser d'autres mécanismes, comme les sections critiques, pour protéger l'accès aux objets VCL ou CLX dans les applications console.

Il n'est pas toujours nécessaire d'utiliser le thread principal. Certains objets sont adaptés aux threads. Il est préférable de ne pas utiliser la méthode *Synchronize* quand vous savez que les méthodes d'un objet sont adaptées à l'utilisation des threads, car cela améliore les performances en évitant d'avoir à attendre le thread VCL ou CLX pour entrer dans la boucle de messages. Il n'est pas nécessaire d'utiliser la méthode *Synchronize* dans les situations suivantes :

- Les composants d'accès aux données sont adaptés aux threads comme suit : pour les ensembles de données BDE chaque thread doit disposer de son propre composant session de base de données. Il n'y a qu'une seule exception : les pilotes Access. Ces pilotes sont conçus en utilisant la bibliothèque ADO Microsoft qui n'est pas adaptée aux threads. Pour dbDirect, il suffit que la bibliothèque client du fournisseur soit adaptée aux threads pour que les composants dbDirect le soient également. Les composants ADO et InterbaseExpress sont adaptés aux threads.

  Lorsque vous utilisez des composants d'accès aux données, vous devez néanmoins encadrer tous les appels aux contrôles orientés données dans la méthode *Synchronize*. Ainsi, il est nécessaire de synchroniser les appels qui relient un contrôle orienté données à un ensemble de données, mais il n'est pas nécessaire de le faire pour accéder aux données d'un champ de l'ensemble de données.

  Pour davantage d'informations sur l'utilisation de sessions de base de données avec les threads dans des applications BDE, voir "Gestion de sessions multiples" à la page 20-32.

- Les objets VisualCLX ne sont pas adaptés aux threads.
- Les objets DataCLX sont adaptés aux threads.
- Les objets graphiques sont adaptés aux threads. Il n'est pas nécessaire d'utiliser le thread principal VCL ou CLX pour accéder aux objets *TFont*, *TPen*, *TBrush*, *TBitmap*, *TMetafile* (VCL seulement), *TDrawing* (CLX seulement) et *TIcon*. Il est possible d'accéder aux objets canevas en dehors de la méthode *Synchronize* en les verrouillant (voir "Verrouillage d'objets" à la page 9-8).
- Les listes d'objets ne sont pas adaptées aux threads, mais vous pouvez utiliser une version adaptée aux threads, *TThreadList*, à la place de *TList*.

Appelez régulièrement la routine *CheckSynchronize* depuis le thread principal de votre application pour que les threads d'arrière-plan synchronisent leur exécution sur le thread principal. Le meilleur emplacement pour appeler *CheckSynchronize* est lorsque l'application est inactive (par exemple, dans le gestionnaire de l'événement *OnIdle*). Cela vous garantit qu'il est sans danger de faire appels aux méthodes du thread d'arrière-plan.

Utilisation de variables locales aux threads

La méthode *Execute* et toutes les routines qu'elle appelle ont leurs propres variables locales comme toute routine Pascal Objet. Ces routines peuvent également accéder à toutes les variables globales. En fait, les variables globales constituent un mécanisme puissant de communication entre les threads.

Mais dans certains cas, vous souhaitez utiliser des variables globales pour les routines du thread sans qu'elles ne soient partagées par les autres instances de la même classe de thread. Il est possible pour ce faire de déclarer des variables locales au thread. Déclarez une variable locale au thread en la déclarant dans une section **threadvar**. Par exemple :

```
threadvar
  x : integer;
```

déclare une variable de type entier privée pour chaque thread de l'application, mais globale à l'intérieur de chaque thread.

La section threadvar ne peut être utilisée que pour des variables globales. Les variables pointeur et fonction ne peuvent pas être des variables de thread. Les types utilisant une sémantique de copie lors de l'écriture, comme les chaînes longues ne peuvent pas non plus faire office de variables de thread.

Vérification de l'arrêt par d'autres threads

Un thread commence son exécution quand la méthode *Execute* est appelée (voir "Exécution d'objets thread" à la page 9-12) et se poursuit jusqu'à l'arrêt de *Execute*. Cela correspond à une situation dans laquelle le thread effectue une tâche spécifique puis s'arrête une fois celle-ci terminée. Dans certains cas, une application a besoin qu'un thread poursuive son exécution jusqu'à ce qu'un critère externe soit respecté.

Il est possible de permettre à d'autres threads de signaler qu'il est temps que votre thread arrête de s'exécuter en testant la propriété *Terminated*. Quand un autre thread tente de terminer votre thread, il appelle la méthode *Terminate*. *Terminate* affecte la valeur *True* à la propriété *Terminated* de votre thread. C'est à la méthode *Execute* de votre thread d'implémenter la méthode *Terminate* en testant la valeur de la propriété *Terminated*. L'exemple suivant illustre une manière de procéder :

```
procedure TMyThread.Execute;
begin
  while not Terminated do
    PerformSomeTask;
end;
```

Gestion des exceptions dans la fonction thread

La méthode *Execute* doit capturer toutes les exceptions qui se produisent dans le thread. Si vous échouez à capturer une exception dans votre fonction thread, votre application risque de provoquer des violations d'accès. Cela ne se voit pas lorsque vous développez car l'EDI capture l'exception, mais lorsque vous

exécuterez votre application hors du débogueur, l'exception provoquera une erreur d'exécution et l'application cessera de s'exécuter.

Pour capturer les exceptions se produisant à l'intérieur de votre fonction thread, ajoutez un bloc **try**...**except** à l'implémentation de la méthode *Execute* :

```
procedure TMyThread.Execute;
begin
  try
    while not Terminated do
      PerformSomeTask;
  except
    { faire quelque chose avec les exceptions }
  end;
end;
```

Ecriture du code de nettoyage

Vous pouvez centraliser le code de nettoyage lors de la fin de l'exécution du thread. Juste avant la fin du thread, un événement *OnTerminate* a lieu. Placez l'éventuel code de nettoyage dans le gestionnaire d'événement *OnTerminate* afin de garantir son exécution quel que soit le chemin d'exécution suivi par la méthode *Execute*.

Le gestionnaire d'événement *OnTerminate* n'est pas exécuté comme partie de votre thread. Il est en fait exécuté dans le contexte du thread principal VCL ou CLX de votre application. Cela a deux implications :

- Il n'est pas possible d'utiliser de variables locales au thread dans un gestionnaire d'événement *OnTerminate* (sauf à vouloir utiliser les valeurs du thread principal VCL ou CLX).
- Il est possible d'accéder en toute sécurité à tous les composants et objets VCL ou CLX dans le gestionnaire d'événement *OnTerminate* sans se préoccuper des conflits avec les autres threads.

Pour davantage d'informations sur le thread principal VCL ou CLX, voir "Utilisation du thread principal VCL/CLX" à la page 9-4.

Coordination de threads

Quand vous écrivez le code exécuté lorsque le thread s'exécute, vous devez tenir compte du comportement des autres threads qui peuvent s'exécuter simultanément. En particulier, il faut éviter que deux threads tentent d'utiliser simultanément le même objet ou la même variable globale. De plus, le code d'un thread peut dépendre de tâches effectuées par d'autres threads.

Eviter les accès simultanés

Pour éviter les conflits avec d'autres threads lors de l'accès à des objets ou des variables, il peut être nécessaire de bloquer l'exécution des autres threads jusqu'à ce que le code d'un thread ait terminé une opération. Mais il ne faut pas bloquer inutilement l'exécution des threads. Cela peut provoquer une dégradation importante des performances et réduire à néant les avantages liés à l'utilisation de threads multiples.

Verrouillage d'objets

Certains objets disposent d'un verrouillage intégré qui empêche les autres threads d'utiliser cette instance d'objet.

Ainsi, les objets canevas (*TCanvas* et ses descendants) ont une méthode *Lock* qui empêche les autres threads d'accéder au canevas jusqu'à l'appel de la méthode *Unlock*.

La VCL et CLX contiennent également tous les deux un objet liste adapté aux threads, *TThreadList*. L'appel de*TThreadList.LockList* renvoie l'objet liste tout en empêchant les autres threads d'exécution d'utiliser la liste jusqu'à l'appel de la méthode *UnlockList*. Les appels des méthodes *TCanvas.Lock* et *TThreadList.LockList* peuvent être imbriqués. Le verrou n'est pas libéré tant que le dernier verrouillage n'est pas associé au déverrouillage correspondant dans le même thread.

Utilisation de sections critiques

Pour les objets ne disposant pas de verrouillage intégré, vous pouvez utiliser une section critique. Les sections critiques fonctionnent comment une porte ne pouvant être franchie que par un seul thread à la fois. Pour utiliser une section critique, créez une instance globale de *TCriticalSection*. *TCriticalSection* dispose de deux méthodes, *Acquire* (qui empêche les autres threads d'exécuter la section) et *Release* (qui retire le blocage).

Chaque section critique est associée à la mémoire globale devant être protégée. Chaque thread accédant à cette mémoire globale doit commencer par utiliser la méthode *Acquire* pour vérifier qu'un autre thread n'est pas en train de l'utiliser. Une fois terminé, le thread appelle la méthode *Release* afin que les autres threads puissent accéder à la mémoire globale en appelant *Acquire*.

Attention Les sections critiques ne peuvent fonctionner que si tous les threads les utilisent pour accéder à la mémoire globale associée. Les threads qui ne tiennent pas compte des sections critiques et accèdent à la mémoire globale sans appeler *Acquire* peuvent provoquer des problèmes d'accès simultanés.

Par exemple, une application a une section critique des variables globales, *LockXY*, qui bloque l'accès aux variables globales X et Y. Tout thread utilisant X ou Y doit encadrer cette utilisation d'appels à la section critique comme ci-dessous :

```
LockXY.Acquire; { bloque les autres threads }
try
```

```
    Y := sin(X);
  finally
    LockXY.Release;
  end;
```

Utilisation du synchronisateur à écriture exclusive et lecture multiple

Lorsque vous utilisez des sections critiques pour protéger la mémoire globale, seul un thread peut utiliser la mémoire à un moment donné. Cette protection peut être exagérée, notamment si un objet ou une variable doit être souvent lu mais dans lequel vous écrivez très rarement. Il n'y a aucun danger à ce que plusieurs threads lisent la même mémoire simultanément, pourvu qu'aucun thread n'y écrit.

Lorsqu'une mémoire globale est souvent lue, mais dans laquelle les threads n'écrivent qu'occasionnellement, vous pouvez la protéger à l'aide de l'objet *TMultiReadExclusiveWriteSynchronizer*. Cet objet agit comme une section critique mais permet à plusieurs threads de lire la mémoire qu'il protège à condition qu'aucun thread n'y écrive. Les threads doivent disposer d'un accès exclusif en écriture à la mémoire protégée par *TMultiReadExclusiveWriteSynchronizer*.

Pour utiliser un synchronisateur à écriture exclusive et à lecture multiple, créez une instance globale de *TMultiReadExclusiveWriteSynchronizer* associée à la mémoire globale que vous souhaitez protéger. Chaque thread qui lit cette mémoire doit au préalable appeler la méthode *BeginRead*. *BeginRead* évite qu'un autre thread n'écrive simultanément dans la mémoire. Lorsqu'un thread a fini de lire la mémoire protégée, il appelle la méthode *EndRead*. Tout thread qui écrit dans la mémoire protégée doit au préalable appeler *BeginWrite*. *BeginWrite* évite qu'un autre thread ne lise ou n'écrive simultanément dans la mémoire. Lorsqu'un thread a fini d'écrire dans la mémoire protégée, il appelle la méthode *EndWrite*, de sorte que les threads en attente puissent commencer à lire la mémoire.

Attention Comme les sections critiques, le synchronisateur à écriture exclusive et à lecture multiple ne fonctionne que si chaque thread l'utilise pour accéder à la mémoire globale associée. Les threads qui ignorent le synchronisateur et accèdent à la mémoire globale sans appeler *BeginRead* ou *BeginWrite* génèrent des problèmes d'accès simultané.

Autres techniques de partage de la mémoire

Si vous utilisez des objets de la VCL ou de CLX, utilisez le thread principal pour exécuter votre code. L'utilisation du thread principal garantit que les objets n'accèdent pas indirectement à de la mémoire utilisée par d'autres objets VCL ou CLX dans d'autres threads. Voir "Utilisation du thread principal VCL/CLX" à la page 9-4, pour davantage d'informations sur le thread principal.

Si la mémoire globale n'a pas besoin d'être partagée par plusieurs threads, envisagez d'utiliser des variables locales aux threads au lieu de variables globales. En utilisant des variables locales aux threads, votre thread n'a pas besoin d'attendre ou de bloquer les autres threads. Voir "Utilisation de variables locales aux threads" à la page 9-6, pour davantage d'informations sur les variables locales aux threads.

Attente des autres threads

Si votre thread doit attendre la fin d'autres threads pour terminer une tâche, vous pouvez demander au thread de suspendre son exécution. Vous pouvez attendre la fin de l'exécution d'un autre thread ou attendre qu'un autre thread signale qu'il a achevé une tâche.

Attente de la fin d'exécution d'un thread

Pour attendre la fin de l'exécution d'un thread, utilisez la méthode *WaitFor* de l'autre thread. *WaitFor* ne revient que lorsque l'autre thread se termine, soit en finissant sa propre méthode *Execute,* soit à la suite d'une exception. Par exemple, le code suivant attend qu'un autre thread remplisse un objet liste de threads avant d'accéder aux objets de la liste :

```
if ListFillingThread.WaitFor then
begin
  with ThreadList1.LockList do
  begin
    for I := 0 to Count - 1 do
      ProcessItem(Items[I]);
  end;
  ThreadList1.UnlockList;
end;
```

Dans l'exemple précédent, l'accès aux éléments de la liste ne se fait que lorsque la méthode *WaitFor* indique que la liste a été remplie. La valeur renvoyée doit être affectée par la méthode *Execute* du thread en attente. Cependant, puisque les threads appelant *WaitFor* veulent connaître le résultat de l'exécution du thread, la méthode *Execute* ne renvoie pas de valeur. A la place, la méthode *Execute* initialise la propriété *ReturnValue*. *ReturnValue* est alors renvoyée par la méthode *WaitFor* quand elle est appelée par d'autres threads. Les valeurs renvoyées sont des entiers. Votre application en détermine la signification.

Attente de l'achèvement d'une tâche

Parfois, il est nécessaire d'attendre qu'un thread termine une opération au lieu d'attendre la fin de l'exécution d'un thread particulier. Pour ce faire, utilisez un objet événement. Les objets événements (*TEvent*) doivent être créés avec une portée globale afin qu'ils puissent agir comme des signaux visibles pour tous les threads.

Quand un thread termine une opération dont dépendent d'autres threads, il appelle *TEvent.SetEvent*. *SetEvent* active le signal afin que les autres threads le surveillant sachent que l'opération a été achevée. Pour désactiver le signal, utilisez la méthode *ResetEvent*.

Par exemple, imaginons une situation dans laquelle vous devez attendre la fin de l'exécution de plusieurs threads au lieu d'un seul. Comme vous ne savez pas quel sera le dernier thread, vous ne pouvez pas utiliser la méthode *WaitFor* de l'un d'eux. A la place, vous pouvez faire en sorte que chaque thread incrémente

un compteur à la fin de son exécution et que le dernier thread signale l'achèvement de l'exécution de tous les threads en générant un événement.

Le code suivant montre la fin du gestionnaire d'événement *OnTerminate* de tous les threads dont l'exécution doit être achevée. *CounterGuard* est un objet section critique global qui évite l'utilisation simultanée du compteur par plusieurs threads. *Counter* est une variable globale qui compte le nombre de threads dont l'exécution est achevée.

```
procedure TDataModule.TaskThreadTerminate(Sender: TObject);
begin
⋮
  CounterGuard.Acquire; { obtient un verrou sur le compteur }
  Dec(Counter);   { décrémente la variable globale du compteur }
  if Counter = 0 then
    Event1.SetEvent; { signale s'il s'agit du dernier thread }
  CounterGuard.Release; { déverrouille le compteur }
⋮
end;
```

Le thread principal initialise la variable Counter, lance les threads de tâche et attend le signal indiquant l'achèvement de l'exécution de tous les threads en appelant la méthode *WaitFor*. *WaitFor* attend l'activation du signal pendant une durée spécifiée et renvoie l'une des valeurs du tableau suivant :

Tableau 9.2 Valeurs renvoyées par WaitFor

Valeur	Signification
wrSignaled	Le signal de l'objet événement a été activé.
wrTimeout	La durée spécifiée s'est écoulée sans que le signal soit défini.
wrAbandoned	L'objet événement a été détruit avant l'écoulement de la durée spécifiée.
wrError	Une erreur a eu lieu pendant l'attente.

Le code suivant montre comment le thread principal lance les threads de tâche et reprend la main lorsque leur exécution est achevée :

```
Event1.ResetEvent; { efface l'événement avant de lancer les threads }
for i := 1 to Counter do
  TaskThread.Create(False); { crée et lance les threads de tâche }
if Event1.WaitFor(20000) <> wrSignaled then
  raise Exception;
{ poursuite avec le thread principal. L'exécution de tous les threads de tâche est achevée
}
```

Remarque Si vous ne voulez pas cesser d'attendre un événement après un délai spécifié, transmettez à la méthode *WaitFor* une valeur de paramètre INFINITE. Faites attention en utilisant INFINITE, car cela peut provoquer le blocage du thread si le signal attendu n'arrive pas.

Exécution d'objets thread

Une fois une classe thread implémentée en définissant sa méthode *Execute*, vous pouvez l'utiliser dans une application pour exécuter le code de sa méthode *Execute*. Pour utiliser un thread, créez une instance de la classe thread. L'instance de thread peut être créée pour un démarrage immédiat ou placée en état d'attente afin de n'être exécutée qu'avec l'appel de la méthode *Resume*. Pour créer un thread s'exécutant immédiatement, affectez la valeur *False* au paramètre *CreateSuspended* du constructeur. Par exemple, la ligne suivante crée un thread et commence son exécution :

```
SecondProcess := TMyThread.Create(false); {crée et exécute le thread }
```

Attention Ne créez pas trop de threads dans une application. Le surcoût lié à la gestion de plusieurs threads peut influer sur les performances. La limite recommandée est de 16 threads par processus sur une machine disposant d'un seul processeur. Cette limite suppose que la plupart de ces threads attendent des événements externes. Si tous les threads sont actifs, il convient de réduire encore ce nombre.

Vous pouvez créer plusieurs instances du même type de thread pour exécuter du code parallèle. Vous pouvez, par exemple, démarrer une nouvelle instance d'un thread en réponse à une action de l'utilisateur, ce qui permet à chaque thread de générer la réponse attendue.

Redéfinition de la priorité par défaut

Quand le temps machine accordé au thread est lié à la tâche accomplie par le thread, sa priorité est définie dans le constructeur, comme décrit dans "Initialisation du thread" à la page 9-3. Par contre, si la priorité du thread varie en fonction du moment de l'exécution du thread, créez le thread en état suspendu, affectez sa priorité puis démarrez l'exécution du thread :

```
SecondProcess :=  TMyThread.Create(True); { création mais pas exécution }
SecondProcess.Priority := tpLower; { réduire la priorité normale }
SecondProcess.Resume; { exécuter le thread }
```

Remarque Si vous écrivez une application multiplate-forme, vous devez séparer le code destiné à attribuer les priorités pour Windows du code pour Linux. Sous Linux, *Priority* est une valeur numérique qui dépend de la politique choisie pour les threads, politique qui ne peut être modifiée que par l'utilisateur root. Voir la version CLX de *TThread* et *Priority* dans l'aide en ligne pour plus de détails.

Démarrage et arrêt des threads

Un thread peut être démarré et arrêté plusieurs fois avant de terminer son exécution. Pour interrompre temporairement l'exécution d'un thread, appelez sa méthode *Suspend*. Quand il faut reprendre l'exécution du thread, appelez sa méthode *Resume*. *Suspend* augmente un compteur interne, il est donc possible d'imbriquer les appels aux méthodes *Suspend* et *Resume*. L'exécution du thread ne reprend que si à chaque appel de *Suspend* correspond un appel de *Resume*.

Vous pouvez mettre un terme à l'exécution d'un thread en appelant sa méthode *Terminate*. *Terminate* affecte la valeur *True* à la propriété *Terminated* du thread. Si vous avez correctement implémenté la méthode *Execute*, elle teste périodiquement la valeur de la propriété *Terminated* et s'arrête si *Terminated* a la valeur *True*.

Débogage d'applications multithreads

Lors du débogage d'applications multithreads, il est compliqué de surveiller l'état de tous les threads s'exécutant simultanément ou même de déterminer quel thread s'exécute quand vous êtes sur un point d'arrêt. Vous pouvez utiliser la boîte d'état des threads pour surveiller et manipuler tous les threads de l'application. Pour afficher la boîte de dialogue Etat des threads, choisissez Voir | Threads dans le menu principal.

Quand un événement de débogage a lieu, (point d'arrêt, exception, pause), la vue Etat des threads indique l'état de chaque thread. Cliquez avec le bouton droit de la souris dans la boîte de dialogue Etat des threads pour accéder aux commandes permettant d'accéder au code source correspondant ou changer le thread courant. Quand un thread est marqué comme en cours, l'étape suivante ou l'opération d'exécution suivante se fait relativement à ce thread.

La boîte de dialogue Etat des threads liste tous les threads d'exécution de l'application par leur identificateur de thread. Si vous utilisez des objets thread, l'identificateur de thread correspond à la valeur de la propriété *ThreadID*. Si vous n'utilisez pas d'objets thread, l'identificateur de chaque thread est renvoyé lors de l'appel de *BeginThread*.

Pour davantage d'informations sur la boîte d'état des threads, voir l'aide en ligne.

Chapitre

10

Utilisation de CLX pour le développement multiplate-forme

Vous pouvez utiliser Delphi pour développer des applications 32 bits multiplates-formes qui s'exécuteront sur les systèmes d'exploitation Windows et Linux. Pour ce faire, vous pouvez démarrer avec une application Windows existante et la modifier, ou bien créer une nouvelle application en suivant les méthodes recommandées pour l'écriture d'un code indépendant de la plate-forme. Kylix est le logiciel Delphi pour Linux de Borland ; il vous permet de compiler et de développer des applications sous Linux. Si vous souhaitez développer et déployer des applications sous Linux et Windows, vous aurez besoin d'utiliser Kylix ainsi que Delphi.

Ce chapitre décrit la manière de modifier les applications Delphi pour qu'elles se compilent sous Linux et il contient des informations sur les différences entre le développement d'applications sous Windows et sous Linux. Il fournit également des lignes directrices pour l'écriture d'un code portable entre les différents environnements.

Remarque La plupart des applications développées en utilisant CLX (sans appels à des API propres à un système d'exploitation) s'exécuteront à la fois sur les plates-formes Linux et Windows. Les application doivent être compilées sur la plate-forme sur laquelle vous voulez qu'elles s'exécutent.

Création d'applications multiplates-formes

Vous créerez des applications multiplates-formes presque comme vous créez n'importe quelle application Delphi. Vous devez utiliser les composants visuels CLX et éviter les appels aux API spécifiques du système d'exploitation pour que votre application soit totalement indépendante de la plate-forme. (Voir "Ecriture de code portable" à la page 10-19, pour des conseils sur l'écriture des applications multiplates-formes.)

Pour créer une application multiplate-forme :

1. Dans l'EDI, choisissez Fichier | Nouveau | Application CLX.
   La palette de composants montre les composants pouvant être utilisés dans les applications CLX.

Remarque Seuls certains composants non visuels de Windows peuvent être utilisés dans les applications CLX. Les pages ADO, BDE, Système, DataSnap, InterBase, Site Express, FastNet, QReport, COM+, WebSnap et Serveurs de la palette de composants incluent des fonctionnalités qui ne fonctionneront que dans les applications CLX sous Windows. Si vous prévoyez de compiler votre application également sous Linux, n'utilisez pas les composants proposés dans ces pages, ou utilisez des **$IFDEF** pour marquer les sections de code qui les utilisent comme étant uniquement destinées à Windows.

2. Développez votre application dans l'EDI. N'oubliez pas de n'utiliser que des composants CLX dans votre application.
3. Compilez et testez l'application sur chaque plate-forme sur laquelle vous envisagez de la faire tourner. Examinez tous les messages d'erreur pour repérer les endroits où des modifications supplémentaires seront nécessaires.

Lorsque vous placez une application sous Kylix, vous devez réinitialiser les options du projet. En effet, le fichier .dof stockant les options du projet est recréé par Kylix et appelé .kof (avec les options par défaut). Vous pouvez également stocker dans l'application un grand nombre des options de compilation en tapant Ctrl+O+O. Les options sont placées au début du fichier ouvert à ce moment-là.

Le fichier fiche d'une application multiplate-forme aura l'extension xfm et non pas dfm. Cela permet de distinguer les fiches multiplates-formes utilisant des composants CLX et les fiches utilisant des composants VCL. Un fichier fiche xfm fonctionnera à la fois sous Windows et sous Linux, mais une fiche dfm ne fonctionnera que sous Windows.

Vous pourriez aussi bien commencer le développement de votre application multiplate-forme en partant de Kylix au lieu de Delphi :

1. Développez, compilez et testez l'application sous Linux en utilisant Kylix.
2. Placez les fichiers source de l'application sous Windows.
3. Réinitialisez les options du projet.
4. Recompilez l'application sous Windows en utilisant Delphi.

Pour plus d'informations sur l'écriture des applications de bases de données ou des applications Internet indépendantes des plates-formes, voir "Applications de bases de données multiplates-formes" à la page 10-26 et "Applications Internet multiplates-formes" à la page 10-34.

Portage d'applications VCL vers CLX

Si vous disposez d'applications Delphi écrites pour l'environnement Windows, vous pouvez les rendre multiplates-formes. La facilité de cette évolution dépend de la nature et de la complexité de l'application, ainsi que du nombre de dépendances Windows.

Les sections suivantes décrivent quelques-unes des différences majeures entre les environnements Windows et Linux et fournissent des lignes directrices sur la manière de démarrer le portage d'une application.

Techniques de portage

Les approches que vous pouvez adopter pour porter une application d'une plate-forme vers une autre sont les suivantes :

Tableau 10.1 Techniques de portage

Technique	Description
Portage propre à une plate-forme	Cible un système d'exploitation et les API sous-jacentes
Portage multiplates-formes	Cible une API multiplates-formes
Emulation Windows	Ne modifie pas le code et porte les API utilisées

Portages propres à une plate-forme

Les portages propres à une plate-forme peuvent être longs et coûteux et ne produisent qu'un résultat ciblé. Ils créent plusieurs bases de code différentes, ce qui les rend particulièrement difficiles à maintenir. Toutefois, chaque portage est conçu pour un système d'exploitation particulier et peut tirer parti des fonctionnalités propres à la plate-forme. L'application s'exécute donc en règle générale plus rapidement.

Portages multiplates-formes

Les portages multiplates-formes offrent généralement la technique la plus rapide et les applications portées ciblent plusieurs plates-formes. En réalité, la quantité de travail nécessaire au développement d'applications multiplates-formes dépend beaucoup du code existant. Si le code a été développé sans souci d'indépendance par rapport à la plate-forme, vous pouvez rencontrer des scénarios où la "logique" indépendante de la plate-forme et la "mise en œuvre" dépendante de la plate-forme sont mélangées.

L'approche multiplates-formes est préférable, car la logique métier s'exprime en termes indépendants de la plate-forme. Certains services sont masqués par une interface interne qui est identique sur toutes les plates-formes mais possède une mise en œuvre particulière sur chacune. La bibliothèque d'exécution de Delphi en est un exemple : l'interface est très similaire sur les deux plates-formes, même si la mise en œuvre peut être très différente. Vous devez séparer les éléments multiplates-formes, puis mettre en œuvre des services particuliers au niveau

supérieur. Cette approche est en fin de compte la solution la moins coûteuse, grâce à la réduction des coûts de maintenance occasionnée par un large partage de la base de source et une amélioration de l'architecture d'application.

Portages d'émulation Windows

L'émulation Windows est la méthode la plus complexe et elle peut être très coûteuse, mais c'est celle qui offrira le plus de similarité entre l'application Linux résultante et une application Windows existante. Cette approche consiste à mettre en œuvre la fonctionnalité Windows sous Linux. D'un point de vue ingénierie, cette solution est très difficile à maintenir.

Là où vous voulez émuler les API Windows, vous pouvez inclure deux sections distinctes en utilisant des **$IFDEF** pour indiquer les sections de code qui s'appliquent spécifiquement à Windows ou à Linux.

Portage de votre application

Si vous portez une application que vous souhaitez exécuter sous Windows et sous Linux, vous devez modifier votre code ou utiliser des **$IFDEF** pour définir les sections de code qui s'appliquent spécifiquement à Windows ou à Linux.

Les étapes générales pour porter votre application VCL vers CLX sont les suivantes :

1. Ouvrez le projet contenant l'application que vous voulez modifier dans Delphi.
2. Copiez les fichiers .dfm dans des fichiers .xfm de même nom (par exemple, renommez unit1.dfm en unit1.xfm). Renommez (ou utilisez un **$IFDEF**) la référence au fichier .dfm dans le ou les fichiers unité de `{$R *.dfm}` en `{$R *.xfm}`. (Le fichier .xfm fonctionnera dans Kylix comme dans Delphi.)

   Par exemple, changez la référence aux fiches dans la section **implementation** de

   ```
   {$R *.dfm}
   ```

   en

   ```
   {$R *.xfm}
   ```
3. Modifiez (ou spécifiez avec un **$IFDEF**) toutes les clauses **uses** pour qu'elles fassent référence aux unités correctes dans CLX. (Pour plus d'informations, consultez "Comparaison entre les unités CLX et VCL" à la page 10-11.)

   Par exemple, changez la clause **uses** suivante d'une application Windows

   ```
   uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs,
   StdCtrls;
   ```

   en la clause ci-dessous pour une application CLX :

   ```
   uses Windows, Messages, SysUtils, Variants, Classes, QForms, QControls, QStdCtrls;
   ```
4. Enregistrez le projet et rouvrez-le. Désormais, la palette de composants propose des composants pouvant être utilisés dans les applications CLX.

Remarque Seuls certains composants non visuels de Windows peuvent être utilisés dans les applications CLX. Les pages ADO, BDE, Système, DataSnap, InterBase, Internet Express, Site Express, FastNet, QReport, COM+, Services Web et Serveurs de la palette de composants incluent des fonctionnalités qui ne fonctionneront que dans les applications CLX sous Windows. Si vous prévoyez de compiler votre application également sous Linux, n'utilisez pas les composants proposés dans ces pages, ou utilisez des **$IFDEF** pour marquer les sections de code qui les utilisent comme étant uniquement destinées à Windows.

5. Réécrivez le code qui ne nécessite pas de dépendances Windows afin de rendre le code plus indépendant de la plate-forme. Utilisez pour cela les routines et constantes de la bibliothèque d'exécution. (Pour plus d'informations, consultez "Ecriture de code portable" à la page 10-19.)
6. Trouvez une fonctionnalité équivalente pour les caractéristiques qui sont différentes sous Linux. Utilisez des **$IFDEF** (avec modération cependant) pour délimiter les informations propres à Windows. (Pour plus d'informations, consultez "Utilisation des directives conditionnelles" à la page 10-20.)

   Par exemple, vous pouvez délimiter par des **$IFDEF** le code propre à la plate-forme dans vos fichiers source :

```
[$IFDEF MSWINDOWS]
IniFile.LoadfromFile('c:\x.txt');
[$ENDIF]

[$IFDEF LINUX]
IniFile.LoadfromFile('/home/name/x.txt');
[$ENDIF]
```

7. Recherchez les références aux noms de chemins dans tous les fichiers du projet.
   - Les noms de chemins dans Linux utilisent une barre oblique droite / comme délimiteur (par exemple, /usr/lib) et les fichiers peuvent être installés dans des répertoires différents sur le système Linux. Utilisez la constante PathDelim (dans SysUtils) pour spécifier le délimiteur de chemin adapté au système. Déterminez l'emplacement correct pour tous les fichiers dans Linux.
   - Modifiez les références à des lettres de lecteurs (par exemple, C:\) et le code qui reconnaît les lettres de lecteurs en recherchant un deux-points en position 2 dans la chaîne. Utilisez la constante DriveDelim (dans SysUtils) pour spécifier l'emplacement selon des termes adaptés au système.
   - Là où vous spécifiez plusieurs chemins, remplacez le séparateur de chemin point-virgule (;) par un deux-points (:). Utilisez la constante PathSep (dans SysUtils) pour spécifier le séparateur de chemin adapté au système.
   - Comme les noms de fichiers sont sensibles à la casse dans Linux, assurez-vous que votre application ne modifie pas la casse des noms de fichiers ou n'assume pas une certaine casse.
8. Compilez, testez et déboguez votre application.

Pour transférer l'application sous Linux :

1 Transférez sur votre ordinateur Linux les fichiers source et autres fichiers liés au projet de votre application Windows Delphi Windows. (Vous pouvez partager des fichiers source entre Linux et Windows si vous souhaitez que le programme s'exécute sur les deux plates-formes. Vous pouvez également transférer les fichiers à l'aide d'un outil comme ftp en mode ASCII.)

   Les fichiers source doivent inclure vos fichiers d'unités (fichiers .pas), le fichier de projet (fichier .dpr) et tous les fichiers de paquets (fichiers .dpk). Les fichiers liés au projet incluent les fichiers de fiches (fichiers .xfm), les fichiers de ressources (fichiers .res) et les fichiers d'options de projet (fichiers .dof – dans Kylix, cela devient des fichiers .kof). Si vous souhaitez compiler votre application uniquement à partir de la ligne de commande (plutôt que d'utiliser l'EDI), le fichier de configuration (fichier .cfg – dans Kylix, cela devient un fichier .conf) sera nécessaire.

2 Ouvre le projet dans Kylix. Vous recevrez des avertissements à propos des fonctionnalités spécifiques à Windows qui sont utilisées.

3 Compilez le projet en utilisant Kylix. Examinez tous les messages d'erreur pour repérer les endroits où des modifications supplémentaires seront nécessaires.

CLX et VCL

Kylix utilise la bibliothèque de composants multiplates-formes (CLX) Borland à la place de la bibliothèque de composants visuels (VCL). Dans la VCL, de nombreux contrôles permettent d'accéder facilement aux contrôles Windows. De même, CLX fournit un accès aux widgets (de window + gadget) Qt dans les bibliothèques partagées Qt. Delphi inclut et CLX et la VCL.

CLX ressemble beaucoup à la VCL. La plupart des noms de composants sont les mêmes et de nombreuses propriétés portent les mêmes noms. En outre, CLX sera disponible sous Windows tout autant que la VCL (vérifiez la dernière version de Delphi pour déterminer les disponibilités).

Les composants de CLX peuvent être groupés en parties comme suit :

Tableau 10.2 Parties de CLX

Parties	Description
VisualCLX	Composants et graphiques d'interface utilisateur multiplates-formes natifs. Les composants de cette partie peuvent différer sous Linux et Windows.
DataCLX	Composants d'accès aux données client. Les composants de cette partie forment un sous-ensemble des ensembles de données locaux, client/serveur et n niveaux basés sur le client. Le code est identique sous Linux et Windows.

Tableau 10.2 Parties de CLX (suite)

Parties	Description
NetCLX	Composants Internet incluant Apache DSO et CGI WebBroker. Ce sont les mêmes sous Linux et Windows.
RTL	Bibliothèque d'exécution jusqu'à Classes.pas inclus. Le code est identique sous Linux et Windows.

Les widgets de VisualCLX remplacent les contrôles Windows. Dans CLX, *TWidgetControl* remplace le *TWinControl de la VCL*. Les autres composants (comme *TScrollingWidget*) portent les mêmes noms. Vous n'avez toutefois pas besoin de remplacer les occurrences de *TWinControl* par *TWidgetControl*. Des déclarations de types, comme

```
TWinControl = TWidgetControl;
```

sont présentes dans le fichier source QControls.pas pour simplifier le partage du code source. *TWidgetControl* et ses descendants possèdent tous une propriété *Handle* qui référence l'objet Qt, et une propriété *Hooks* qui référence les objets de liens gérant le mécanisme des événements.

Les noms et les emplacements des unités de certaines classes sont différents pour CLX. Vous devrez modifier les clauses **uses** pour éliminer les références aux unités qui n'existent pas dans CLX et pour les remplacer par les noms d'unités CLX. (La plupart des fichiers de projet et les sections d'interface de la plupart des unités contiennent une clause **uses**. La section d'implémentation d'une unité peut également contenir sa propre clause **uses**.)

Différences de CLX

Bien que la plus grande partie de CLX soit mise en œuvre de la même façon que la VCL, certaines fonctionnalités sont mises en œuvre autrement. Cette section présente les différences entre les mises en œuvre de CLX et de la VCL auxquelles vous devez faire attention lorsque vous écrivez des applications multiplates-formes.

Présentation visuelle

L'environnement visuel de Linux est quelque peu différent de celui de Windows. L'aspect des boîtes de dialogue peut dépendre du gestionnaire de fenêtres utilisé (par exemple, selon qu'il s'agisse de KDE ou de Gnome).

Styles

Il est possible d'utiliser des "styles" au niveau application en plus des propriétés *OwnerDraw*. Vous pouvez utiliser la propriété *TApplication.Style* pour définir l'aspect visuel des éléments graphiques d'une application. Avec des styles, un widget ou une application peut prendre une toute nouvelle apparence. Vous pouvez toujours utiliser le dessin propriétaire sous Linux mais l'utilisation des styles est recommandée.

Variants

Tout le code des tableaux protégé pour les variants qui se trouvait dans System se trouve dans deux nouvelles unités :

- Variants.pas
- VarUtils.pas

Le code dépendant du système d'exploitation est maintenant isolé dans VarUtils.pas, et il contient également des versions génériques de tout ce qui est nécessaire à Variants.pas. Si vous convertissez une application VCL qui inclut des appels à Windows en une application CLX, vous devrez remplacer ceux-ci par des appels à VarUtils.pas.

Si vous souhaitez utiliser des variants, vous devrez inclure l'unité Variants dans votre clause **uses**.

VarIsEmpty effectue un test simple sur *varEmpty* pour vérifier si un variant est effacé, et sous Linux vous devez utiliser la fonction *VarIsClear* pour effacer un variant.

Gestionnaire de données variants personnalisées

Vous pouvez définir des types de données personnalisés pour les variants. Cela introduit une surcharge d'opérateurs lorsque le type est affecté au variant. Pour créer un nouveau type de variant, descendez de la classe *TCustomVariantType* et instanciez votre nouveau type de variant.

Considérons, par exemple, VarCmplx.pas. Cette unité met en œuvre une prise en charge des mathématiques complexes par l'intermédiaire de variants personnalisés. Elle prend en charge les opérations de variants suivantes : addition, soustraction, multiplication, division (pas de division entière) et négation. Elle gère également les conversions entre : SmallInt, Integer, Single, Double, Currency, Date, Boolean, Byte, OleStr et String. Toute conversion flottante/ordinale provoquera la perte de la partie imaginaire de la valeur complexe.

Registre

Linux n'utilise pas de registre pour stocker les informations de configuration. Vous devez utiliser des fichiers texte de configuration et des variables d'environnement à la place du registre. Les fichiers de configuration système sous Linux sont souvent installés dans /etc, par exemple /etc/hosts. Les autres profils utilisateur sont installés dans des fichiers cachés (dont le nom commence par un point), comme .bashrc, qui contient les paramètres du shell bash ou .XDefaults, qui sert à définir des valeurs par défaut pour les programmes X.

Le code dépendant du registre peut être modifié pour utiliser à la place un fichier texte de configuration local stocké, par exemple, dans le même répertoire que l'application. Ecrire une unité contenant toutes les fonctions de registre mais en détournant toutes les sorties vers un fichier de configuration local est un moyen que vous pouvez utiliser pour gérer une dépendance par rapport au registre.

Pour mémoriser des informations à un emplacement global sous Linux, vous pouvez stocker un fichier de configuration global dans le répertoire racine. Toutes vos applications pourront alors accéder au même fichier de configuration. Vous devez toutefois vous assurer que les permissions du fichier et les droits d'accès sont correctement définis.

Vous pouvez également utiliser des fichiers ini dans des applications multiplates-formes. Toutefois, dans CLX, vous devrez utiliser *TMemIniFile* au lieu de *TRegIniFile.*

Autres différences

La mise en œuvre de CLX possède également d'autres différences affectant le fonctionnement de votre application. Cette section décrit certaines de ces différences.

ToggleButton ne peut pas être inversée par la touche Entrée. Dans Kylix, l'utilisation de la touche Entrée ne simule pas un événement clic comme dans Delphi.

TColorDialog ne possède pas de propriété *TColorDialog.Options à initialiser*. Par conséquent, vous ne pouvez pas personnaliser l'apparence et la fonctionnalité de la boîte de dialogue de sélection de couleur. De plus, *TColorDialog* n'est pas toujours modale. Sur Kylix, vous pouvez manipuler la barre de titre d'une application avec une boîte de dialogue modale (c'est-à-dire que vous pouvez sélectionner la fiche parent de la boîte de dialogue de couleur et effectuer des opérations telles qu'un agrandissement lorsque la boîte de dialogue de couleur est ouverte).

A l'exécution, les listes déroulantes fonctionnent différemment sur Kylix et sur Delphi. Sur Kylix, vous pouvez ajouter un élément à une liste déroulante en tapant le texte et en appuyant sur Entrée dans la zone de saisie de la liste déroulante. Vous pouvez désactiver cette fonctionnalité en initialisant *InsertMode* à ciNone. Il est également possible d'ajouter des éléments vides (aucune chaîne de caractères) à la liste déroulante. De plus, si vous maintenez appuyée la flèche vers le bas, l'affichage ne s'arrête pas au dernier élément de la liste déroulante. Il effectue une boucle en revenant au début de la liste.

TCustomEdit ne met pas en œuvre *Undo*, *ClearUndo* ou *CanUndo.* Il n'existe donc aucune méthode pour annuler les saisies par programme. Mais les utilisateurs d'une application peuvent annuler leurs saisies dans une boîte de saisie (*TEdit*) lors de l'exécution en cliquant avec le bouton droit de la souris sur cette boîte puis en choisissant la commande Undo.

Dans un événement *OnKeyDown* ou *KeyUp*, la valeur de la touche Entrée est 13 sous Windows. Sous Linux, cette valeur est 4100. Si vous testez la valeur numérique "codée en dur" d'une touche, comme lorsque vous testez la valeur 13 pour la touche Entrée, vous devez modifier ce test lorsque vous portez une application Delphi vers Kylix.

Il existe d'autres différences. Reportez-vous à la documentation en ligne de CLX pour des détails sur tous les objets CLX ou, dans les versions de Delphi qui

incluent le code source, reportez-vous au code que vous trouverez dans le répertoire ..\Delphi6\Source\VCL\CLX.

Fonctionnalités manquantes dans CLX

Lorsque vous utilisez CLX à la place de la VCL, beaucoup d'objets sont identiques. Mais, ces objets peuvent avoir perdu certaines fonctionnalités (comme des propriétés, des méthodes, ou des événements). Les fonctionnalités générales suivantes sont absentes de CLX :

- Propriétés bidirectionnelles (*BidiMode*) pour les entrées ou les sorties de texte de la droite vers la gauche
- Propriétés de biseau génériques sur les contrôles courants (remarquez que certains objets possèdent toujours des propriétés de biseau)
- Propriétés et méthodes d'ancrage
- Fonctionnalités de compatibilité ascendante comme les composants de l'onglet Win3.1 et *Ctl3D*
- *DragCursor* et *DragKind* (mais le glisser-déplacer est inclus)

Fonctionnalités non portées

Certaines fonctionnalités propres à Windows, prises en charge par Delphi, ne sont pas portées directement vers les environnements Linux. Des fonctionnalités comme COM, ActiveX, OLE, BDE et ADO sont dépendantes de la technologie Windows et ne sont pas disponibles sur Kylix. Le tableau suivant présente les fonctionnalités qui sont différentes sur les deux plates-formes, avec la fonctionnalité Kylix équivalente lorsqu'elle est disponible.

Tableau 10.3 Fonctionnalités modifiées ou différentes

Fonctionnalité Delphi/Windows	Fonctionnalité Kylix/Linux
Composants ADO	Composants de bases de données classiques
Serveurs Automation	Non disponible
BDE	dbExpress et composants de bases de données classiques
Composants COM+ (incluant ActiveX)	Non disponible
DataSnap	Non encore disponible
FastNet	Non disponible
Internet Express	Non encore disponible
Composants existants (comme les éléments de la page Win 3.1 de la palette de composants)	Non disponible

Tableau 10.3 Fonctionnalités modifiées ou différentes (suite)

Fonctionnalité Delphi/Windows	Fonctionnalité Kylix/Linux
L'interface de programmation d'application de messagerie (MAPI, Messaging Application Programming Interface) incluant une bibliothèque standard des fonctions de messagerie Windows.	SMTP/POP3 vous permet d'envoyer, de recevoir et d'enregistrer des messages électroniques
Quick Reports	Non disponible
Services Web (SOAP)	Non encore disponible
WebSnap	Non encore disponible
Appels de l'API Windows	Méthodes CLX, appels Qt, appels libc ou appels à d'autres bibliothèques système
Messages Windows	Evénements Qt
Winsock	Sockets BSD

Les équivalents Linux des DLL Windows sont les bibliothèques d'objets partagés (fichiers .so), qui contiennent du code indépendant de la position (PIC). Il en résulte que :

- Les variables se référant à une adresse absolue en mémoire (au moyen de la directive **absolute**) ne sont pas autorisées.
- Les références mémoire globales et les appels à des fonctions externes sont relatifs au registre EBX, qui doit être préservé d'un appel à l'autre.

Vous ne devez vous préoccuper des références mémoire globales et des appels à des fonctions externes que si vous utilisez l'assembleur — Kylix ou Delphi génère le code correct. (Pour plus d'informations, consultez "Inclusion de code assembleur inline" à la page 10-23.)

Les modules et les paquets des bibliothèques Kylix sont mis en œuvre à l'aide de fichiers .so.

Comparaison entre les unités CLX et VCL

Tous les objets de la VCL ou de CLX sont définis dans des fichiers d'unités (fichiers source .pas). Par exemple, vous trouverez la mise en œuvre de *TObject* dans l'unité System, et l'unité Classes définit la classe de base *TComponent*. Lorsque vous déposez un objet sur une fiche ou que vous utilisez un objet dans votre application, le nom de l'unité est ajouté à la clause **uses** qui indique au compilateur quelles unités doivent être liées au projet.

Cette section contient des tableaux qui présentent les unités CLX avec l'unité comparable de la VCL, les unités qui existent uniquement pour CLX et celles qui existent uniquement pour VCL.

Le tableau suivant présente les unités VCL et CLX comparables :

Tableau 10.4 Unités VCL et leurs équivalents CLX

Unités VCL	Unités CLX
ActnList	QActnList
Buttons	QButtons
CheckLst	QCheckLst
Classes	Classes
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Consts	Consts, QConsts et RTLConsts
Contnrs	Contnrs
Controls	QControls
DateUtils	DateUtils
DB	DB
DBActns	QDBActns
DBClient	DBClient
DBCommon	DBCommon
DBConnAdmin	DBConnAdmin
DBConsts	DBConsts
DBCtrls	QDBCtrls
DBGrids	QDBGrids
DBLocal	DBLocal
DBLocalS	DBLocalS
DBLogDlg	DBLogDlg
DBXpress	DBXpress
Dialogs	QDialogs
DSIntf	DSIntf
ExtCtrls	QExtCtrls
FMTBCD	FMTBCD
Forms	QForms
Graphics	QGraphics
Grids	QGrids
HelpIntfs	HelpIntfs
ImgList	QImgList
IniFiles	IniFiles
Mask	QMask
MaskUtils	MaskUtils
Masks	Masks
Math	Math
Menus	QMenus
Midas	Midas

Tableau 10.4 Unités VCL et leurs équivalents CLX (suite)

Unités VCL	Unités CLX
MidConst	MidConst
Printers	QPrinters
Provider	Provider
Qt	Qt
Search	QSearch
Sockets	Sockets
StdActns	QStdActns
StdCtrls	QStdCtrls
SqlConst	SqlConst
SqlExpr	SqlExpr
SqlTimSt	SqlTimSt
SyncObjs	SyncObjs
SysConst	SysConst
SysInit	SysInit
System	System
SysUtils	SysUtils
Types	Types et QTypes
TypInfo	TypInfo
Variants	Variants
VarUtils	VarUtils

Les unités suivantes existent dans CLX mais pas dans la VCL :

Tableau 10.5 Unités de CLX absentes de la VCL

Unité	Description
DirSel	Sélection de répertoire
QStyle	Aspect graphique de l'interface utilisateur

Les unités VCL Windows suivantes ne sont pas incluses dans CLX, principalement parce qu'elles concernent des fonctionnalités propres à Windows qui ne sont pas disponibles sous Linux comme ADO, COM et BDE. La raison de l'exclusion de chaque unité est précisée.

Tableau 10.6 Unités de la VCL uniquement

Unité	Raison de son exclusion
ADOConst	Pas de fonctionnalité ADO
ADODB	Pas de fonctionnalité ADO
AppEvnts	Pas d'objet TApplicationEvent
AxCtrls	Pas de fonctionnalité COM
BdeConst	Pas de fonctionnalité BDE
ComStrs	Pas de fonctionnalité COM

Tableau 10.6 Unités de la VCL uniquement (suite)

Unité	Raison de son exclusion
ConvUtils	Nouvelle fonctionnalité de Delphi 6
CorbaCon	Pas de fonctionnalité Corba
CorbaStd	Pas de fonctionnalité Corba
CorbaVCL	Pas de fonctionnalité Corba
CtlPanel	Pas de prise en charge du Panneau de configuration Windows
DataBkr	Susceptible d'apparaître plus tard en montée en gamme
DBCGrids	Pas de fonctionnalité BDE
DBExcept	Pas de fonctionnalité BDE
DBInpReq	Pas de fonctionnalité BDE
DBLookup	Obsolète
DbOleCtl	Pas de fonctionnalité COM
DBPWDlg	Pas de fonctionnalité BDE
DBTables	Pas de fonctionnalité BDE
DdeMan	Pas de fonctionnalité DDE
DRTable	Pas de fonctionnalité BDE
ExtActns	Nouvelle fonctionnalité de Delphi 6
ExtDlgs	Pas de boîtes de dialogue d'images
FileCtrl	Obsolète
ListActns	Nouvelle fonctionnalité de Delphi 6
MConnect	Pas de fonctionnalité COM
Messages	Domaine propre à Windows
MidasCon	Obsolète
MPlayer	Lecteur de médias propre à Windows
Mtsobj	Pas de fonctionnalité COM
MtsRdm	Pas de fonctionnalité COM
Mtx	Pas de fonctionnalité COM
mxConsts	Pas de fonctionnalité COM
ObjBrkr	Susceptible d'apparaître plus tard en montée en gamme
OleConstMay	Pas de fonctionnalité COM
OleCtnrs	Pas de fonctionnalité COM
OleCtrls	Pas de fonctionnalité COM
OLEDB	Pas de fonctionnalité COM
OleServer	Pas de fonctionnalité COM
Outline	Obsolète
Registry	Prise en charge du registre Windows
ScktCnst	Remplacée par des sockets
ScktComp	Remplacée par des sockets
SConnect	Protocoles de connexion non pris en charge
StdConvs	Nouvelle fonctionnalité de Delphi 6
SvcMgr	Prise en charge des services NT

Tableau 10.6 Unités de la VCL uniquement (suite)

Unité	Raison de son exclusion
Tabnotbk	Obsolète
Tabs	Obsolète
ToolWin	Pas de fonctionnalité d'ancrage
VarCmplx	Nouvelle fonctionnalité de Delphi 6
VarConv	Nouvelle fonctionnalité de Delphi 6
VCLCom	Pas de fonctionnalité COM
WebConst	Constantes propres à Windows
Windows	Propre à Windows (API)

Différences dans les constructeurs d'objets CLX

Lorsqu'un objet CLX est créé, soit implicitement dans le concepteur de fiches en plaçant l'objet sur la fiche, soit explicitement dans le code en utilisant la méthode *Create* de l'objet, une instance du widget sous-jacent associé est également créée. Cette instance est détenue par l'objet CLX. Lorsque l'objet CLX est supprimé au moyen d'un appel à la méthode *Free* ou qu'il est automatiquement supprimé par le conteneur parent de l'objet CLX, le widget sous-jacent est également supprimé. C'est le même type de fonctionnalité qu'offre la VCL dans les applications Windows.

Lorsque vous créez explicitement un objet CLX dans du code, en faisant des appels dans la bibliothèque d'interface Qt comme QWidget_Create(), vous créez une instance d'un widget Qt dont le propriétaire n'est pas un objet CLX. Cela transmet l'instance d'un widget Qt existant à l'objet CLX à utiliser pendant sa construction. Cet objet CLX n'est pas propriétaire du widget Qt qui lui est transmis. Par conséquent, lorsque vous appelez la méthode *Free* après avoir créé l'objet de cette manière, seul l'objet CLX est détruit, et non l'instance du widget Qt sous-jacent. Ce comportement est différent de celui de la VCL.

Certains objets CLX vous permettent d'assumer la propriété du widget sous-jacent en utilisant la méthode *OwnHandle*. Après l'appel à *OwnHandle*, si vous supprimez l'objet CLX, le widget sous-jacent est aussi supprimé.

Partage des fichiers source entre Windows et Linux

Si vous voulez que votre application s'exécute sous Windows et sous Linux, vous pouvez partager les fichiers source en les rendant accessibles aux deux systèmes d'exploitation. Vous pouvez procéder de nombreuses manières, par exemple en plaçant les fichiers sur un serveur accessible aux deux ordinateurs ou en utilisant Samba sur la machine Linux pour permettre d'accéder aux fichiers au travers d'un partage réseau Microsoft pour Linux et Windows. Vous pouvez choisir de conserver le source sous Linux et de créer un partage sous Linux. Ou vous pouvez conserver le source sous Windows et créer un partage sous Windows pour que la machine Linux puisse y accéder.

Vous pouvez continuer à développer et compiler le fichier dans Kylix en utilisant des objets qui sont pris en charge par la VCL et CLX. Lorsque vous avez fini, vous pouvez compiler sous Linux et Windows.

Les fichiers fiche (fichiers .dfm dans Delphi) sont appelés fichiers .xfm dans Kylix. Si vous créez une nouvelle application CLX dans Delphi ou Kylix, un .xfm est créé à la place d'un .dfm. Dans le cas des applications multiplates-formes, le .xfm fonctionnera dans Delphi comme dans Kylix.

Différences d'environnement entre Windows et Linux

Habituellement, multiplate-forme signifie qu'une application peut s'exécuter théoriquement sans aucune modification sous les systèmes d'exploitation Windows et Linux. Le tableau suivant présente les différences entre les systèmes d'exploitation Linux et Windows.

Tableau 10.7 Différences dans les environnements d'exploitation Linux et Windows

Différence	Description
Nom de fichier sensible à la casse	Dans Linux, une lettre majuscule n'est *pas* identique à une lettre minuscule. Le fichier Test.txt n'est *pas* identique au fichier test.txt. Vous devez faire très attention aux lettres majuscules dans les noms de fichiers sous Linux.
Caractères de fin de ligne	Dans Windows, les lignes d'un texte se terminent par CR/LF (c'est-à-dire, ASCII 13 + ASCII 10), alors qu'elles se terminent par LF dans Linux. Bien que l'éditeur de code dans Kylix soit capable de gérer la différence, vous devez être conscient de cela lorsque vous importez du code depuis Windows.
Caractère de fin de fichier	Dans DOS et dans Windows, le caractère de valeur #26 (Ctrl-Z) est considéré comme la fin du fichier texte, même s'il existe des données dans le fichier après ce caractère. Linux n'a pas de caractère de fin de fichier particulier ; les données du texte se terminent à la fin du fichier.
Fichiers de traitement par lots/scripts shell	L'équivalent Linux du fichier .bat est le script shell. Un script est un texte contenant des instructions enregistrées et exécutables à l'aide de la commande `chmod +x <fichierscript>`. Pour l'exécuter, tapez son nom. (Le langage de script dépend du shell que vous utilisez sous Linux. Bash est communément utilisé.)
Confirmation de commande	Sous DOS ou Windows, si vous tentez de détruire un fichier ou un dossier, une confirmation est demandée ("Etes-vous sûr ?"). Linux ne demande généralement pas de confirmation ; il effectue simplement l'opération. Cela accentue le risque de destruction accidentelle d'un fichier ou de tout un système de fichiers. Il n'existe aucun moyen d'annuler une suppression sous Linux à moins d'avoir sauvegardé un fichier sur un autre support.
Retour de commande	Si une commande réussit sous Linux, son invite est réaffichée sans message d'état.
Commutateurs de commande	Linux utilise un tiret simple (-) pour représenter les commutateurs de commande ou un tiret double (--) pour les options à plusieurs caractères, là où DOS utilise une barre oblique droite (/) ou un tiret (-).

Tableau 10.7 Différences dans les environnements d'exploitation Linux et Windows (suite)

Différence	Description
Fichiers de configuration	Sous Windows, la configuration s'effectue dans le registre ou dans des fichiers comme autoexec.bat. Sous Linux, les fichiers de configuration sont créés sous la forme de fichiers cachés dont le nom commence par un point (.). La plupart sont installés dans le répertoire /etc ou dans votre répertoire initial. Linux utilise également des variables d'environnement telles que LD_LIBRARY_PATH (chemin de recherche pour les bibliothèques). Autres variables d'environnement importantes : HOMEVotre répertoire initial (/home/sam) TERMType de terminal (xterm, vt100, console) SHELLChemin vers votre interpréteur (/bin/bash) USERVotre identifiant de session (sfuller) PATHListe de recherche pour les programmes Ces variables sont spécifiées dans l'interpréteur ou dans les fichiers rc tels que .bashrc.
DLL	Sous Linux, vous utilisez des fichiers d'objets partagés (.so). Sous Windows, il s'agit de bibliothèques de liens dynamiques (DLL).
Lettres de lecteurs	Linux ne possède pas de lettres de lecteurs. Un nom de chemin Linux est, par exemple, de la forme /lib/security. Voir DriveDelim dans la bibliothèque d'exécution.
Exceptions	Les exceptions du système d'exploitation sont appelées signaux sous Linux.
Fichiers exécutables	Sous Linux, les fichiers exécutables ne nécessitent pas d'extension. Sous Windows, les fichiers exécutables possèdent l'extension exe.
Extensions de nom de fichier	Linux n'utilise pas d'extensions de nom de fichier pour identifier les types de fichiers ou associer les fichiers à des applications.
Permissions de fichier	Sous Linux, des permissions de lecture, d'écriture et d'exécution sont affectées aux fichiers (et aux répertoires) pour le propriétaire du fichier, le groupe et les autres. Par exemple, `-rwxr-xr-x` signifie, de la gauche vers la droite : `-` est le type de fichier (`-` = fichier ordinaire, `d` = répertoire, `l` = lien) ; `rwx` représente les permissions pour le propriétaire du fichier (lecture, écriture, exécution), `r-x` représente les permissions pour le groupe du propriétaire du fichier (lecture, exécution) et `r-x` représente les permissions pour tous les autres utilisateurs (lecture, exécution). L'utilisateur root (super-utilisateur) peut outrepasser ces permissions. Vous devez vous assurer que votre application s'exécute sous l'utilisateur correct et dispose d'un droit d'accès correct aux fichiers requis.
Utilitaire Make	L'utilitaire make de Borland n'est pas disponible sur la plate-forme Linux. A la place, vous pouvez utiliser l'utilitaire make GNU de Linux.

Tableau 10.7 Différences dans les environnements d'exploitation Linux et Windows (suite)

Différence	Description
Multitâche	Linux prend totalement en charge le fonctionnement multitâche. Vous pouvez exécuter en même temps plusieurs programmes (appelés processus sous Linux). Vous pouvez lancer des processus en arrière-plan (en utilisant & après la commande) et continuer immédiatement à travailler. Linux vous permet également de disposer de plusieurs sessions.
Noms de chemins	Linux utilise une barre oblique droite (/) là où DOS utilise une barre oblique inverse (\). Une constante PathDelim peut être utilisée pour spécifier le caractère adapté à la plate-forme. Voir PathDelim dans la bibliothèque d'exécution.
Chemin de recherche	Lors de l'exécution de programmes, Windows consulte toujours en premier le répertoire en cours, puis il examine la variable d'environnement PATH. Linux ne consulte jamais le répertoire en cours mais recherche uniquement les répertoires énumérés dans PATH. Pour exécuter un programme du répertoire en cours, vous devez généralement taper ./ avant. Vous pouvez également modifier votre PATH pour inclure ./ comme premier chemin à rechercher.
Séparateur de chemin de recherche	Windows utilise le point-virgule comme séparateur de chemin de recherche.
Liens symboliques	Sous Linux, un lien symbolique est un fichier spécial qui pointe sur un autre fichier du disque. En plaçant les liens symboliques dans le répertoire global bin qui contient les principaux fichiers de votre application, vous n'aurez pas à modifier le chemin de recherche système. Un lien symbolique se crée à l'aide de la commande ln (link). Windows possède des raccourcis pour le bureau de l'interface utilisateur graphique. Pour mettre un programme à la disposition de la ligne de commande, les programmes d'installation Windows modifient généralement le chemin de recherche système.

Structure de répertoires sous Linux

Les répertoires sont différents sous Linux. N'importe quel fichier ou périphérique peut être monté n'importe où dans le système de fichiers.

Remarque Les noms de chemins Linux utilisent des barres obliques droites contrairement à Windows qui utilise des barres obliques inverses. La barre oblique initiale représente le répertoire racine.

Voici quelques répertoires couramment utilisés dans Linux.

Tableau 10.8 Répertoires Linux courants

Répertoire	Contenu
/	Le répertoire racine ou de niveau supérieur de tout le système de fichiers Linux
/root	Le système de fichier root ; le répertoire initial du super-utilisateur
/bin	Commandes, utilitaires

Tableau 10.8 Répertoires Linux courants (suite)

Répertoire	Contenu
/sbin	Utilitaires système
/dev	Périphériques représentés comme des fichiers
/lib	Bibliothèques
/home/ nom_utilisateur	Fichiers appartenant à l'utilisateur où nom_utilisateur est le nom de connexion de l'utilisateur
/opt	Facultatif
/boot	Noyau appelé lorsque le système démarre
/etc	Fichiers de configuration
/usr	Applications, programmes. Inclut généralement des répertoires comme / usr/spool, /usr/man, /usr/include, /usr/local
/mnt	Autre support installé sur le système, comme un CD-ROM ou un lecteur de disquettes
/var	Journaux, messages, fichiers de spool
/proc	Système de fichiers virtuels et statistiques du système
/tmp	Fichiers temporaires

Remarque Les différentes distributions de Linux installent parfois les fichiers à différents emplacements. Un programme utilitaire peut être installé dans /bin avec une distribution Red Hat et dans /usr/local/bin avec une distribution Debian.

Reportez-vous à www.pathname.com pour des détails supplémentaires sur l'organisation du système de fichiers hiérarchique UNIX/Linux. Vous pourrez également consulter le document *Filesystem Hierarchy Standard*.

Ecriture de code portable

Si vous écrivez des applications multiplates-formes destinées à être exécutées sous Windows et Linux, vous pouvez écrire du code qui se compilera sous différentes conditions. En utilisant la compilation conditionnelle, vous pouvez conserver votre codage Windows, en prenant toujours en considération les différences du système d'exploitation Linux.

Pour créer des applications facilement portables entre Windows et Linux, pensez à

- réduire ou isoler les appels aux API propres à une plate-forme (Win32 ou Linux) ; utilisez à la place les méthodes CLX.
- éliminer les constructions de messagerie Windows (PostMessage, SendMessage) dans une application.
- utiliser *TMemIniFile* à la place de *TRegIniFile*.
- respecter et conserver la distinction minuscule/majuscule pour les noms de fichiers et de répertoires.

- porter tout code TASM assembleur externe. L'assembleur GNU "as" ne prend pas en charge la syntaxe TASM. (Consultez "Inclusion de code assembleur inline" à la page 10-23.)

Essayez d'écrire le code de manière à utiliser les routines de la bibliothèque d'exécution indépendantes de la plate-forme et utilisez les constantes de System, SysUtils et des autres unités de la bibliothèque d'exécution. Par exemple, utilisez la constante PathDelim pour que votre code ne soit pas affecté par les différences de plate-forme de '/' par rapport à '\'.

Un autre exemple concerne l'utilisation de caractères multi-octets sur les deux plates-formes. Traditionnellement, le code Windows attend uniquement 2 octets par caractère multi-octet. Sous Linux, les caractères multi-octets peut comporter beaucoup plus d'octets (jusqu'à 6 octets pour UTF-8). Les deux plates-formes peuvent être adaptées en utilisant la fonction StrNextChar dans SysUtils. Le code Windows existant suivant

```
while p^ <> #0 do
begin
   if p^ in LeadBytes then
      inc(p);
   inc(p);
end;
```

peut être remplacé par un code indépendant de la plate-forme comme celui-ci :

```
while p^ <> #0 do
begin
  if p^ in LeadBytes then
    p := StrNextChar(p)
  else
    inc(p);
end;
```

Cet exemple est portable d'une plate-forme à l'autre et prend en charge les caractères multioctets sur plus de 2 octets, mais il évite toujours de nuire aux performances en appelant une procédure pour les environnement régionaux non multi-octets.

Si l'utilisation de fonctions de bibliothèque d'exécution n'est pas une solution envisageable, essayez d'isoler le code propre à une plate-forme dans une partie de votre routine ou une sous-routine. Essayez de limiter le nombre de blocs **$IFDEF** pour conserver la lisibilité et la portabilité du code source. Le symbole conditionnel WIN32 n'est pas défini sous Linux. Le symbole conditionnel LINUX est défini, pour indiquer que le code source est compilé pour la plate-forme Linux.

Utilisation des directives conditionnelles

L'utilisation des directives de compilation **$IFDEF** est une méthode recommandée pour introduire des conditions dans votre code pour les plates-formes Windows et Linux. Toutefois, comme les **$IFDEF** rendent le code source plus difficile à comprendre et à maintenir, vous devez savoir à quelles occasions il est raisonnable d'utiliser des **$IFDEF**. En envisageant l'utilisation des **$IFDEF**,

les meilleures questions à se poser devraient être "Pourquoi ce code a-t-il besoin d'un **$IFDEF** ?" et "Ce traitement peut-il s'écrire sans **$IFDEF** ?"

Suivez ces lignes directrices pour utiliser des **$IFDEF** dans des applications multiplates-formes :

- N'essayez pas d'utiliser de **$IFDEF** sans absolue nécessité. Les **$IFDEF** dans un fichier source sont uniquement évalués lorsque le code source est compilé. A la différence de C/C++, Delphi ne nécessite pas de sources d'unité (fichiers en-tête) pour compiler un projet. La reconstruction totale de tout le code source est un événement rare pour la plupart des projets Delphi.
- N'utilisez pas de **$IFDEF** dans les fichiers paquet (.dpk). Limitez leur utilisation aux fichiers source. Les créateurs de composants doivent créer deux paquets de conception lors du développement multiplates-formes, et non un seul paquet avec des **$IFDEF**.
- En général, utilisez **$IFDEF** MSWINDOWS pour tester n'importe quelle plate-forme Windows, y compris WIN32. Réservez l'utilisation de **$IFDEF** WIN32 à la distinction entre des plates-formes Windows spécifiques, comme Windows 32 bits et 64 bits. Ne limitez pas votre code à WIN32 à moins d'être sûr qu'il ne fonctionne pas sous WIN64.
- Evitez les tests négatifs du type **$IFNDEF** si ce n'est pas absolument nécessaire. **$IFNDEF** LINUX n'est *pas* équivalent à **$IFDEF** MSWINDOWS.
- Evitez les combinaisons **$IFNDEF/$ELSE**. Utilisez plutôt un test positif (**$IFDEF**) pour une meilleure lisibilité.
- Evitez les clauses **$ELSE** sur des **$IFDEF** de test de plate-forme. Utilisez des blocs **$IFDEF** séparés pour du code LINUX ou MSWINDOWS au lieu de **$IFDEF** LINUX/**$ELSE** ou **$IFDEF** MSWINDOWS/**$ELSE**.

  Par exemple, un ancien code peut contenir

  ```
  {$IFDEF WIN32}
     (Code Windows 32 bits)
  {$ELSE}
     (Code Windows 16 bits)   //!! Linux pourra tomber par erreur dans ce code.
  {$ENDIF}
  ```

  Pour tout code non portable dans des **$IFDEF**, il est préférable que le code source échoue à la compilation que de voir la plate-forme tomber dans une clause **$ELSE** et échouer mystérieusement à l'exécution. Les échecs de compilation sont plus faciles à résoudre que les erreurs à l'exécution.
- Utilisez la syntaxe **$IF** pour les tests compliqués. Remplacez les **$IFDEF** imbriqués par une expression booléenne dans une directive **$IF**. La directive **$IF** doit être terminée par **$IFEND**, et non par **$ENDIF**. Cela vous permet de placer des expressions **$IF** dans des **$IFDEF** pour dissimuler la nouvelle syntaxe **$IF** aux compilateurs précédents.

Toutes les directives conditionnelles sont documentées dans l'aide en ligne. Pour plus d'informations, consultez également la rubrique "Compilation conditionnelle" dans l'aide.

Terminaison des directives conditionnelles

Utilisez la directive **$IFEND** pour terminer les directives conditionnelles **$IF** et **$ELSEIF**. Cela permet de dissimuler les blocs **$IF**/**$IFEND** pour les anciens compilateurs, avec l'utilisation de **$IFDEF**/**$ENDIF**. Les anciens compilateurs ne reconnaîtront pas la directive **$IFEND**. **$IF** peut uniquement se terminer par **$IFEND**. Les anciennes directives (**$IFDEF**, **$IFNDEF**, **$IFOPT**) ne peuvent se terminer que par **$ENDIF**.

Remarque Lors de l'imbrication d'un **$IF** dans un **$IFDEF**/**$ENDIF**, n'utilisez pas **$ELSE** avec **$IF**. Les anciens compilateurs verront le **$ELSE** et penseront qu'il fait partie du **$IFDEF**, ce qui entraînera une erreur de compilation plus loin. Dans cette situation, vous pouvez utiliser {**$ELSEIF** True} comme substitut de {**$ELSE**}, puisque le **$ELSEIF** ne sera pas pris en compte si le **$IF** est pris en compte d'abord, et que les anciens compilateurs ne connaissent pas **$ELSEIF**. La dissimulation des **$IF** pour la compatibilité ascendante est principalement un problème pour les fournisseurs et les développeurs d'applications tierces parties qui souhaitent que leur code s'exécute sur plusieurs versions différentes.

$ELSEIF est une combinaison de **$ELSE** et de **$IF**. La directive **$ELSEIF** vous permet d'écrire des blocs conditionnels à plusieurs parties où un seul des blocs conditionnels sera pris en compte. Par exemple :

```
{$IFDEF faire}
   fais_le
{$ELSEIF  RTLVersion >= 14}
    fais_le_vraiment
{$ELSEIF  chaine = 'oui'}
    bip
{$ELSE}
    dernière chance
{$IFEND}
```

De ces quatre cas, un seul est pris en compte. Si aucune de ces trois premières conditions n'est vraie, la clause **$ELSE** est prise en compte. **$ELSEIF** doit être terminée par **$IFEND**. **$ELSEIF** ne peut pas être placée après **$ELSE**. Les conditions sont évaluées de haut en bas comme une séquence **$IF...$ELSE** normale. Dans l'exemple, si faire n'est pas défini, RTLVersion a la valeur 15 et chaine vaut 'oui', seul le bloc "fais_le_vraiment" sera pris en compte, le bloc "bip" ne le sera pas, même si les conditions sont vraies pour les deux.

Si vous oubliez d'utiliser un **$ENDIF** pour mettre fin à l'un de vos **$IFDEF**, le compilateur signale l'erreur suivante à la fin du fichier source :

```
Missing ENDIF
```

S'il existe plusieurs directives **$IF**/**$IFDEF** dans votre fichier source, il peut être difficile de déterminer laquelle est à l'origine du problème. Kylix ou Delphi renvoie le message d'erreur suivant sur la ligne source de la dernière directive **$IF**/**$IFDEF** du compilateur sans correspondance avec un **$ENDIF**/**$IFEND** :

```
Unterminated conditional directive
```

Vous pouvez commencer à rechercher le problème à cet emplacement.

Emission de messages

La directive de compilation **$MESSAGE** permet au code source d'émettre des conseils, des avertissements et des erreurs exactement comme le compilateur.

```
{$MESSAGE   HINT|WARN|ERROR|FATAL 'chaine texte' }
```

Le type de message est facultatif. Si aucun type de message n'est indiqué, la valeur par défaut est HINT. La chaîne de texte est obligatoire et doit être encadrée par des apostrophes.

Exemples :

`{$MESSAGE 'Bouh!'}` émet un conseil.

`{$Message Hint 'Donner à manger au chat'}` émet un conseil.

`{$Message Warn 'Il semble pleuvoir'}` émet un avertissement.

`{$Message Error 'Non mis en oeuvre'}` émet un message d'erreur, mais la compilation continue.

`{$Message Fatal 'Pan! Tu es mort.'}` émet un message d'erreur et la compilation s'arrête.

Inclusion de code assembleur inline

Si vous incluez du code assembleur inline dans vos applications Windows, il est possible que vous ne puissiez pas utiliser le même code sous Linux à cause des exigences du code indépendant de la position (PIC) *sous Linux*. Les bibliothèques d'objets partagés Linux (équivalentes aux DLL) nécessitent que tout le code soit relogeable en mémoire sans modification. Cela affecte principalement les routines assembleur inline qui utilisent des variables globales ou d'autres adresses absolues, ou qui appellent des fonctions externes.

Pour les unités qui contiennent uniquement du code Pascal Objet, le compilateur génère automatiquement un PIC lorsque c'est nécessaire. Les unités PIC possèdent une extension .dpu (au lieu de .dcu). Il est recommandé de compiler chaque fichier source d'unité Pascal aux formats PIC et non PIC ; utilisez le commutateur de compilation -p pour générer du PIC. Les unités précompilées sont disponibles aux deux formats.

Il est possible que vous souhaitiez coder différemment des routines assembleur selon que vous compilez un exécutable ou une librairie partagée ; utilisez {**$IFDEF** PIC} pour fusionner les deux versions de votre code assembleur. Vous pouvez aussi réécrire la routine en Pascal Objet pour éviter le problème.

Les règles PIC pour le code assembleur inline sont les suivantes :

- PIC nécessite que toutes les références mémoire soient relatives au registre EBX, qui contient le pointeur d'adresse de base du module en cours (dans Linux, ce pointeur s'appelle Global Offset Table ou GOT). Ainsi, plutôt que

  ```
  MOV EAX,GlobalVar
  ```

  utilisez

  ```
  MOV EAX,[EBX].GlobalVar
  ```

- PIC impose de préserver le registre EBX d'un appel à l'autre dans votre code assembleur (comme sous Win32), et de restaurer également le registre EBX *avant* d'effectuer des appels à des fonctions externes (à l'inverse de Win32).
- Même si le code PIC fonctionnera dans des exécutables de base, il peut ralentir les performances et générer plus de code. Vous n'avez pas le choix pour les objets partagés, mais dans les exécutables, vous souhaiterez probablement toujours obtenir le plus haut niveau de performances possible.

Messages et événements système

Les boucles de messages et les événements fonctionnent différemment sous Linux et dans CLX, mais cela concerne principalement l'écriture de composants. La plupart des composants et des éditeurs de propriétés se portent facilement. *TObject.Dispatch* et la syntaxe des méthodes de messages sur les classes fonctionnent très bien sous Linux ; les notifications du système d'exploitation Linux sont toutefois gérées à l'aide d'événements système plutôt que de messages.

Pour créer un gestionnaire d'événements dans une application multiplate-forme, vous pouvez surcharger l'une des méthodes décrites au tableau suivant afin d'écrire votre propre message personnalisé au lieu de répondre aux messages Windows. Dans la surcharge, appelez la méthode héritée pour que les processus par défaut soient toujours exécutés.

Tableau 10.9 Méthodes protégées de TWidgetControl pour la réponse aux événements système

Méthode	Description
ChangeBounds	Utilisée lorsqu'un *TWidgetControl* est redimensionné. A peu près analogue à WM_SIZE ou WM_MOVE dans Windows. Qt définit la "géométrie" d'un widget basé sur la zone client, la VCL utilise toute la taille du contrôle, ce qui inclut ce que Qt nomme le cadre.
ChangeScale	Appelée automatiquement lors du redimensionnement des contrôles. Utilisée pour modifier l'échelle d'une fiche et tous ses contrôles pour une résolution d'écran ou une taille de police différente. Comme ChangeScale modifie les propriétés Top, Left, Width et Height, elle modifie la position du contrôle et ses enfants aussi bien que leur taille.
ColorChanged	Appelée lorsque la couleur du contrôle a été modifiée.
CursorChanged	Appelée lorsque le curseur change de forme. Le curseur de la souris adopte cette forme lorsqu'il est au-dessus de ce widget.
EnabledChanged	Appelée lorsqu'une application modifie l'état activé d'une fenêtre ou d'un contrôle.
FontChanged	Appelée lorsque l'ensemble des ressources de polices a été modifié. Elle définit la police pour le widget et informe tous les enfants de la modification. A peu près analogue au message WM_FONTCHANGE.
PaletteChanged	Appelée lorsque la palette système a été modifiée.
ShowHintChanged	Appelée lorsque les conseils d'aide sont affichés ou cachés pour un contrôle.
StyleChanged	Appelée lorsque les styles de l'interface graphique utilisateur de la fenêtre ou des contrôles ont été modifiés.

Tableau 10.9 Méthodes protégées de TWidgetControl pour la réponse aux événements système (suite)

Méthode	Description
TabStopChanged	Appelée lorsque l'ordre de tabulation de la fiche a été modifié.
VisibleChanged	Appelée lorsqu'un contrôle est caché ou rendu visible.
WidgetDestroyed	Appelée lorsqu'un widget sous-jacent à un contrôle est détruit.

Qt est un kit de développement C++, aussi tous ses widgets sont-ils des objets C++. CLX est écrit en Pascal Objet et le Pascal Objet n'interagit pas directement avec les objets C+. De plus, Qt utilise plusieurs héritages à plusieurs endroits. Delphi inclut donc une couche d'interface qui convertit toutes les classes Qt en une série de fonctions C simples. Celles-ci sont ensuite enveloppées dans un objet partagé sous Linux et dans une DLL sous Windows.

Chaque *TWidgetControl* possède des méthodes virtuelles *CreateWidget*, *InitWidget* et *HookEvents* qui doivent presque toujours être surchargées. *CreateWidget* crée le widget Qt et affecte le Handle à la variable de champ privée FHandle. *InitWidget* est appelée lorsque le widget a été construit et que le Handle est valide.

Certaines affectations de propriétés dans CLX Delphi ont été déplacées du constructeur Create vers *InitWidget*. Cela permet de différer la construction de l'objet Qt jusqu'à ce qu'elle soit vraiment nécessaire. Par exemple, supposons que vous disposiez d'une propriété appelée *Color*. Dans SetColor, vous pouvez vérifier avec HandleAllocated si vous disposez d'un Handle de Qt. Si le Handle est alloué, vous pouvez effectuer l'appel Qt pour définir la couleur. Sinon, vous pouvez stocker la valeur dans une variable de champ privée, et dans *InitWidget*, vous définissez la propriété.

Linux prend en charge deux types d'événements : Widget et System. *HookEvents* est une méthode virtuelle qui connecte les méthodes d'événements des contrôles CLX à un objet de lien particulier qui communique avec l'objet Qt. L'objet de lien n'est rien d'autre qu'un ensemble de pointeurs de méthodes. Les événements système dans Kylix passent par *EventHandler*, qui remplace fondamentalement *WndProc*.

Différences de programmation sous Linux

Le type widechar Linux wchar_t comporte 32 bits par caractère. La norme Unicode 16 bits prise en charge par les widechars du Pascal objet est un sous-ensemble de la norme UCS 32 bits prise en charge par Linux et les bibliothèques GNU. Les données Pascal de type widechar doivent être étendues à 32 bits par caractère avant de pouvoir être transmises à une fonction de système d'exploitation en tant que wchar_t.

Sous Linux, les widestrings sont comptées par référence comme les chaînes longues (alors que ce n'est pas le cas sous Windows).

La gestion multi-octet est différente sous Linux. Sous Windows, les caractères multi-octets (MBCS) sont représentés sous la forme de codes de caractères de 1 et 2 octets. Sous Linux, ils sont représentés par 1 à 6 octets.

Les AnsiStrings peut transporter des séquences de caractères multi-octets, en fonction des paramètres régionaux de l'utilisateur. Le codage Linux pour les caractères multi-octets comme le japonais, le chinois, l'hébreu et l'arabe peuvent être incompatibles avec le codage Windows pour ces mêmes paramètres régionaux. Unicode est portable, alors que le multi-octet ne l'est pas.

Sous Linux, vous ne pouvez pas utiliser de variables sur des adresses absolues. La syntaxe `var X: Integer absolute $1234;` n'est pas prise en charge dans PIC et elle est interdite dans Delphi.

Applications de bases de données multiplates-formes

Sous Windows, Delphi fournit plusieurs méthodes pour accéder aux informations de bases de données. Cela inclut l'utilisation d'ADO, du moteur de bases de données Borland (BDE) et d'InterBase Express. Toutefois, ces trois méthodes ne sont pas disponibles dans Kylix. Vous pouvez utiliser à la place *dbExpress*, une nouvelle technologie d'accès aux données multiplates-formes, qui est également disponible sous Windows, en commençant par la version 6 de Delphi.

Avant de porter une application de base de données vers *dbExpress* pour qu'elle s'exécute sous Linux, vous devez comprendre les différences entre l'utilisation de *dbExpress* et le mécanisme d'accès aux données que vous utilisiez. Ces différences se situent à différents niveaux.

- Au niveau le plus bas, il existe une couche qui communique entre votre application et le serveur de base de données. Cela peut être ADO, le moteur BDE, ou le logiciel client InterBase. Cette couche est remplacée par *dbExpress*, qui est un ensemble de pilotes légers pour le traitement SQL dynamique.
- L'accès aux données de bas niveau est enveloppé dans un ensemble de composants que vous ajoutez à des modules ou des fiches de données. Ces composants incluent des composants de connexion à la base de données, qui représentent la connexion à un serveur de base de données, et des ensembles de données, qui représentent les données obtenues à partir du serveur. Bien qu'il existe des différences très importantes, en raison de la nature unidirectionnelle des curseurs *dbExpress*, elles sont moins prononcées à ce niveau, car les ensembles de données partagent tous un ancêtre commun, comme les composants de connexion de base de données.
- Au niveau de l'interface utilisateur, il y a moins de différences. Les contrôles CLX orientés données sont conçus pour être autant que possible similaires aux contrôles Windows correspondants. Les différences les plus importantes au niveau de l'interface utilisateur résultent des modifications nécessaires à l'utilisation des mises à jour en mémoire cache.

Pour des informations sur le portage d'applications de bases de données existantes vers *dbExpress*, reportez-vous à "Portage d'applications de bases de données vers Linux" à la page 10-29. Pour des informations sur la conception de nouvelles applications *dbExpress*, consultez le chapitre 14, "Conception d'applications de bases de données".

Différences de dbExpress

Sous Linux, *dbExpress* gère la communication avec les serveurs de bases de données. *dbExpress* se compose d'un ensemble de pilotes légers qui mettent en œuvre un ensemble d'interfaces communes. Chaque pilote est un objet partagé (fichier .so) qui doit être lié à votre application. Comme *dbExpress* est conçu pour être multiplates-formes, il sera également disponible sous Windows sous la forme d'un ensemble de bibliothèques de liens dynamiques (.dll).

Comme avec n'importe quelle couche d'accès aux données, *dbExpress* a besoin du logiciel client du fournisseur de base de données. De plus, il utilise un pilote propre à la base de données, plus deux fichiers de configuration, dbxconnections et dbxdrivers. C'est beaucoup moins que, par exemple, pour le moteur BDE, qui nécessite la bibliothèque principale du moteur de bases de données Borland (Idapi32.dll) plus un pilote propre à la base de données et un certain nombre d'autres bibliothèques de gestion.

Voici quelques autres différences entre *dbExpress* et les autres couches d'accès aux données à partir desquelles vous devez porter votre application :

- *dbExpress* fournit un chemin plus simple et plus rapide aux bases de données distantes. Par conséquent, vous pouvez vous attendre à une amélioration sensible des performances pour un accès aux données simple et direct.
- *dbExpress* peut traiter les requêtes et les procédures stockées, mais il ne prend pas en charge l'ouverture des tables.
- *dbExpress* renvoie uniquement des curseurs unidirectionnels.
- *dbExpress* ne dispose pas d'autre possibilité de mise à jour intégrée que l'exécution d'une requête INSERT, DELETE ou UPDATE.
- *dbExpress* n'a pas de mémoire cache pour les métadonnées et l'interface d'accès aux métadonnées lors de la conception est mise en œuvre à l'aide de l'interface centrale d'accès aux données.
- *dbExpress* exécute uniquement des requêtes transmises par l'utilisateur, optimisant ainsi l'accès à la base de données sans introduire de requêtes supplémentaires.
- *dbExpress* gère un tampon d'enregistrement ou un bloc de tampons d'enregistrement de manière interne. Il diffère en cela du moteur BDE, avec lequel les clients doivent allouer la mémoire utilisée pour les enregistrements du tampon.
- *dbExpress* ne prend pas en charge les tables locales qui ne sont pas basées sur SQL (comme Paradox, dBase ou FoxPro).
- *Il existe des pilotes dbExpress* pour InterBase, Oracle, DB2 et MySQL. Si vous utilisez un serveur de base de données différent, vous devrez porter vos données sur l'une de ces bases de données, écrire un pilote *dbExpress* pour le serveur que vous utilisez ou obtenir un pilote *dbExpress* tierce partie pour votre serveur de base de données.

Différences au niveau composant

Lorsque vous écrivez une application *dbExpress*, celle-ci a besoin d'un ensemble de composants d'accès aux données différent de ceux utilisés dans vos applications de bases de données existantes. Les composants *dbExpress* partagent les mêmes classes de base que d'autres composants d'accès aux données (*TDataSet* et *TCustomConnection*), ce qui signifie qu'un grand nombre de propriétés, de méthodes et d'événements sont les mêmes que pour les composants utilisés dans vos applications existantes.

Le tableau suivant présente la liste des composants de bases de données importants utilisés dans InterBase Express, le BDE et ADO dans l'environnement Windows, et montre les composants *dbExpress* comparables pour une utilisation sous Linux et dans des applications multiplates-formes.

Tableau 10.10 Composants d'accès aux données comparables

Composants vInterBase Express	Composants BDE	Composants ADO	Composants dbExpress
TIBDatabase	*TDatabase*	*TADOConnection*	**TSQLConnection**
TIBTable	*TTable*	*TADOTable*	**TSQLTable**
TIBQuery	*TQuery*	*TADOQuery*	**TSQLQuery**
TIBStoredProc	*TStoredProc*	*TADOStoredProc*	**TSQLStoredProc**
TIBDataSet		*TADODataSet*	**TSQLDataSet**

Les ensembles de données *dbExpress (TSQLTable, TSQLQuery, TSQLStoredProc* et *TSQLDataSet)* sont toutefois plus limités que leurs équivalents, car ils ne prennent pas en charge les modifications et permettent uniquement la navigation vers l'avant. Pour plus de détails sur les différences entre les ensembles de données *dbExpress* et les autres ensembles de données disponibles sous Windows, consultez le chapitre 22, "Utilisation d'ensembles de données unidirectionnels".

A cause de l'absence de prise en charge des modifications et de la navigation, la plupart des applications *dbExpress* ne fonctionnent pas directement avec les ensembles de données *dbExpress*. Elles connectent plutôt l'ensemble de données *dbExpress* à un ensemble de données client, qui conserve les enregistrements dans une mémoire tampon et assure une prise en charge des modifications et de la navigation. Pour plus d'informations sur cette architecture, consultez "Architecture des bases de données" à la page 14-6.

Remarque Pour les applications très simples, vous pouvez utiliser *TSQLClientDataSet* à la place d'un ensemble de données *dbExpress* connecté à un ensemble de données client. Cela a pour avantage la simplicité, car il existe une correspondance 1 pour 1 entre l'ensemble de données de l'application que vous portez et l'ensemble de données de l'application portée, mais cette solution est moins souple qu'une connexion explicite entre un ensemble de données *dbExpress* et un ensemble de données client. Pour la plupart des applications, il est recommandé d'utiliser un ensemble de données *dbExpress* connecté à un composant *TClientDataSet*.

Différences au niveau de l'interface utilisateur

Les contrôles CLX orientés données sont conçus pour être aussi similaires que possible aux contrôles Windows correspondants. Par conséquent, le portage de la partie interface utilisateur de vos applications de bases de données introduit quelques considérations supplémentaires par rapport à celles du portage d'une application Windows quelconque vers CLX.

Les principales différences au niveau de l'interface utilisateur proviennent des différences dans la façon dont les ensembles de données *dbExpress* ou les ensembles de données client fournissent les données.

Si vous utilisez uniquement des ensembles de données *dbExpress*, vous devez ajuster votre interface utilisateur à cause du fait que les ensembles de données ne prennent pas en charge la modification mais uniquement la navigation vers l'avant. Il peut, par exemple, être nécessaire de supprimer des contrôles permettant aux utilisateurs de se positionner sur un enregistrement précédent. Comme les ensembles de données *dbExpress* ne gèrent pas de mémoire tampon des données, vous ne pouvez pas afficher les données dans une grille orientée données : vous pouvez seulement afficher un enregistrement à la fois.

Si vous avez connecté l'ensemble de données *dbExpress* à un ensemble de données client, les éléments de l'interface utilisateur associés à la modification et à la navigation devraient toujours fonctionner. Vous devez uniquement les reconnecter à l'ensemble de données client. Dans ce cas, l'attention doit principalement être mise sur la gestion de l'écriture des mises à jour dans la base de données. Par défaut, la plupart des ensembles de données sous Windows écrivent automatiquement les mises à jour sur le serveur de base de données lorsqu'elles sont transmises (par exemple, lorsque l'utilisateur se déplace vers un nouvel enregistrement). Les ensembles de données client, par contre, conservent toujours les mises à jour en mémoire cache. Pour des informations sur la manière de gérer ces différences, consultez "Mise à jour des données dans les applications dbExpress" à la page 10-32.

Portage d'applications de bases de données vers Linux

Le portage de votre application de bases de données vers *dbExpress* vous permet de créer une application multiplates-formes qui s'exécutera à la fois sous Windows et sous Linux. La procédure de portage implique d'apporter des modifications à votre application, car la technologie est différente. La difficulté du portage dépend du type de l'application, de sa complexité et de ce qu'il est nécessaire d'accomplir. Une application qui utilise largement les technologies propres à Windows comme ADO sera plus difficile à porter qu'une autre utilisant la technologie de base de données Delphi.

Suivez ces étapes générales pour porter votre application de base de données Windows/VCL vers Kylix/CLX :

1. Examinez l'endroit où les données de la base de données sont stockées. *dbExpress* fournit des pilotes pour Oracle, Interbase, DB2 et MySQL. Les données doivent résider sur l'un de ces serveurs SQL.

   Certaines versions de Delphi incluent l'utilitaire Data Pump dont vous pouvez vous servir pour transférer les données de la base de données locale, depuis les plates-formes comme Paradox, dBase et FoxPro vers l'une des plates-formes prises en charge. (Consultez le fichier datapump.hlp dans Program Files\Common Files\Borland\Shared\BDE pour des informations sur l'emploi de cet utilitaire.)

2. Si vous n'avez pas séparé vos fiches d'interface utilisateur des modules de données contenant les ensembles de données et les composants de connexion, vous devrez le faire avant de démarrer le portage. Vous isolerez ainsi les parties de votre application qui nécessitent un ensemble de composants totalement nouveau dans les modules de données. Les fiches représentant l'interface utilisateur pourront alors être portées comme n'importe quelle autre application. Pour les détails, reportez-vous à "Portage de votre application" à la page 10-4.

   Les étapes suivantes supposent que vos ensembles de données et composants de connexion sont isolés dans leurs propres modules de données.

3. Créez un nouveau module de données qui contiendra les versions CLX de vos ensembles de données et composants de connexion.

4. Pour chaque ensemble de données de l'application d'origine, ajoutez un ensemble de données *dbExpress*, un composant *TDataSetProvider* et un composant *TClientDataSet*. Utilisez les correspondances du tableau 10.10 pour décider de l'ensemble de données *dbExpress* à utiliser. Donnez à ces composants des noms significatifs.

   - Initialisez la propriété *ProviderName* du composant *TClientDataSet* avec le nom du composant *TDataSetProvider*.
   - Initialisez la propriété *DataSet* du composant *TDataSetProvider* avec l'ensemble de données *dbExpress*.
   - Modifiez la propriété *DataSet* de tous les composants sources de données qui faisaient référence à l'ensemble de données d'origine afin qu'ils fassent à présent référence à l'ensemble de données client.

5. Définissez les propriétés du nouvel ensemble de données pour qu'elles correspondent à celles de l'ensemble de données d'origine :

   - Si l'ensemble de données d'origine était un composant *TTable*, *TADOTable* ou *TIBTable*, initialisez la propriété *TableName* du nouveau composant *TSQLTable* avec la propriété *TableName de l'ensemble de données d'origine*. Copiez également toutes les propriétés utilisées pour définir les liens maître-détail ou spécifier des index. Les propriétés spécifiant des plages et des filtres devraient être initialisées sur l'ensemble de données client plutôt que sur le nouveau composant *TSQLTable*.

- Si l'ensemble de données d'origine est un composant *TQuery*, *TADOQuery* ou *TIBQuery*, initialisez la propriété *SQL* du nouveau composant *TSQLQuery* avec la propriété *SQL* de l'ensemble de données d'origine. Initialisez la propriété *Params* du nouveau *TSQLQuery* conformément à la valeur de la propriété *Params* ou *Parameters de l'ensemble de données d'origine*. Si vous avez initialisé la propriété *DataSource* pour établir un lien maître-détail, copiez-la également.
- Si l'ensemble de données d'origine était un composant *TStoredProc*, *TADOStoredProc* ou *TIBStoredProc*, initialisez la propriété *StoredProcName* du nouveau composant *TSQLStoredProc* avec la propriété *StoredProcName* ou *ProcedureName* de l'ensemble de données d'origine. Initialisez la propriété *Params* du nouveau composant *TSQLStoredProc* conformément à la valeur de la propriété *Params* ou *Parameters* de l'ensemble de données d'origine.

6 Pour tout composant de connexion de base de données dans l'application d'origine (*TDatabase*, *TIBDatabase* ou *TADOConnection*), ajoutez un composant *TSQLConnection* au nouveau module de données. Vous devez également ajouter un composant *TSQLConnection* pour chaque serveur de base de données auquel vous vous connectiez sans composant de connexion (par exemple, en utilisant la propriété *ConnectionString* sur un ensemble de données ADO ou en initialisant la propriété *DatabaseName* de l'ensemble de données BDE avec un alias BDE).

7 Pour chaque ensemble de données *dbExpress* positionné à l'étape 4, initialisez sa propriété *SQLConnection* avec le composant *TSQLConnection* correspondant à la connexion de base de données appropriée.

8 Sur chaque composant *TSQLConnection*, spécifiez les informations nécessaires à l'établissement d'une connexion de base de données. Pour cela, double-cliquez sur le composant *TSQLConnection* pour afficher l'éditeur de connexion et affectez les valeurs correspondant aux paramètres appropriés. Si vous avez dû transférer des données vers un nouveau serveur de base de données à l'étape 1, spécifiez les paramètres appropriés pour ce nouveau serveur. Si vous utilisez le même serveur qu'auparavant, vous pouvez obtenir certaines de ces informations sur le composant de connexion d'origine :

- Si l'application d'origine utilisait *TDatabase*, vous devez transférer les informations qui apparaissent dans les propriétés *Params* et *TransIsolation*.
- Si l'application d'origine utilisait *TADOConnection*, vous devez transférer les informations qui apparaissent dans les propriétés *ConnectionString* et *IsolationLevel*.
- Si l'application d'origine utilisait *TIBDatabase*, vous devez transférer les informations qui apparaissent dans les propriétés *DatabaseName* et *Params*.
- S'il n'y avait pas de composant de connexion d'origine, vous devez transférer les informations associées à l'alias BDE ou qui apparaissaient dans la propriété *ConnectionString* de l'ensemble de données.

Vous pouvez enregistrer cet ensemble de paramètres sous un nouveau nom de connexion. Pour plus de détails sur cette procédure, reportez-vous à "Contrôles des connexions" à la page 17-3.

Mise à jour des données dans les applications dbExpress

Les applications *dbExpress* utilisent des ensembles de données client pour prendre en charge la modification. Lorsque vous transmettez des modifications à un ensemble de données client, celles-ci sont écrites dans le cliché en mémoire des données de l'ensemble de données client, mais elles ne sont pas automatiquement écrites sur le serveur de base de données. Si votre application d'origine utilisait un ensemble de données client pour les mises à jour en mémoire cache, vous n'avez rien à modifier pour prendre en charge la modification sous Linux. Par contre, si vous vous basiez sur le comportement par défaut de la plupart des ensembles de données sous Windows, qui consiste à écrire les modifications sur le serveur de base de données lors de la transmission des enregistrements, vous devrez apporter des changements pour prendre en charge l'utilisation d'un ensemble de données client.

Deux méthodes sont possibles pour convertir une application qui ne conservait pas auparavant les mises à jour en mémoire cache :

- Vous pouvez reproduire le comportement de l'ensemble de données sous Windows en écrivant du code pour appliquer chaque enregistrement mis à jour au serveur de base de données dès qu'il est transmis. Pour cela, fournissez à l'ensemble de données client un gestionnaire d'événement *AfterPost* qui appliquera la mise à jour au serveur de base de données :

```
procedure TForm1.ClientDataSet1AfterPost(DataSet: TDataSet);
begin
  with DataSet as TClientDataSet do
    ApplyUpdates(1);
end;
```

- Vous pouvez ajuster votre interface utilisateur pour traiter les mises à jour en mémoire cache. Cette approche possède certains avantages, comme la réduction du trafic réseau et de la durée des transactions. Toutefois, si vous adoptez l'enregistrement des mises à jour en mémoire cache, vous devrez décider du moment où ces mises à jour devront être ré-appliquées au serveur de base de données, et apporter probablement des modifications à l'interface utilisateur pour laisser les utilisateurs spécifier l'application des mises à jour ou leur indiquer si leurs modifications ont été écrites dans la base de données. En outre, comme les erreurs de mise à jour ne sont pas détectées quand l'utilisateur transmet un enregistrement, vous devrez changer la manière dont vous signalez de telles erreurs à l'utilisateur, afin qu'il puisse voir quelle mise à jour a causé un problème, ainsi que le type de problème.

Si votre application d'origine utilisait la prise en charge fournie par le BDE ou ADO pour conserver les mises à jour en mémoire cache, vous aurez besoin d'apporter des modifications à votre code pour passer à l'utilisation d'un ensemble de données client. Le tableau ci-dessous présente les propriétés, les événements et les méthodes qui prennent en charge les mises à jour en mémoire cache sur les ensembles de données BDE et ADO, ainsi que les propriétés, méthodes et événements correspondants sur *TClientDataSet*.

Tableau 10.11 Propriétés, méthodes et événements pour les mises à jour en mémoire cache

Sur les ensembles de données BDE (ou TDatabase)	Sur les ensembles de données ADO	Sur TClientDataSet	Objectif
CachedUpdates	*LockType*	Non obligatoires, les ensembles de données client conservent toujours les mises à jour en mémoire cache.	Détermine si les mises à jour en mémoire cache sont effectives.
Non pris en charge.	*CursorType*	Non pris en charge	Indique à quel point l'ensemble de données est isolé des modifications sur le serveur.
UpdatesPending	Non pris en charge	*ChangeCount*	Indique si la mémoire cache locale contient des enregistrements mis à jour qui doivent être transmis à la base de données.
UpdateRecordTypes	*FilterGroup*	*StatusFilter*	Indique le type d'enregistrements mis à jour à rendre visibles lors de la transmission de mises à jour en mémoire cache.
UpdateStatus	*RecordStatus*	*UpdateStatus*	Indique si un enregistrement est inchangé, modifié, inséré ou supprimé.
OnUpdateError	Non pris en charge	*OnReconcileError*	Evénement pour traiter les erreurs de mise à jour enregistrement par enregistrement.
ApplyUpdates (sur un ensemble de données ou une base de données)	UpdateBatch	*ApplyUpdates*	Transmet les enregistrement de la mémoire cache locale à la base de données.
CancelUpdates	CancelUpdates ou CancelBatch	*CancelUpdates*	Efface les mises à jour en cours dans la mémoire cache sans les transmettre.
CommitUpdates	Gérés automatiquement	*Reconcile*	Réinitialise la mémoire cache de mise à jour après la transmission réussie des mises à jour.
FetchAll	Non pris en charge	*GetNextPacket* (et *PacketRecords*)	Copie des enregistrements de bases de données dans la mémoire cache locale à des fins de modification et de mise à jour.
RevertRecord	CancelBatch	*RevertRecord*	Annule les mises à jour de l'enregistrement en cours si elles ne sont pas encore effectuées.

Applications Internet multiplates-formes

Une application Internet est une application client/serveur qui utilise des protocoles Internet standard pour connecter le client au serveur. Comme vos applications utilisent des protocoles Internet standard pour les communications client/serveur, vous pouvez rendre ces applications multiplates-formes. Par exemple, un programme côté serveur pour une application Internet communique avec le client par l'intermédiaire du logiciel serveur Web de la machine. L'application serveur est généralement écrite pour Linux ou Windows, mais elle peut aussi être multiplates-formes. Les clients peuvent se trouver sur n'importe quelle plate-forme.

Delphi ou Kylix vous permet de créer des applications de serveur Web sous la forme d'applications CGI ou Apache pour un déploiement sous Linux. Sous Windows, vous pouvez créer d'autres types de serveurs Web comme des DLL Microsoft Server (ISAPI), des DLL Netscape Server (NSAPI) et des applications CGI Windows. Les applications strictement CGI et quelques applications utilisant WebBroker seront les seules à s'exécuter à la fois sous Windows et sous Linux.

Portage d'applications Internet vers Linux

Si vous disposez d'applications Internet existantes que vous souhaitez rendre indépendantes des plates-formes, vous devrez choisir de porter votre application de serveur Web ou de créer une application nouvelle sous Linux. Consultez le chapitre 27, "Création d'applications Internet" pour des informations sur l'écriture de serveurs Web. Si votre application utilise WebBroker, écrit dans l'interface WebBroker et n'utilise pas d'appels d'API natifs, elle ne sera pas aussi difficile à porter vers Linux.

Si votre application écrit dans ISAPI, NSAPI, CGI Windows ou d'autres API Web, elle sera plus difficile à porter. Vous devrez effectuer des recherches dans vos fichiers source et convertir ces appels d'API en appels Apache (consultez httpd.pas dans le répertoire Internet pour des prototypes de fonctions pour les API Apache) ou CGI. Vous devrez également apporter les autres modifications suggérées dans "Portage d'applications VCL vers CLX" à la page 10-3.

Chapitre

11

Utilisation des paquets et des composants

Un paquet est une bibliothèque liée dynamiquement spéciale, utilisée par les applications Delphi, l'EDI ou les deux. Les *paquets d'exécution* fournissent des fonctionnalités lorsqu'un utilisateur exécute une application. Les *paquets de conception* sont utilisés pour installer des composants dans l'EDI et pour créer des éditeurs de propriétés particuliers pour des composants personnalisés. Un même paquet peut fonctionner à la fois en conception et en exécution, les paquets de conception faisant souvent appel à des paquets d'exécution. Pour les distinguer des autres DLL, les paquets sont stockés dans des fichiers dont l'extension est .bpl (Borland Package Library).

Comme les autres bibliothèques d'exécution, les paquets contiennent du code pouvant être partagé par plusieurs applications. Par exemple, les composants Delphi les plus couramment utilisés se trouvent dans un paquet appelé vcl. Chaque fois que vous créez une application, vous pouvez spécifier qu'elle utilise vcl. Lorsque vous compilez une application créée de cette manière, l'image exécutable de l'application ne contient que son propre code et ses propres données, le code commun étant dans le paquet d'exécution appelé vcl60.bpl. Un ordinateur sur lequel sont installées plusieurs applications utilisant des paquets n'a besoin que d'une seule copie de vcl60.bpl, qui est partagé par toutes les applications et par l'EDI de Delphi même.

Delphi est livré avec plusieurs paquets d'exécution précompilés, qui encapsulent les composants VCL et CLX. Delphi utilise également des paquets de conception pour faciliter la manipulation des composants dans l'EDI.

Vous pouvez construire des applications avec ou sans paquets. Mais, si vous voulez ajouter à l'EDI des composants personnalisés, vous devez les installer en tant que paquets de conception.

Vous pouvez créer vos propres paquets d'exécution afin de les partager entre plusieurs applications. Si vous écrivez des composants Delphi, compilez-les dans des paquets de conception avant de les installer.

Pourquoi utiliser des paquets ?

Les paquets de conception simplifient la distribution et l'installation de composants personnalisés. L'utilisation, optionnelle, des paquets d'exécution offre plusieurs avantages par rapport à la programmation conventionnelle. En compilant dans une bibliothèque d'exécution du code réutilisé, vous pouvez le partager entre plusieurs applications. Par exemple, toutes vos applications, y compris Delphi, peuvent accéder aux composants standard par le biais des paquets. Comme les applications n'intègrent pas de copies séparées de la bibliothèque des composants dans leur exécutable, ces dernières sont plus petites, ce qui économise à la fois de l'espace disque et des ressources système. De plus, les paquets permettent d'accélérer la compilation car seul le code spécifique à l'application est compilé à chaque génération.

Les paquets et les DLL standard

Créez un paquet quand vous voulez qu'un composant personnalisé soit utilisable dans l'EDI. Créez une DLL standard quand vous voulez générer une bibliothèque utilisable par toute application, quel que soit l'outil de développement utilisé pour la créer.

Le tableau suivant donne la liste des types de fichiers associés aux paquets :

Tableau 11.1 Fichiers paquet compilés

Extension du fichier	Contenu
.dpk	Le fichier source donnant la liste des unités contenues dans le paquet.
dcp	Une image binaire contenant une en-tête de paquet et la concaténation de tous les fichiers dcu du paquet, y compris les informations de symbole nécessaires au compilateur. Un seul fichier dcp est créé pour chaque paquet. Le nom de base pour le dcp est celui du fichier source dpk. Vous devez avoir un fichier .dcp pour construire une application avec des paquets.
dcu	Une image binaire pour un fichier unité contenu dans un paquet. Lorsque cela est nécessaire, un seul fichier dcu est créé pour chaque fichier unité.
bpl	Le paquet d'exécution. Ce fichier est une DLL avec des caractéristiques spécifiques à Delphi. Le nom de base du bpl est celui du fichier source dpk.

Vous pouvez inclure VCL ou CLX, ou les deux types de composants dans un paquet. Les paquets destinés à être multiplates-formes ne doivent inclure que des composants CLX.

Remarque Les paquets partagent leurs données globales avec les autres modules d'une application.

Pour plus d'informations sur les paquets et les DLL, voir le *Guide du langage Pascal Objet*.

Paquets d'exécution

Les paquets d'exécution sont déployés avec les applications Delphi. Ils fournissent des fonctionnalités lorsqu'un utilisateur exécute une application.

Pour exécuter une application utilisant des paquets, le fichier exécutable de l'application et tous les fichiers paquet (fichiers .bpl) qu'elle utilise doivent se trouver sur l'ordinateur. Les fichiers .bpl doivent être dans le chemin du système pour qu'une application puisse les utiliser. Quand vous déployez une application, vérifiez que les utilisateurs possèdent la version correcte de chaque bpl nécessaire.

Utilisation des paquets dans une application

Pour utiliser des paquets dans une application :

1. Chargez ou créez un projet dans l'EDI.
2. Choisissez Projet | Options.
3. Choisissez la page Paquets.
4. Cochez la case "Construire avec les paquets d'exécution" et saisissez un ou plusieurs noms de paquets dans la boîte de saisie placée en dessous. Les paquets d'exécution associés avec les paquets de conception déjà installés apparaissent déjà dans la boîte de saisie.
5. Pour ajouter un paquet à une liste existante, cliquez sur le bouton Ajouter puis entrez le nom du nouveau paquet dans la boîte de dialogue Ajout de paquet d'exécution. Pour parcourir la liste des paquets disponibles, cliquez sur le bouton Ajouter puis sur le bouton Parcourir placé à côté de la boîte de saisie Nom de paquet dans la boîte de dialogue Ajout de paquet d'exécution.

   Si vous modifiez la boîte de saisie Chemin de recherche dans la boîte de dialogue Ajout de paquet d'exécution, le chemin d'accès global à la bibliothèque Delphi est modifié.

   Il n'est pas nécessaire d'inclure l'extension dans le nom de paquet (ni le nombre représentant la version de Delphi) ; autrement dit, vcl60.bpl s'écrit vcl. Si vous tapez les noms directement dans la boîte de saisie Paquets d'exécution, séparez-les par des points-virgules. Par exemple :

```
rtl;vcl;vcldb;vclado;vclx;Vclbde;
```

Les paquets énumérés dans la boîte de saisie Paquets d'exécution sont automatiquement liés à l'application lors de sa compilation. Les paquets en double sont ignorés et si la boîte de saisie est vide, l'application est compilée sans paquet.

Les paquets d'exécution sont sélectionnés uniquement pour le projet en cours. Pour que les choix en cours deviennent des choix par défaut pour les futurs projets, cochez la case Défaut, en bas de la boîte de dialogue.

Remarque Lorsque vous créez une application avec des paquets, vous devez inclure les noms originels des unités Delphi dans la clause **uses** des fichiers source. Par exemple, le fichier source de la fiche principale pourrait commencer ainsi :

```
unit MainForm;

interface
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;
```

Les unités référencées dans cet exemple sont contenues dans les paquets vcl et rtl. Néamoins, vous devez conserver ces références dans la clause **uses**, même si vous utilisez vcl et rtl dans votre application, ou vous aurez des erreurs de compilation. Dans les fichiers source générés, Delphi ajoute automatiquement ces unités à la clause **uses**.

Paquets chargés dynamiquement

Pour charger un paquet à l'exécution, appelez la fonction *LoadPackage*. *LoadPackage* charge le paquet, recherche les unités dupliquées, appelle les blocs d'initialisation de toutes les unités contenues dans le paquet. Par exemple, le code suivant est exécuté lorsqu'un fichier est choisi dans la boîte de dialogue de sélection des fichiers.

```
with OpenDialog1 do
  if Execute then
    with PackageList.Items do
      AddObject(FileName, Pointer(LoadPackage(FileName)));
```

Pour décharger un paquet dynamiquement, appelez *UnloadPackage.* Soyez prudent en détruisant toute instance de classe définie dans le paquet et en dérecensant les classes précédemment recensées.

Choix des paquets d'exécution à utiliser

Delphi est livré avec plusieurs paquets d'exécution précompilés, dont rtl et vcl, qui fournissent le langage de base et le support des composants.

Le paquet vcl contient les composants les plus couramment utilisés, le paquet rtl contient toutes les fonctions système qui ne sont pas des composants et les éléments de l'interface Windows. Il ne comprend pas les composants base de données ni les autres composants spéciaux, qui se trouvent dans des paquets distincts.

Pour créer une application de base de données client/serveur utilisant des paquets, vous avez besoin d'au moins deux paquets d'exécution : vcl et vcldb. Si vous voulez également utiliser dans votre application des composants arborescence (*Outline*), vous avez besoin en plus de vclx. Pour utiliser ces

paquets, choisissez Projet | Options, sélectionnez la page Paquets et entrez la liste suivante dans la boîte de saisie Paquets d'exécution.

```
rtl;vcl;Vcldb;vclx;
```

Il n'est, en fait, pas nécessaire de préciser vcl et rtl, car elles sont référencées dans la clause Requires de vcldb (voir "Clause Requires" à la page 11-10). Votre application se compilera de la même façon que vcl et rtl figurent ou non dans la liste des paquets d'exécution.

Paquets personnalisés

Un paquet personnalisé est soit une bpl que vous programmez et compilez vous-même, soit un paquet précompilé développé par un fournisseur tiers. Pour utiliser dans une application un paquet d'exécution personnalisé, choisissez Projet | Options et ajoutez le nom du paquet à la boîte de saisie Paquets d'exécution de la page Paquets. Par exemple, si vous avez créé un paquet effectuant des statistiques, nommé stats.bpl, la boîte de saisie Paquets d'exécution doit avoir la forme :

```
rtl;vcl;vcldb;stats
```

Si vous créez vos propres paquets, vous pouvez les ajouter selon vos besoins à la liste.

Paquets de conception

Des paquets de conception sont utilisés pour installer des composants dans la palette des composants de l'IDE ou pour créer les éditeurs de propriétés spéciaux de composants personnalisés.

Delphi est livré avec de nombreux paquets de conception déjà installés dans l'EDI. Ils dépendent de la version de Delphi que vous utilisez et de la personnalisation à laquelle vous avez pu la soumettre. Vous pouvez voir la liste des paquets installés sur votre système en choisissant la commande Composant | Installer des paquets.

Les paquets de conception fonctionnent en appelant des paquets d'exécution référencés dans leur clause Requires. Voir "Clause Requires" à la page 11-10. Par exemple, dclstd référence vcl. Dclstd contient lui-même des fonctionnalités supplémentaires qui rendent la plupart des composants standard disponibles dans la palette des composants.

Outre les paquets pré-installés, vous pouvez installer dans l'EDI vos propres paquets de composants ou ceux développés par des tiers. Le paquet de conception dclusr est fourni en guise de conteneur par défaut des nouveaux composants.

Installation de paquets de composants

Tous les composants installés dans l'EDI le sont sous la forme de paquets. Si vous écrivez vos propres composants, créez et compilez un paquet les contenant, voir "Création et modification de paquets" à la page 11-7. Le code source des composants doit respecter le modèle décrit dans la partie V, "Création de composants personnalisés".

Pour installer ou désinstaller vos propres composants ou les composants fournis par un tiers, procédez de la manière suivante :

1. Si vous installez un nouveau paquet, copiez ou déplacez les fichiers paquet dans un répertoire local. Si le paquet est livré avec des fichiers .bpl, .dcp et .dcu,vous devez tous les copier. Pour davantage d'informations sur ces fichiers, voir "Fichiers paquets créés lors d'une compilation réussie" à la page 11-13.

   Le répertoire dans lequel sont stockés le fichier .dcp et les fichiers .dcu, s'ils font partie de la distribution, doit être dans le chemin de la bibliothèque Delphi.

   Si le paquet est distribué sous forme d'un fichier .dpc (collection de paquets), seul cet unique fichier doit être copié ; le fichier .dpc contient les autres fichiers. Pour plus d'informations sur les fichiers de collection de paquets, voir "Fichiers de collection de paquets" à la page 11-14.)

2. Choisissez Composant | Installer des paquets dans le menu de l'EDI ou choisissez Projet | Options et cliquez sur l'onglet Paquets.

3. Une liste des paquets disponibles apparaît sous "Paquets de conception".

   - Pour installer un paquet dans l'EDI, cochez la case à cocher placée à côté du nom de paquet.
   - Pour désinstaller un paquet, désélectionnez la case à cocher.
   - Pour voir une liste des composants inclus dans un paquet installé, sélectionnez le paquet et cliquez sur Composants.
   - Pour ajouter un paquet à la liste, cliquez sur le bouton Ajouter puis recherchez dans la boîte de dialogue d'ouverture de paquet le répertoire dans lequel se trouve le fichier .bpl ou .dpc (voir l'étape 1). Sélectionnez le fichier .bpl ou .dpc et cliquez sur Ouvrir. Si vous sélectionnez un fichier .dpc, une nouvelle boîte de dialogue apparaît pour la gestion de l'extraction du fichier .bpl et d'autres fichiers de la collection de paquets.
   - Pour retirer un paquet de la liste, sélectionnez le paquet, puis cliquez sur Supprimer.

4. Cliquez sur OK.

Les composants du paquet sont installés sur la page de la palette des composants spécifiée dans la procédure *RegisterComponents* des composants, avec les noms dont ils étaient affectés dans cette même procédure.

Sauf si vous changez les options par défaut, les nouveaux projets sont créés avec tous les paquets disponibles installés. Si vous voulez que vos choix d'installation deviennent les options par défaut pour les nouveaux projets, cochez Défaut, en bas de la page Paquets de la boîte de dialogue Options du projet.

Pour supprimer des composants de la palette des composants sans désinstaller de paquet, sélectionnez Composant | Configurer la palette, ou bien Outils | Options d'environnement et cliquez sur l'onglet Palette. La page Palette contient la liste de tous les composants installés avec le nom de la page où chacun apparaît. Sélectionnez le composant à supprimer de la palette et cliquez sur Cacher.

Création et modification de paquets

Pour créer un paquet, il faut:

- Un *nom* pour le paquet.
- Une liste des autres paquets *requis* (liés) par le nouveau paquet.
- Une liste des fichiers unité devant être *contenus* par le paquet lors de sa compilation. Un paquet est avant tout un conteneur pour ses unités de code source qui contiennent les fonctionnalités du bpl compilé. C'est dans la clause Contains que vous placez les unités de code source des composants personnalisés devant être compilés dans un paquet

Les fichiers source de paquets, qui se terminent par l'extension .dpk, sont générés par l'éditeur de paquets.

Création d'un paquet

Pour créer un paquet, suivez les étapes suivantes. Pour davantage d'informations sur les étapes suivantes, voir "Présentation de la structure d'un paquet" à la page 11-9.

Pour créer un paquet, suivez les étapes suivantes. Pour davantage d'informations sur ces étapes, voir "Présentation de la structure d'un paquet" à la page 11-9.

Remarque N'utilisez pas de IFDEF dans un fichier de paquet (.dpk) comme dans le développement multiplate-forme. Mais, vous pouvez les utiliser dans le code source.

1. Choisissez Fichier | Nouveau, sélectionnez l'icône Paquet et choisissez OK.
2. Le paquet généré est affiché dans l'éditeur de paquet.
3. L'éditeur de paquet affiche pour le nouveau paquet un nœud *Requires* et un nœud *Contains*.
4. Pour ajouter une unité à la clause **contains**, cliquez sur le turbobouton Ajouter au paquet. Dans la page Ajouter unité, tapez un nom de fichier .pas dans la boîte de saisie Nom de fichier unité, ou cliquez sur Parcourir... pour

rechercher le fichier, puis cliquez sur OK. L'unité sélectionnée apparaît sous le nœud Contains de l'éditeur de paquet. Vous pouvez ajouter des unités supplémentaires en répétant cette étape.

5 Pour ajouter un paquet à la clause **requires**, cliquez sur le turbobouton Ajouter au paquet. Dans la page Nécessite, tapez un nom de fichier .dcp dans la boîte de saisie Nom de paquet, ou cliquez sur Parcourir... pour rechercher le fichier, puis cliquez sur OK. Le paquet sélectionné apparaît sous le nœud Requires dans l'éditeur de paquet. Vous pouvez ajouter des paquets supplémentaires en répétant cette étape.

6 Cliquez sur le turbobouton Options, et sélectionnez le type de paquet à générer.

- Pour créer un paquet de conception uniquement (un paquet ne pouvant s'utiliser à l'exécution), sélectionnez le bouton radio Seulement en conception. (Ou ajoutez la directive de compilation {$DESIGNONLY} au fichier dpk.)
- Pour créer un paquet d'exécution uniquement (un paquet ne pouvant être installé), sélectionnez le bouton radio d'exécution seule. (Ou ajoutez la directive de compilation {$RUNONLY} au fichier dpk.)
- Pour créer un paquet utilisable à l'exécution et à la conception, sélectionnez les deux boutons radio Conception et Exécution.

7 Dans l'éditeur de paquet, cliquez sur le turbobouton Compiler le paquet pour compiler votre paquet.

Modification d'un paquet existant

Il y a plusieurs manières d'ouvrir un paquet existant afin de le modifier :

- Choisissez Fichier | Ouvrir (ou Fichier | Réouvrir) et sélectionnez un fichier dpk.
- Choisissez Composant | Installer des paquets, sélectionnez un paquet dans la liste Paquets de conception et choisissez le bouton Modifier.
- Quand l'éditeur de paquet est ouvert, sélectionnez un des paquets du nœud Requires, cliquez avec le bouton droit de la souris et choisissez Ouvrir.

Pour modifier la description d'un paquet ou définir les options d'utilisation, cliquez sur le turbobouton Options dans l'éditeur de paquet et sélectionnez l'onglet Description.

La boîte de dialogue Options du projet possède une case à cocher Défaut dans le coin inférieur gauche. Si vous cliquez sur OK quand cette case est cochée, les options choisies sont enregistrées comme paramètres par défaut pour les nouveaux projets . Pour restaurer les valeurs par défaut originales, supprimez ou renommez le fichier defproj.dof.

Modification manuelle de fichiers source de paquets

Les fichiers source de paquets, comme les fichiers projet, sont générés par Delphi à partir des informations que vous lui avez fourni. Comme les fichiers projet, ils peuvent aussi être modifiés manuellement. Un fichier source de paquet devrait être enregistré avec l'extension .dpk (pour Delphi PacKage) afin d'éviter toute confusion avec les autres fichiers contenant du code source Pascal Objet.

Pour ouvrir un fichier source de paquet dans l'éditeur de code,

1 Ouvrez les paquets dans l'éditeur de paquets.

2 Cliquez avec le bouton droit de la souris dans l'éditeur de paquets et sélectionnez Voir le source.

- L'en-tête **package** spécifie le nom du paquet.
- La clause **requires** énumère d'autres paquets externes utilisés par le paquet en cours. Si un paquet ne contient pas d'unité qui utilise des unités d'un autre paquet, il n'a pas besoin de clause **requires**.
- La clause **contains** identifie les fichiers unité à compiler et à rassembler dans le paquet. Toutes les unités utilisées par des unités de la clause **contains** qui ne se trouvent pas dans les paquets de la clause **requires** seront aussi rassemblées dans le paquet, même si elle ne sont pas indiquées dans la clause (le compilateur indique un avertissement).

Par exemple, le code suivant déclare le paquet vcldb (dans le fichier source vcldb60.bpl) :

```
package vcldb;
   requires vcldb;
   contains rtl, vcl, Db, DBActns, DBOleCtl, Dbcgrids, dbCommon, dbConsts, Dbctrls,
Dbgrids, Dblogdlg, SQLTimSt, FmtBcd;
end.
```

Présentation de la structure d'un paquet

Les paquets incluent les parties suivantes :

- Nom de paquet
- clause Requires
- clause Contains

Nom de paquets

Les noms de paquets doivent être uniques dans un projet. Si un paquet a le nom STATS, l'éditeur de paquet génère un fichier source correspondant appelé STATS.dpk ; le compilateur génère un exécutable et une image binaire appelés respectivement STATS.bpl et STATS.dcp. Utilisez STATS pour vous référer au paquet dans la clause **requires** d'un autre paquet, ou lors de l'utilisation du paquet dans une application.

Clause Requires

La clause **requires** spécifie les autres paquets, externes, utilisés par le paquet en cours. Un paquet externe inclus dans la clause **requires** est automatiquement lié lors de la compilation dans toute application utilisant le paquet en cours ou l'une des unités contenues dans le paquet externe.

Si les fichiers unité contenus dans votre paquet font référence à d'autres unités empaquetées, les autres paquets doivent apparaître dans la clause **requires** de votre paquet, sinon vous devrez les ajouter. Si les autres paquets sont omis de la clause **requires**, le compilateur les importera dans votre paquet comme 'unités contenues implicitement'.

Remarque La plupart des paquets que vous créez nécessitent rtl. Si vous utilisez des composants VCL, vous devez inclure également le paquet vcl. Si vous utilisez des composants CLX en programmation multiplate-forme, vous devez inclure VisualCLX.

Pas de référence circulaire

Les paquets ne doivent pas contenir de référence circulaire dans leur clause **requires**. Par conséquent :

- Un paquet ne doit pas se référencer lui-même dans sa clause **requires**.
- Une chaîne de références doit se terminer sans référencer un paquet de la chaîne. Si le paquet A requiert le paquet B, alors le paquet B ne doit pas requérir le paquet A ; si le paquet A requiert le paquet B qui requiert le paquet C, alors le paquet C ne doit pas requérir le paquet A.

Gestion des références de paquet dupliquées

Les références en double dans la clause **requires** d'un paquet, ou dans la boîte de saisie Paquet d'exécution, sont ignorées. Mais, pour la lisibilité du programme, il vaut mieux les supprimer.

Clause Contains

La clause **contains** identifie les fichiers unité à lier dans le paquet. Si vous écrivez votre propre paquet, placez votre code source dans des fichiers pas et incluez-les dans la clause **contains**.

Eviter l'utilisation de code source redondant

Un paquet ne peut apparaître dans la clause **contains** d'un autre paquet.

Toutes les unités incluses directement dans la clause **contains** d'un paquet, ou indirectement dans l'une de ces unités sont liées dans le paquet au moment de la compilation.

Une unité ne peut être contenue (directement ou indirectement) dans plusieurs des paquets utilisés par une même application, *y compris l'EDI Delphi*. Cela signifie que si vous créez un paquet contenant l'une des unités de vcl, vous ne pourrez pas installer ce paquet dans l'EDI. Pour utiliser une unité déjà empaquetée dans un autre paquet, placez le premier paquet dans la clause **requires** du second paquet.

Compilation de paquets

Vous pouvez compiler un paquet dans l'EDI ou depuis la ligne de commande. Pour recompiler un paquet directement dans l'EDI :

1. Choisissez Fichier | Ouvrir.
2. Sélectionnez le paquet Delphi (*.dpk) à partir de la liste déroulante Fichiers de type.
3. Sélectionnez un fichier .dpk dans la boîte de dialogue.
4. Lorsque l'éditeur de paquet est ouvert, cliquez sur le turbobouton Compiler.

Vous pouvez insérer des directives de compilation dans le code source du paquet. Pour plus d'informations, voir le paragraphe ci-dessous "Directives de compilation propres aux paquets".

Si vous compilez à partir de la ligne de commande, de nouvelles options d'édition de lien spécifiques aux paquets sont utilisables. Pour plus d'informations, voir "Utilisation du compilateur et du lieur en ligne de commande" à la page 11-13.

Directives de compilation propres aux paquets

Le tableau suivant liste les directives de compilation propres aux paquets qu'il est possible d'insérer dans le code source.

Tableau 11.2 Directives de compilation propres aux paquets

Directive	Fonction
{$IMPLICITBUILD OFF}	Empêche une recompilation implicite du paquet. Utilisez-la dans les fichiers .dpk lors de la compilation de paquets qui fournissent des fonctionnalités de bas niveau et qui ne sont pas modifiées fréquemment ou dont le source n'est pas distribué.
{$G-} ou {IMPORTEDDATA OFF}	Désactive la création de références de données importées. Cette directive augmente l'efficacité des accès mémoire, mais empêche l'unité où elle se trouve de faire référence à des variables d'une autre unité.
{$WEAKPACKAGEUNIT ON}	Les unités sont "faiblement empaquetées". Voir "Paquets faibles" à la page 11-12 ci-dessous.
{$DENYPACKAGEUNIT ON}	Empêche les unités d'être placées dans un paquet.
{$DESIGNONLY ON}	Compile le paquet pour une installation dans l'EDI. (Mettre dans le fichier .dpk.)
{$RUNONLY ON}	Compile le paquet comme exécutable seulement. (Mettre dans le fichier .dpk.)

Remarque Utilisez **{$DENYPACKAGEUNIT ON}** dans votre code pour que l'unité ne soit pas mise dans un paquet. L'utilisation de **{$G-}** ou {**$IMPORTEDDATA OFF}** permet à un paquet de ne pas être utilisé dans la même application avec d'autres paquets. Les paquets compilés avec la directive **{$DESIGNONLY ON}** ne devrait pas être utilisés dans les applications puisque qu'ils contiennent du

code nécessaire à l'EDI. D'autres directives de compilation peuvent être utilisées dans le code source de paquet. Voir Directives de compilation dans l'aide en ligne pour les directives de compilation qui n'ont pas été abordées ici.

Reportez-vous à "Création de paquets et de DLL" à la page 5-10, pour d'autres directives pouvant être utilisées dans toutes les bibliothèques.

Paquets faibles

La directive **$WEAKPACKAGEUNIT** affecte la manière dont un fichier .dcu est stocké dans les fichiers .dcp et .bpl d'un paquet. Pour des informations sur les fichiers générés par le compilateur, voir "Fichiers paquets créés lors d'une compilation réussie" à la page 11-13.) Si **{$WEAKPACKAGEUNIT ON}** apparaît dans un fichier unité, le compilateur omet l'unité des bpl si c'est possible, et crée une version locale non empaquetée de l'unité quand elle est nécessaire à une autre application ou un autre paquet. Une unité compilée avec cette directive est dite "faiblement empaquetée".

Si, par exemple, vous créez un paquet appelé PACK ne contenant que l'unité UNIT1. Supposez que UNIT1 n'utilise aucune autre unité, mais fait des appels à RARE.dll. Si vous placez la directive **{$WEAKPACKAGEUNIT ON}** dans UNIT1.pas, lors de la compilation du paquet, UNIT1 n'est pas incluse dans PACK.bpl ; vous n'avez donc pas à distribuer de copie de RARE.dll avec PACK. Néanmoins, UNIT1 sera toujours incluse dans PACK.dcp. Si UNIT1 est référencée dans un autre paquet ou une autre application utilisant PACK, elle sera copiée dans PACK.dcp et directement compilée dans le projet.

Supposons maintenant que vous ajoutiez à PACK une deuxième unité, UNIT2 qui utilise UNIT1. Cette fois, même si vous compilez PACK avec la directive **{$WEAKPACKAGEUNIT ON}** dans UNIT1.pas, le compilateur inclut UNIT1 dans PACK.bpl. Par contre les autres paquets ou applications faisant référence à UNIT1 utiliseront la copie (non empaquetée) prise dans PACK.dcp.

Remarque Les fichiers unité contenant la directive **{$WEAKPACKAGEUNIT ON}** ne doivent pas contenir de variables globales, de section d'initialisation ou de sections de finalisation.

La directive **$WEAKPACKAGEUNIT** est une caractéristique avancée proposée pour les développeurs distribuant leurs paquets à d'autres programmeurs Delphi. Cela permet d'éviter la distribution de DLL rarement utilisées ou d'éliminer des conflits entre des paquets dépendant d'une même bibliothèque externe.

Ainsi, l'unité PenWin de Delphi référence PenWin.dll. La plupart des projets n'utilisent pas PenWin et la plupart des ordinateurs n'ont pas de fichier PenWin.dll installé. C'est pour cela que l'unité PenWin est faiblement empaquetée dans vcl. Quand vous compilez un projet utilisant PenWin et le paquet vcl, PenWin est copié depuis VCL60.dcp et lié directement à votre projet ; l'exécutable résultant est lié statiquement à PenWin.dll.

Si PenWin n'était pas faiblement empaqueté, il se poserait deux problèmes. Tout d'abord, il faudrait que vcl soit lié de manière statique à PenWin.dll et ne pourrait de ce fait être chargé que sur un système disposant de PenWin.dll

installé. De plus, si vous tentez de créer un paquet contenant PenWin, une erreur de compilation aurait lieu, puisque l'unité PenWin serait contenue dans vcl et dans votre paquet. Ainsi, sans "empaquetage faible", l'unité PenWin n'aurait pas pu être incluse dans la distribution standard de vcl.

Utilisation du compilateur et du lieur en ligne de commande

Quand vous compilez depuis la ligne de commande, utilisez les options spécifiques aux paquets présentées dans le tableau suivant.

Tableau 11.3 Options du compilateur en ligne de commande propres aux paquets

Option	Fonction
-$G-	Désactive la création de références de données importées. L'utilisation de cette option augmente l'efficacité des accès mémoire, mais empêche les paquets compilés avec cette option de référencer des variables appartenant à d'autres paquets.
-LE*chemin*	Spécifie le répertoire où se trouvera le fichier bpl du paquet.
-LN*chemin*	Spécifie le répertoire où se trouvera le fichier dcp du paquet.
-LU*paquet*	Utilise les paquets.
-Z	Empêche la recompilation implicite ultérieure d'un paquet. Utilisez cette option lors de la compilation de paquets qui fournissent des fonctionnalités de bas niveau, qui changent peu souvent entre les builds, ou dont le code source ne sera pas distribué.

Remarque Lutilisation de loption **-$G-** empêche un paquet dêtre utilisé dans une même application avec dautres paquets. Les autres options en ligne de commande peuvent être utilisées de manière appropriée lors de la compilation des paquets. Voir "Le compilateur en ligne de commande" dans l'aide en lignepour les options en ligne de commande qui n'ont pas été abordées ici.

Fichiers paquets créés lors d'une compilation réussie

Pour créer un paquet, compilez un fichier source ayant l'extension .dpk. Le nom de base du fichier .dpk doit correspondre au nom de base des fichiers générés par le compilateur ; c'est-à-dire que si le fichier source du paquet s'appelle traypak.dpk, le compilateur crée un paquet appelé traypak.bpl.

Le tableau suivant donne la liste des fichiers générés par une compilation réussie d'un paquet.

Tableau 11.4 Fichiers compilés d'un paquet

Extension de fichier	Contenu
dcp	Une image binaire contenant un en-tête de paquet et la concaténation de tous les fichiers dcu du paquet. Un seul fichier dcp est créé pour chaque paquet. Le nom pour le fichier dcp est celui du fichier source dpk.
dcu	Une image binaire pour un fichier unité contenu dans un paquet. Lorsque cela est nécessaire, un seul fichier dcu est créé pour chaque fichier unité.
bpl	Le paquet d'exécution. Ce fichier est une DLL avec des caractéristiques spécifiques à Delphi. Le nom de base du bpl est celui du fichier source dpk.

Lors de la compilation, les fichiers bpi, bpl et lib sont générés par défaut dans les répertoires spécifiés dans la page Bibliothèque de la boîte de dialogue des options d'environnement. Vous pouvez redéfinir les options par défaut en cliquant sur le turbobouton Options de l'éditeur de paquets afin d'afficher la boîte de dialogue des options du projet ; faites alors les modifications nécessaires dans la page Répertoires/Conditions.

Déploiement de paquets

Vous déploierez les paquets de la même façon que les applications. Pour plus d'informations sur le déploiement en général, reportez-vous au chapitre 13, "Déploiement des applications".

Déploiement d'applications utilisant des paquets

Pour distribuer une application utilisant des paquets d'exécution, vérifiez que l'utilisateur dispose du fichier .exe de l'application, ainsi que de toutes les bibliothèques (.bpl ou .dll) appelées par l'application. Si les fichiers bibliothèque sont dans un répertoire différent de celui du fichier .exe, ils doivent être accessibles via les chemins d'accès de l'utilisateur. Vous pouvez suivre la convention consistant à placer les fichiers des bibliothèques dans le répertoire Windows\System. Si vous utilisez InstallShield Express, le script d'installation peut vérifier la présence des paquets nécessaires sur le système de l'utilisateur avant de les réinstaller aveuglément.

Distribution de paquets à d'autres développeurs

Si vous distribuez des paquets d'exécution ou de conception à d'autres développeurs Delphi, assurez-vous de fournir les fichiers .dcp et .bpl. Vous aurez probablement besoin d'inclure aussi les fichiers .dcu.

Fichiers de collection de paquets

Les collections de paquet (fichiers .dpc) offrent un moyen pratique de distribuer des paquets à d'autres développeurs. Chaque collection de paquets contient un ou plusieurs paquets, comprenant les bpl et les fichiers supplémentaires que vous voulez distribuer avec. Lorsqu'une collection de paquets est sélectionnée pour une installation de l'EDI, les fichiers qui la constituent sont automatiquement extraits du fichier conteneur .pce ; la boîte de dialogue Installation offre la possibilité d'installer tous les paquets de la collection ou ceux sélectionnés.

Pour créer une collection de paquets,

1 Choisissez Outils | Editeur de collection de paquets pour ouvrir l'éditeur de collection de paquets.

2. Cliquez sur le turbobouton Ajouter un paquet, sélectionnez un bpl dans la boîte de dialogue Sélectionner le paquet et choisissez Ouvrir. Pour ajouter d'autres bpl à la collection, choisissez à nouveau le turbobouton Ajouter un paquet. Sur le côté gauche de l'éditeur de paquets, un diagramme arborescence affiche les bpl que vous avez ajouté. Pour supprimer un paquet, sélectionnez-le et choisissez le turbobouton Retirer le paquet.
3. Sélectionnez le nœud Collection en haut du diagramme arborescence. Sur la partie droite de l'éditeur de collection de paquets, deux champs apparaissent :
   - Dans la boîte de saisie Nom auteur/vendeur, vous pouvez saisir des informations optionnelles à propos de votre collection de paquets qui apparaîtront dans la boîte de dialogue Installation lorsque les utilisateurs installeront les paquets.
   - Dans la liste Nom de répertoire, énumérez les répertoires dans lesquels vous voulez installer les fichiers de la collection de paquets. Utilisez les boutons Ajouter, Edition et Supprimer pour modifier cette liste. Par exemple, supposons que vous voulez copier tous les fichiers de code source dans un même répertoire. Dans ce cas, vous pouvez saisir comme `Source` le nom de répertoire `C:\MyPackage\Source`, comme le chemin suggéré. La boîte de dialogue Installation affichera C:\`C:\MyPackage\Source` comme chemin suggéré pour le répertoire.
4. En plus des bpl, la collection de paquets peut contenir des fichiers .dcp, .dcu, et des unités .pas, de la documentation, et d'autres fichiers que l'on peut inclure pour la distribution. Les fichiers annexes sont placés dans des groupes de fichiers associés aux paquets spécifiques (bpl). Les fichiers d'un groupe ne sont installés que lorsque le bpl associé est installé. Pour mettre les fichiers annexes dans la collection de paquets, sélectionnez un bpl dans le diagramme arborescence et choisissez le turbobouton Ajouter un groupe de fichiers ; saisissez un nom pour le groupe de fichiers. Ajoutez d'autres fichiers si vous le désirez en procédant de la même manière. Lorsque vous sélectionnez un groupe de fichiers, de nouveaux champs apparaissent sur la droite de l'éditeur de collection de paquets.
   - Dans la boîte liste Répertoire d'installation, sélectionnez le répertoire dans lequel vous voulez installer les fichiers de ce groupe. La liste déroulante comprend les répertoires saisis dans liste de répertoires de l'étape 3 ci-dessus.
   - Vérifiez la case à cocher Groupe optionnel si vous voulez que l'installation des fichiers de ce groupe soit facultative.
   - Sous Fichiers inclus, énumérez les fichiers que vous voulez inclure dans ce groupe. Utilisez les boutons Ajouter, Supprimer et Auto pour modifier la liste. Le bouton Auto permet de sélectionner tous les fichiers avec l'extension spécifiée dont la liste se trouve dans la **clause contains** du paquet ; l'éditeur de collection de paquets utilise le chemin de la bibliothèque de Delphi pour rechercher ces fichiers.
5. Vous pouvez sélectionner des répertoires d'installation pour les paquets dont la liste se trouve dans la **clause requires** de n'importe quel paquet de votre

collection. Lorsque vous sélectionnez un bpl dans la liste arborescente, quatre nouveaux champs apparaissent sur la partie droite de l'éditeur de collection de paquets :

- Dans la boîte liste Fichiers exécutables requis, sélectionnez le répertoire dans lequel vous voulez installer les fichiers .bpl pour les paquets dont la liste se trouve dans la **clause requires**. La liste déroulante comprend les répertoires saisis dans Nom de répertoire à l'étape 3 ci-dessus. L'éditeur de collection de paquets recherche ces fichiers en utilisant le chemin de la bibliothèque Delphi et donne la liste sous Fichiers exécutables requis.
- Dans la boîte liste Répertoire des bibliothèques requises, sélectionnez le répertoire d'installation des fichiers.dcp pour les paquets de la **clause requires**. La liste déroulante comprend les répertoires spécifiés sous Nom répertoire à l'étape 3, ci-dessus. L'éditeur de collection de paquets recherche ces fichiers en utilisant le chemin de bibliothèque global de Delphi, et les affiche sous Fichiers bibliothèque requis.

6 Pour enregistrer votre fichier source de collection de paquets, choisissez Fichier|Enregistrer. Les fichiers source de collection de paquets doivent être enregistrés avec l'extension .pce.

7 Pour construire votre collection de paquets, choisissez le turbobouton Compiler. L'éditeur de collection de paquets génère un fichier .dpc avec le nom de votre fichier source (.pce). Si vous n'avez pas encore enregistré le fichier source, l'éditeur vous demande un nom de fichier avant la compilation.

Pour éditer ou recompiler un fichier .pce existant, sélectionnez Fichier|Ouvrir dans l'éditeur de collection de paquets et localisez le fichier sur lequel vous voulez travailler.

Chapitre

12

Création d'applications internationales

Ce chapitre présente les règles d'écriture d'applications qui seront distribuées sur les marchés internationaux. En planifiant le processus, il est possible de réduire le temps et le code nécessaires pour que vos applications puissent fonctionner parfaitement à l'étranger comme sur le marché domestique.

Internationalisation et localisation

Pour créer une application distribuable sur les marchés étrangers, vous devez accomplir deux étapes :

- Internationalisation
- Localisation

Si votre version de Delphi comprend les Outils de traduction, vous pouvez les utiliser pour gérer la localisation. Pour plus d'informations, voir l'aide en ligne des outils de traduction (ETM.hlp).

Internationalisation

L'internationalisation est le processus permettant à votre application de fonctionner selon divers paramètres régionaux. Les paramètres régionaux sont l'environnement de l'utilisateur qui inclut les conventions culturelles du pays cible aussi bien que sa langue. Windows gère un grand nombre de paramètres régionaux, chacun d'eux décrits par l'association d'une langue et d'un pays.

Localisation

La localisation est le processus de traduction d'une application pour qu'elle fonctionne pour des paramètres régionaux spécifiques. Outre la traduction de l'interface utilisateur, la localisation peut également consister à personnaliser les fonctionnalités. Par exemple, une application financière peut être modifiée afin de respecter les règles fiscales dans différents pays.

Internationalisation des applications

Vous devez effectuer les étapes suivantes pour créer des applications internationalisées :

- Le code de votre application doit être capable de gérer des jeux de caractères internationaux.
- Vous devez concevoir l'interface utilisateur de l'application afin de l'adapter aux modifications résultant de la localisation.
- Vous devez isoler toutes les ressources qui ont besoin d'être localisées.

Codage de l'application

Vous vous assurerez que le code de l'application peut gérer les chaînes qu'elle rencontrera dans les divers environnements régionaux cible.

Jeux de caractères

Les versions de Windows diffusées aux USA utilisent le jeu de caractères ANSI Latin-1 (1252). Mais d'autres éditions de Windows utilisent des jeux de caractères différents. Ainsi, la version japonaise de Windows utilise le jeu de caractères Shift-JIS (page de code 932) qui représente les caractères japonais avec des codes sur plusieurs octets.

Il y a, en général, trois types de jeux de caractères :

- Uni-octet (codé sur un seul octet)
- Multi-octet (codé sur plusieurs octets)
- Multi-octet et de longueur fixe

Windows et Linux supportent tous les deux les jeux de caractères sur un seul octet et sur plusieurs octets, comme le jeu Unicode. Dans un jeu sur un seul octet, chaque octet d'une chaîne représente un caractère. Le jeu de caractères ANSI utilisé par de nombreux systèmes d'exploitation occidentaux utilise un seul octet.

Dans un jeu sur plusieurs octets, certains caractères sont représentés par un octet et d'autres par plusieurs octets. Le premier octet d'un caractère sur plusieurs octets est appelé l'octet de poids fort. En général, les 128 premiers caractères d'un jeu multi-octet correspondent aux caractères ASII sur 7 bits, et tout octet dont la valeur ordinale est supérieure à 127 est l'octet de poids fort d'un

caractère multi-octet. Seuls les caractères sur un octet peuvent contenir la valeur null (#0). Les jeux de caractères multi-octets — en particulier les jeux sur deux octets (DBCS) — sont employés pour les langues asatiatiques, tandis que le jeu UTF-8 utilisé par Linux est un encodage sur plusieurs octets du jeu Unicode.

Jeux de caractères OEM et ANSI

Il est parfois nécessaire de faire des conversions entre le jeu de caractères Windows (ANSI) et le jeu de caractères spécifié par la page de code de la machine de l'utilisateur (appelé jeu de caractères OEM).

Jeux de caractères sur plusieurs octets

Les idéogrammes utilisés en Asie ne peuvent se satisfaire de la correspondance 1 pour 1 existant entre les caractères d'une langue et le type *char* qui occupe un seul octet (8 bits). Ces langues ont trop de caractères pour qu'ils soient représentés sur un seul octet comme le fait le type *char*. En revanche, une chaîne multi-octet peut contenir un ou plusieurs octets par caractère. AnsiStrings peut contenir un mélange de caractères sur un et plusieurs octets.

L'octet de poids fort de tous les codes utilisant plusieurs octets appartient à un intervalle réservé et spécifique du jeu de caractères concerné. Le second octet et les octets suivants peuvent être le code d'un autre caractère s'il s'agit d'un caractère codé sur un seul octet, ou il peut appartenir à l'intervalle réservé indiqué par le premier octet s'il s'agit d'un caractère codé sur plusieurs octets. Aussi, la seule façon de savoir si un octet particulier d'une chaîne représente seul un caractère ou fait partie d'un groupe d'octets est de lire la chaîne à partir de l'origine, en la décomposant en caractères de deux ou plusieurs octets chaque fois qu'est rencontré un octet de poids fort appartenant à l'intervalle réservé.

Lors de l'écriture de code destiné aux pays asiatiques, vous devez traiter toutes les manipulations de chaînes avec des fonctions capables de décomposer les chaînes en caractères de plusieurs octets. Delphi fournit de nombreux ses fonctions de la bibliothèque d'exécution pour effectuer cela. En voici quelques unes :

AdjustLineBreaks
AnsiCompareFileName
AnsiExtractQuotedStr
AnsiLastChar
AnsiLowerCase
AnsiLowerCaseFileName
AnsiPos
AnsiQuotedStr
AnsiStrComp
AnsiStrIComp
AnsiStrLastChar
AnsiStrLComp
AnsiStrLIComp
AnsiStrLower
AnsiStrPos
AnsiStrRScan
AnsiStrScan
AnsiStrUpper
AnsiUpperCase
AnsiUpperCaseFileName
ByteToCharIndex
ByteToCharLen
ByteType
ChangeFileExt
CharToByteIndex
CharToByteLen
ExtractFileDir
ExtractFileExt
ExtractFileName
ExtractFilePath
ExtractRelativePath
FileSearch
IsDelimiter
IsPathDelimiter
LastDelimiter
StrByteType
StringReplace
WrapText

N'oubliez pas que la longueur de la chaîne en octets ne correspond pas nécessairement à la longueur de la chaîne en caractères. Faites attention à ne pas tronquer les chaînes en coupant en deux un caractère codé sur plusieurs octets. Vous ne pouvez pas passer des caractères comme paramètres aux fonctions ou aux procédures puisque la taille d'un caractère n'est pas connue directement. Vous devez passer un pointeur sur le caractère ou sur la chaîne.

Caractères larges

Une autre approche de l'utilisation des jeux de caractères pour idéogrammes est de convertir tous les caractères dans un système de caractères larges, comme Unicode. Les caractères et les chaînes Unicode sont également appelés caractères larges et chaînes de caractères larges. Dans le jeu Unicode, chaque caractère est représenté par deux octets. Ainis, une chaîne Unicode est une suite non d'octets séparés mais de mots de deux octets.

Les 256 premiers caractères Unicode correspondent au jeu de caractères ANSI. Le système d'exploitation Windows supporte Unicode (UCS-2). Le système d'exploitation Linux supporte UCS-4, super-ensemble de UCS-2. Delphi/Kylix supporte UCS-2 sur les deux plates-formes. Les caractères larges utilisant deux octets et non un, le jeu de caractères peut représenter beaucoup plus de caractères différents.

En outre, les caractères larges présentent un avantage sur les caractères MBCS : ils vous permettent de conserver vos habitudes car il existe une relation directe entre le nombre d'octets d'une chaîne et son nombre de caractères. Et, vous ne risquez plus de couper un caractère en deux, ni de confondre la seconde moitié d'un caractère avec la première d'un autre.

L'inconvénient majeur des caractères larges est que Windows 9x n'en reconnaît qu'un petit nombre dans les appels aux fonctions API. C'est pourquoi, les composants VCL représentent toutes les valeurs chaînes par des chaînes à caractères d'un seul octet ou par des chaînes MBCS. Vous devrez passer du système caractères larges au système MBCS à chaque fois que définir la propriété d'une chaîne ou en lire la valeur exigerait un supplément de code et ralentirait votre application. Cependant, vous pouvez souhaiter traduire en caractères larges certains algorithmes de traitement des chaînes pour profiter de la correspondance 1 pour 1 entre caractères et *WideChars*.

Inclure des fonctionnalités bi-directionnelles dans les applications

Certaines langues ne se lisent pas de gauche à droite comme la plupart des langues occidentales, mais elles lisent les mots de droite à gauche et comptent de gauche à droite. Ces langues sont dites bi-directionnelles (BiDi) du fait de cette séparation. Les langues bi-directionnelles les plus courantes sont l'Arabe et l'Hébreu, sans parler d'autres langues de l'Est.

TApplication dispose de deux propriétés, *BiDiKeyboard* et *NonBiDiKeyboard*, vous permettant de spécifier la disposition clavier. En outre, la VCL gère la localisation bi-directionnelle via les propriétés *BiDiMode* et *ParentBiDiMode*. Le tableau suivant énumère les objets de la VCL possédant ces propriétés.

Tableau 12.1 Objets de la VCL supportant les BiDi

Page de la palette de composants	Objet de la VCL
Standard	TButton
	TCheckBox
	TComboBox
	TEdit
	TGroupBox
	TLabel
	TListBox
	TMainMenu
	TMemo
	TPanel
	TPopupMenu
	TRadioButton
	TRadioGroup
	TScrollBar
Supplément	TActionMainMenuBar
	TActionToolBar
	TBitBtn
	TCheckListBox
	TColorBox
	TDrawGrid
	TLabeledEdit
	TMaskEdit
	TScrollBox
	TSpeedButton
	TStaticLabel
	TStaticText
	TStringGrid
	TValueListEditor
Win32	TComboBoxEx
	TDateTimePicker
	THeaderControl
	THotKey
	TListView
	TMonthCalendar
	TPageControl
	TRichEdit
	TStatusBar
	TTreeView

Tableau 12.1 Objets de la VCL supportant les BiDi (suite)

Page de la palette de composants	Objet de la VCL
Contrôles de données	TDBCheckBox
	TDBComboBox
	TDBEdit
	TDBGrid
	TDBListBox
	TDBLookupComboBox
	TDBLookupListBox
	TDBMemo
	TDBRadioGroup
	TDBRichEdit
	TDBText
QReport	TQRDBText
	TQRExpr
	TQRLabel
	TQRMemo
	TQRPreview
	TQRSysData
Autres classes	TApplication (sans *ParentBiDiMode*)
	TBoundLabel
	TControl (sans *ParentBiDiMode*)
	TCustomHeaderControl (sans *ParentBiDiMode*)
	TForm
	TFrame
	THeaderSection
	THintWindow (sans *ParentBiDiMode*)
	TMenu
	TStatusPanel
	TTabControl
	TValueListEditor

Remarque *THintWindow* capte la valeur de *BiDiMode* du contrôle qui a activé le conseil.

Propriétés bi-directionnelles

Les objets dont la liste est donnée dans le tableau 12.1, "Objets de la VCL supportant les BiDi", ont les propriétés : *BiDiMode* et *ParentBiDiMode*. Ces propriétés, ainsi que *BiDiKeyboard* et *NonBiDiKeyboard* de *TApplication* gèrent la localisation bi-directionnelle.

Remarque Les propriétés bi-directionnelles ne sont pas disponibles dans CLX pour la programmation multiplate-forme.

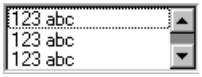
Propriété BiDiMode

La propriété *BiDiMode* est un nouveau type d'énuméré, *TBiDiMode*, qui possède quatre états : *bdLeftToRight*, *bdRightToLeft*, *bdRightToLeftNoAlign* et *bdRightToLeftReadingOnly*.

bdLeftToRight

bdLeftToRight dessine le texte en utilisant le sens de lecture de gauche à droite, l'alignement et la barre de défilement étant inchangés. Par exemple, lors de la saisie de texte de droite à gauche, comme pour l'Arabe ou l'Hébreu, le curseur passe en mode poussoir et le texte est saisi de droite à gauche. Pour du texte latin, comme l'Anglais ou le Français, il est saisi de gauche à droite. *bdLeftToRight* est la valeur par défaut.

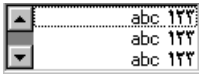
Figure 12.1 Contrôles initialisés à bdLeftToRight



bdRightToLeft

bdRightToLeft dessine le texte en utilisant le sens de lecture de droite à gauche, l'alignement étant modifié et la barre de défilement déplacée. Le texte est saisi normalement pour les langues allant de droite à gauche comme l'Arabe ou l'Hébreu. Lorsque le clavier est modifié pour une langue latine, le curseur passe en mode poussoir et le texte est saisi de gauche à droite.

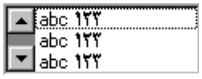
Figure 12.2 Contrôles initialisés à bdRightToLeft



bdRightToLeftNoAlign

bdRightToLeftNoAlign dessine le texte en utilisant le sens de lecture de droite à gauche, l'alignement étant inchangé et la barre de défilement déplacée.

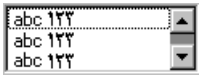
Figure 12.3 Contrôles initialisés à bdRightToLeftNoAlign



bdRightToLeftReadingOnly

bdRightToLeftReadingOnly dessine le texte en utilisant le sens de lecture de droite à gauche, l'alignement et la barre de défilement étant inchangés.

Figure 12.4 Contrôles initialisés à bdRightToLeftReadingOnly



Propriété ParentBiDiMode

ParentBiDiMode est une propriété booléenne. Lorsqu'elle est à *True* (la valeur par défaut), le contrôle regarde la propriété de son parent pour connaître la valeur à utiliser pour *BiDiMode*. Si le contrôle est un objet *TForm*, la fiche utilise la valeur *BiDiMode* de *Application*. Si toutes les propriétés *ParentBiDiMode* sont à *True*, lorsque la propriété *BiDiMode* de *Application* est modifiée, toutes les fiches et tous les contrôles du projet sont initialisés avec la nouvelle valeur.

Méthode FlipChildren

La méthode *FlipChildren* vous permet de faire basculer la position des enfants d'un contrôle conteneur. Les contrôles conteneur sont des contrôles qui contiennent d'autres contrôles, comme *TForm*, *TPanel* et *TGroupBox*. *FlipChildren* possède un seul paramètre booléen, *AllLevels*. Lorsqu'il est à *False*, seuls les enfants directs du contrôle conteneur sont basculés de position. Lorsqu'il est à *True*, tous les enfants du contrôle conteneur sont basculés de position.

Delphi fait basculer la position des contrôles en modifiant la propriété Left et l'alignement du contrôle. Si le côté gauche d'un contrôle est à cinq pixels de la limite gauche de son parent, le basculement provoque l'affichage du côté droit du contrôle de saisie à cinq pixels de la limite droite de son parent. Si le contrôle de saisie est aligné à gauche, un appel à *FlipChildren* provoquera un alignement à droite.

Pour basculer la position d'un contrôle lors de la conception, il faut sélectionner Edition | Transposer les enfants et sélectionner Tous ou Sélectionnés suivant que vous voulez basculer la position de tous les contrôles ou seulement les enfants du contrôle sélectionné. Il est aussi possible de basculer la position d'un contrôle en sélectionnant le contrôle sur la fiche, en cliquant sur le bouton droit de la souris pour sélectionner le choix Transposer les enfants dans le menu contextuel.

Remarque La sélection d'un contrôle de saisie suivi de la commande Transposer les enfants | Sélectionnés ne fait rien. Cela est du au fait que les contrôles de saisie ne sont pas des conteneurs.

Autres méthodes

Il existe d'autres méthodes utiles afin de développer des applications pour des utilisateurs bi-directionnels.

Méthode	Description
OkToChangeFieldAlignment	Utilisée avec les contrôles base de données. Vérifie si l'alignement d'un contrôle peut être modifié.
DBUseRightToLeftAlignment	Utilisée pour vérifier l'alignement des contrôles base de données.
ChangeBiDiModeAlignment	Modifie le paramètre d'alignement qui lui est transmis. Aucune vérification n'est faite pour l'initialisation de *BiDiMode*, car il y a juste conversion de l'alignement à gauche vers l'alignement à droite et vice-versa, en laissant centré les contrôles seuls.
IsRightToLeft	Renvoie *True* si une des options allant de droite à gauche est sélectionnée. Renvoie *False* si le contrôle est dans un mode allant de gauche à droite.

Méthode	Description
UseRightToLeftReading	Renvoie *True* si le contrôle utilise le sens de lecture allant de droite à gauche.
UseRightToLeftAlignment	Renvoie *True* si le contrôle utilise le sens d'alignement allant de droite à gauche. Il peut être surchagé pour être personnalisé.
UseRightToLeftScrollBar	Renvoie *True* si le contrôle utilise une barre de défilement à gauche.
DrawTextBiDiModeFlags	Renvoie les bons paramètres pour le mode BiDi du contrôle.
DrawTextBiDiModeFlagsReadingOnly	Renvoie les bons paramètres pour le mode BiDi du contrôle, en les limitant à la lecture.
AddBiDiModeExStyle	Ajoute le paramètre *ExStyle* flags approprié au contrôle créé.

Fonctionnalités spécifiques aux cibles locales

Vous pouvez ajouter à votre application des fonctionnalités supplémentaires pour des cibles locales spécifiques. En particulier, pour les langues asiatiques, il peut être nécessaire à votre application de contrôler l'IME (Input Method Editor) utilisé pour convertir en chaînes de caractères les touches frappées au clavier par l'utilisateur.

Les composants VCL supportent la programmation de l'IME. La plupart des contrôles fenêtrés autorisant directement la saisie de texte possèdent une propriété *ImeName* qui permet de spécifier l'IME à utiliser lorsque le contrôle reçoit la saisie. Ces contrôles possèdent également une propriété *ImeMode* qui permet de spécifier en quoi l'IME doit convertir ce qui est frappé au clavier. *TWinControl* introduit plusieurs méthodes protégées que vous pouvez utiliser pour contrôler l'IME depuis les classes que vous avez définies. De plus, la variable globale *Screen* vous fournit des informations concernant les IME disponibles sur le système de l'utilisateur.

La variable globale *Screen* (disponible dans la VCL et dans CLX) fournit également des informations concernant l'affectation des touches utilisée sur le système de l'utilisateur. Vous pouvez l'utiliser pour obtenir des informations sur les paramètres régionaux de l'environnement dans lequel tourne votre application.

Conception de l'interface utilisateur

Lorsque vous créez une application pour plusieurs marchés étrangers, il est important de concevoir son interface utilisateur afin qu'elle s'adapte aux modifications effectuées lors de sa traduction.

Texte

Tout le texte apparaissant dans l'interface utilisateur doit être traduit. Le texte anglais étant presque toujours plus court que les traductions, vous devez concevoir les éléments de votre interface utilisateur qui affiche du texte en

réservant de l'espace pour l'expansion de ce texte. Concevez également les boîtes de dialogue, les menus, les barres d'état et les autres éléments de l'interface utilisateur affichant du texte de telle sorte qu'ils puissent facilement afficher des chaînes plus longues. Evitez les abréviations qui ne peuvent exister dans les langues utilisant des idéogrammes.

Les chaînes courtes grandissent plus que les phrases longues. Le tableau suivant fournit une approximation des taux de foisonnement selon la longueur de la chaîne initiale (en anglais) :

Tableau 12.2 Estimation des longueurs de chaîne

Longueur de la chaîne anglaise (en caractères)	Augmentation prévisible
1-5	100%
6-12	80%
13-20	60%
21-30	40%
31-50	20%
over 50	10%

Images graphiques

Le mieux est d'utiliser des images qui ne nécessitent pas de traduction, c'est-à-dire des images qui ne contiennent pas de texte. Si vous devez inclure du texte dans vos images, il est préférable d'utiliser un objet libellé avec arrière-plan transparent par dessus l'image, plutôt que d'inclure le texte dans l'image elle-même.

Voici quelques autres considérations à prendre en compte lors de la création des images graphiques. Essayez d'éviter les images spécifiques à une culture. Par exemple, les boîtes à lettres sont très différentes selon les pays. Les symboles religieux ne conviennent pas aux pays où il existe plusieurs religions dominantes. Même les couleurs ont des connotations symboliques différentes selon les cultures.

Formats et ordre de tri

Les formats de date, formats horaires, numériques et monétaires utilisés dans votre application doivent être localisés selon les paramètres régionaux. Si vous utilisez uniquement les formats de Windows, vous n'avez rien à traduire puisque Windows les lit dans la base de registres de l'utilisateur. Cependant, si vous spécifiez vos propres chaînes de format, déclarez-les comme constantes de ressource afin de pouvoir les localiser.

L'ordre dans lequel les chaînes sont classées dépend également du pays. De nombreuses langues européennes utilisent des caractères accentués et sont classées différemment selon les paramètres régionaux. En outre, certaines combinaisons de deux caractères peuvent être traitées par le tri comme un seul caractère. Par exemple, en espagnol, la combinaison *ch* est triée comme étant un caractère unique compris entre le *c* et le *d*. Parfois, un caractère est trié comme s'il s'agissait de deux caractères séparés, par exemple le *eszett allemand*.

Correspondances entre claviers

Faites attention aux combinaisons de touches utilisées comme raccourcis. Les caractères disponibles sur le clavier américain ne sont pas tous accessibles facilement sur les autres claviers. Lorsque cela est possible, utilisez les touches numériques et les touches de fonction comme raccourcis, puisqu'elles sont aisément accessibles sur tous les claviers.

Isolement des ressources

La partie la plus évidente de la localisation d'une application consiste à traduire les chaînes apparaissant dans l'interface utilisateur. Pour créer une application pouvant être traduite sans modifier le moindre code, les chaînes de l'interface utilisateur doivent être toutes placées dans un seul module. Delphi crée automatiquement un fichier a .dfm (.xfm dans les applications CLX) contenant les ressources des menus, boîtes de dialogue et des bitmaps.

Outre les éléments d'interface apparents, vous devez isoler toutes les chaînes, comme les messages d'erreur proposés à l'utilisateur. Les ressources chaîne ne sont pas incluses dans le fichier fiche. Vous pouvez les isoler en déclarant des constantes au moyen du mot clé **resourcestring**. Pour plus d'informations sur les constantes de chaîne de ressource, voir le guide du langage Pascal Objet. Il vaut mieux inclure toutes les chaînes de ressource dans une seule unité séparée.

Création de DLL de ressources

L'isolement des ressources simplifie le processus de traduction. Le niveau suivant d'isolement des ressources consiste à créer un module DLL. Un module .DLL contient toutes les ressources et uniquement les ressources d'un programme. Les DLL de ressource permettent de créer un programme gérant plusieurs localisations en changeant simplement de DLL de ressource.

Utilisez l'expert Ressource DLL pour créer un module de ressource pour une application. Vous devez avoir ouvert un projet compilé et enregistré pour utiliser l'expert module de ressource. Cela crée un fichier RC contenant les tables de chaîne à partir des fichiers RC utilisés et des chaînes **resourcestring** du projet, et génère un projet pour une DLL de ressource qui contient les fiches et le fichier RES créé. Le fichier RES est compilé à partir du nouveau fichier RC.

Vous devez créer un module de ressource pour chaque traduction que vous voulez gérer. Chaque module de ressource doit avoir une extension du nom de fichier spécifique à la localisation cible. Les deux premiers caractères indiquent la langue cible et le troisième le pays pour la localisation. Si vous utilisez l'expert Ressource DLL, cela est géré pour vous. Sinon, utilisez le code suivant pour obtenir le code local de la traduction cible :

```
unit locales;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
```

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    LocaleList: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    { déclarations privées }
  public
    { déclarations publiques }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}

function GetLocaleData(ID: LCID; Flag: DWORD): string;
var
  BufSize: Integer;
begin
  BufSize := GetLocaleInfo(ID, Flag, nil, 0);
  SetLength(Result, BufSize);
  GetLocaleinfo(ID, Flag, PChar(Result), BufSize);
  SetLength(Result, BufSize - 1);
end;

{ Appelé pour chaque localisation supportée. }
function LocalesCallback(Name: PChar): Bool; stdcall;
var
  LCID: Integer;
begin
  LCID := StrToInt('$' + Copy(Name, 5, 4));
  Form1.LocaleList.Items.Add(GetLocaleData(LCID, LOCALE_SLANGUAGE));
  Result := Bool(1);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  with Languages do
  begin
    for I := 0 to Count - 1 do
    begin
      ListBox1.Items.Add(Name[I]);
    end;
  end;
end;
```

Utilisation des DLL de ressource

L'exécutable, les DLL et les paquets constituant l'application contiennent toutes les ressources nécessaires. Cependant, pour remplacer ces ressources par leurs versions localisées, il suffit simplement de fournir à l'application les DLL de ressource localisées portant le même nom que les fichiers EXE, DLL ou BPL.

Lorsque votre application démarre, elle vérifie les paramètres régionaux du système. Si elle trouve des DLL de ressource ayant les mêmes noms que les fichiers EXE, DLL ou BPL qu'elle utilise, elle examine l'extension de ces DLL. Si l'extension d'un module ressource correspond à la langue et au pays des paramètres régionaux du système, votre application utilise les ressources de ce module plutôt que les ressources de l'exécutable, de la DLL ou du paquet. S'il n'y a pas de module ressource correspondant à la fois à la langue et au pays, votre application essaie de trouver un module ressource correspondant à la langue seule. S'il n'y a pas de module ressource correspondant à la langue, votre application utilise les ressources compilées avec l'exécutable, la DLL ou le paquet.

Si vous voulez que votre application utilise un module de ressource différent de celui correspondant aux paramètres régionaux du système sur lequel elle s'exécute, vous pouvez redéfinir l'entrée spécifiant la localisation dans les registres Windows. Sous l'entrée HKEY_CURRENT_USER\Software\Borland\Locales, ajoutez le chemin d'accès de l'application et le nom de fichier sous la forme d'une chaîne et définissez comme valeur de la donnée l'extension du DLL de ressource. Au démarrage, l'application recherche les DLL de ressource portant cette extension avant de rechercher la localisation du système. L'affectation de cette entrée de registre permet de tester des versions localisées de l'application sans modifier la localisation de votre système.

Par exemple, la procédure suivante peut être utilisée dans le programme d'installation ou de configuration afin de définir la localisation à utiliser au moment de charger des applications Delphi :

```
procedure  SetLocalOverrides(FileName: string, LocaleOverride: string);
var
  Reg: TRegistry;
begin
  Reg := TRegistry.Create;
  try
    if Reg.OpenKey('Software\Borland\Locales', True) then
      Reg.WriteString(LocalOverride, FileName);
  finally
    Reg.Free;
end;
```

Dans votre application, utilisez la fonction globale *FindResourceHInstance* pour obtenir le handle du module de ressource en cours. Par exemple :

```
LoadStr(FindResourceHInstance(HInstance), IDS_AmountDueName, szQuery, SizeOf(szQuery));
```

Vous pouvez ainsi distribuer une seule application qui s'adapte automatiquement à la localisation du système sur laquelle elle s'exécute en fournissant simplement les DLL de ressource.

Basculement dynamique de DLL de ressource

En plus de la localisation d'une DLL de ressource au démarrage de l'application, il est possible de basculer de DLL de ressource dynamiquement lors de l'exécution. Pour ajouter cette fonctionnalité à vos propres applications, vous

devez inclure l'unité ReInit dans votre instruction **uses**. ReInit se trouve dans l'exemple Richedit du répertoire Demos. Pour basculer de langue, vous devez appeler *LoadResourceModule*, en passant le LCID du nouveau langage, puis appeler *ReinitializeForms*.

Par exemple, le code suivant bascule la langue en Français :

```
const
  FRENCH = (SUBLANG_FRENCH shl 10) or LANG_FRENCH;
if LoadNewResourceModule(FRENCH) <> 0 then
  ReinitializeForms;
```

L'avantage de cette technique est que l'instance en cours de l'application et de toutes ses fiches est utilisée. Il n'est pas nécessaire de mettre à jour les paramètres du registre et de redémarrer l'application ou de recharger les ressources nécessaires à l'application, comme la connexion aux serveurs base de données.

Lorsqu'il y a basculement de la DLL de ressource, les propriétés spécifiées dans la nouvelle DLL écrasent celles des instances en cours d'exécution des fiches.

Remarque Toute modification effectuée dans les propriétés d'une fiche lors de l'exécution est perdue. Une fois que la nouvelle DLL est chargée, les valeurs par défaut ne sont pas initialisées. Evitez le code qui réinitialise les objets fiche dans leur état de démarrage, mise à part les différences dues à la localisation.

Localisation des applications

Lorsque votre application est internationalisée, vous devez créer les versions localisées pour les différents marchés étrangers sur lesquels vous souhaitez la distribuer.

Localisation des ressources

Idéalement, vos ressources ont été isolées dans une DLL de ressource qui contient les fichiers fiche (.dfm ou .xfm) et un fichier ressource. Vous pouvez ouvrir vos fiches dans l'EDI et traduire les propriétés importantes.

Remarque Dans un projet DLL de ressource, vous ne pouvez pas ajouter ou supprimer de composant. Pourtant, il est possible de modifier les propriétés, au risque de générer des erreurs d'exécution. Veillez donc à ne modifier que les propriétés qui requièrent une traduction. Pour éviter les erreurs, vous pouvez configurer l'inspecteur d'objets pour n'afficher que les propriétés localisables ; pour ce faire, cliquez avec le bouton droit sur l'inspecteur d'objets et utilisez le menu Voir pour filtrer les catégories de propriétés non souhaitées.

Vous pouvez ouvrir le fichier RC et traduire des chaînes appropriées. Utilisez l'éditeur StringTable en ouvrant le fichier RC à partir du gestionnaire de projet.

Chapitre

13

Déploiement des applications

Une fois que votre application Delphi est terminée et qu'elle fonctionne, vous pouvez la déployer. C'est-à-dire que vous la rendez disponible pour que d'autres l'exécutent. Il est nécessaire d'effectuer un certain nombre d'opérations pour déployer une application sur un autre ordinateur afin que l'application soit entièrement opérationnelle. Les étapes nécessaires pour une application donnée varient suivant le type de l'application. Les sections suivantes décrivent les points à prendre en compte pour déployer les différentes applications :

- Déploiement d'applications généralistes
- Déploiement d'applications CLX
- Déploiement d'applications de bases de données
- Déploiement d'applications Web
- Programmation pour des environnements hôtes hétérogènes
- Termes du contrat de licence logicielle

Remarque Les informations présentées dans ces sections concernent le déploiement d'applications sous Windows. Si vous écrivez des applications multiplates-formes à déployer sous Linux, vous devez vous reporter aux informations sur le déploiement de la documentation Kylix.

Déploiement d'applications généralistes

En dehors du fichier exécutable, une application peut nécessiter des fichiers complémentaires, par exemple des DLL, des fichiers paquets ou des applications complémentaires. De plus, l'application peut nécessiter des entrées dans les registres Windows que ce soit pour spécifier l'emplacement de fichiers auxiliaires ou le paramétrage de l'application. Il est possible d'automatiser avec un programme d'installation, comme InstallShield Express, le processus de copie des fichiers d'une application sur un ordinateur, ainsi que le paramétrage des entrées

de registre. Les étapes suivantes sont les principales étapes d'un déploiement et concernent quasiment tous les types d'application :

- Utilisation des programmes d'installation
- Identification des fichiers de l'application

Les applications Delphi qui accèdent à des bases de données ou qui fonctionnent sur le Web nécessitent des étapes complémentaires d'installation en plus de celles s'appliquant aux applications générales. Pour davantage d'informations sur l'installation d'applications de bases de données, voir "Déploiement d'applications de bases de données" à la page 13-7. Pour davantage d'informations sur l'installation d'applications Web, voir "Déploiement d'applications Web" à la page 13-11. Pour davantage d'informations sur l'installation de contrôles ActiveX, voir "Déploiement d'un contrôle ActiveX sur le Web" à la page 38-16.

Utilisation des programmes d'installation

Les applications Delphi simples, constituées d'un seul fichier exécutable, s'installent facilement sur un ordinateur cible. Il suffit de copier le fichier sur l'ordinateur. Mais les applications plus complexes composées de plusieurs fichiers exigent une procédure d'installation plus sophistiquée. De telles applications nécessitent un programme d'installation spécifique.

Les boîtes à outils d'installation automatisent le processus de création d'un programme d'installation, le plus souvent sans avoir besoin d'écrire une seule ligne de code. Les programmes d'installation créés avec les boîtes à outils d'installation effectuent différentes tâches inhérentes à l'installation des applications Delphi, notamment : copier les fichiers exécutables et les fichiers annexes sur l'ordinateur hôte, établir les entrées dans le registre Windows et installer le moteur BDE pour les applications de bases de données basées sur BDE.

InstallShield Express est une boîte à outils d'installation fournie avec Delphi. InstallShield Express est spécialement adapté à l'utilisation de Delphi et du moteur de bases de données Borland. Il est basé sur la technologie MSI, l'installateur de Windows.

InstallShield Express n'est pas installé automatiquement lors de l'installation de Delphi, et doit être installé manuellement si vous voulez l'utiliser pour créer des programmes d'installation. Exécutez le programme d'installation du CD Delphi pour installer InstallShield Express. Pour davantage d'informations sur l'utilisation de InstallShield Express, voir son aide en ligne.

D'autres boîtes à outils d'installation sont disponibles. Cependant, si vous déployez des applications de bases de données, vous ne devez utiliser que celles basées sur la technologie MSI et celles qui sont certifiées pour déployer le moteur de bases de données Borland (BDE).

Identification des fichiers de l'application

En plus du fichier exécutable, il peut être nécessaire de distribuer de nombreux autres fichiers avec une application.

- Fichiers de l'application
- Fichiers paquet
- Modules de fusion
- Contrôles ActiveX

Fichiers de l'application

Il peut être nécessaire de distribuer les types suivants de fichiers avec une application :

Tableau 13.1 Fichiers de l'application

Type	Extension de nom de fichier
Fichiers programme	.exe et .dll
Fichiers paquet	.bpl et .dcp
Fichiers d'aide	.hlp, .cnt et .toc (s'il est utilisé) ou d'autres fichiers d'aide supportés par votre application
Fichiers ActiveX	.ocx (parfois supportés par une DLL)
Tables des fichiers locaux	.dbf, .mdx, .dbt, .ndx, .db, .px, .y*, .x*, .mb, .val, .qbe, .gd*

Fichiers paquet

Si l'application utilise des paquets d'exécution, il faut distribuer les fichiers paquet avec l'application. InstallShield Express gère l'installation des fichiers paquet de la même manière que les DLL, copie ces fichiers et crée les entrées nécessaires dans les registres Windows. Vous pouvez aussi utiliser des modules de fusion pour déployer des paquets d'exécution avec des outils d'installation basés sur MSI, comme InstallShield Express. Voir la section suivante pour plus de détails.

Borland recommande l'installation des fichiers paquet d'exécution d'origine Borland dans le répertoire Windows\System. Cela sert d'emplacement commun afin que plusieurs applications puissent accéder à une seule instance de ces fichiers. Pour les paquets que vous avez créés, il est recommandé de les installer dans le répertoire de l'application. Seuls les fichiers .BPL doivent être distribués.

Remarque Si vous déployez des paquets avec des applications CLX, vous devez inclure clx60.bpl au lieu de vcl60.bpl.

Si vous distribuez des paquets à d'autres développeurs, fournissez les fichiers .BPL et .DCP.

Modules de fusion

InstallShield Express 3.0 est basé sur la technologie MSI, l'installateur de Windows. C'est pour cette raison que Delphi inclut les modules de fusion. Les modules de fusion fournissent une méthode standard que vous pouvez utiliser pour offrir aux applications du code partagé, des fichiers, des ressources, des entrées du registre et la logique d'installation sous forme d'un seul fichier composé. Vous pouvez utiliser des modules de fusion pour déployer des paquets d'exécution avec des outils d'installation basés sur MSI, comme InstallShield Express.

Les bibliothèques d'exécution ont certaines interdépendances en raison de la façon dont elles ont été regroupées. Le résultat est que lorsqu'un paquet est ajouté à un projet d'installation, l'outil d'installation ajoute automatiquement un ou plusieurs autres paquets ou signale une dépendance sur eux. Par exemple, si vous ajoutez le module de fusion VCLInternet à un projet d'installation, l'outil d'installation va aussi ajouter automatiquement les modules VCLDatabase et StandardVCL ou signaler une dépendance sur ces modules.

Les dépendances de chaque module de fusion sont énumérées dans le tableau suivant. Les divers outils d'installation peuvent réagir différemment à ces dépendances. InstallShield pour l'installateur Windows ajoute automatiquement les modules requis s'il peut les trouver. Les autres outils peuvent se contenter de signaler une dépendance ou peuvent générer un échec de construction si les modules requis ne sont pas tous inclus dans le projet.

Tableau 13.2 Modules de fusion et leurs dépendances

Module de fusion	BPL inclus	Dépendances
ADORTL	adortl60.bpl	DatabaseRTL, BaseRTL
BaseClientDataSet	cds60.bpl	DatabaseRTL, BaseRTL, DataSnap, dbExpress
BaseRTL	rtl60.bpl	Pas de dépendance
BaseVCL	vcl60.bpl, vclx60.bpl	BaseRTL
BDEClientDataSet	bdecds60.bpl	BaseClientDataSet, DataBaseRTL, BaseRTL, DataSnap, DatabaseVCL, BaseVCL, BDERTL
BDEInternet	inetdbbde60.bpl	Internet, DatabaseRTL, BaseRTL, BDERTL
BDERTL	bdertl60.bpl	DatabaseRTL, BaseRTL
DatabaseRTL	dbrtl60.bpl	BaseRTL
DatabaseVCL	vcldb60.bpl	BaseVCL, DatabaseRTL, BaseRTL
DataSnap	dsnap60.bpl	DatabaseRTL, BaseRTL
DataSnapConnection	dsnapcon60.bpl	DataSnap, DatabaseRTL, BaseRTL
DataSnapCorba	dsnapcrba60.bpl	DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL, BaseVCL
DataSnapEntera	dsnapent60.bpl	DataSnap, DatabaseRTL, BaseRTL, BaseVCL
DBCompatVCL	vcldbx60.bpl	DatabaseVCL, BaseVCL, BaseRTL
dbExpress	dbexpress60.bpl	DatabaseRTL, BaseRTL

Tableau 13.2 Modules de fusion et leurs dépendances (suite)

Module de fusion	BPL inclus	Dépendances
dbExpressClientDataSet	dbxcds60.bpl	BaseClientDataSet, DataBaseRTL, BaseRTL, DataSnap, dbExpress
DBXInternet	inetdbxpress60.bpl	Internet, DatabaseRTL, BaseRTL, dbExpress, DatabaseVCL, BaseVCL
DecisionCube	dss60.bpl	TeeChart, BaseVCL, BaseRTL, DatabaseVCL, DatabaseRTL, BDERTL
FastNet	nmfast60.bpl	BaseVCL, BaseRTL
InterbaseVCL	vclib60.bpl	BaseClientDataSet, DatabaseRTL, BaseRTL, DataSnap, DatabaseVCL, BaseVCL
Internet	inet60.bpl, inetdb60.bpl	DatabaseRTL, BaseRTL
InternetDirect	indy60.bpl	BaseVCL, BaseRTL
Office2000Components	dcloffice2k60.bpl	DatabaseVCL, BaseVCL, DatabaseRTL, BaseRTL
QuickReport	qrpt60.bpl	BaseVCL, BaseRTL, BDERTL, DatabaseRTL
SampleVCL	vclsmp60.bpl	BaseVCL, BaseRTL
TeeChart	tee60.bpl, teedb60.bpl, teeqr60.bpl, teeui60.bpl	BaseVCL, BaseRTL
VCLIE	vclie60.bpl	BaseVCL, BaseRTL
VisualCLX	visualclx60.bpl	BaseRTL
WebDataSnap	webdsnap60.bpl	XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL
WebSnap	websnap60.bpl, vcljpg60.bpl	WebDataSnap, XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL, BaseVCL
XMLRTL	xmlrtl60.bpl	Internet, DatabaseRTL, BaseRTL

Contrôles ActiveX

Certains composants fournis avec Delphi sont des contrôles ActiveX. Le conteneur du composant est lié au fichier exécutable de l'application (ou à un paquet d'exécution), mais le fichier .OCX du composant doit également être distribué avec l'application. Ces composants sont :

- Chart FX, copyright par SoftwareFX Inc.
- VisualSpeller Control, copyright par Visual Components, Inc.
- Formula One (tableur), copyright par Visual Components, Inc.
- First Impression (VtChart), copyright par Visual Components, Inc.
- Graph Custom Control, copyright par Bits Per Second Ltd.

Les contrôles ActiveX que vous créez doivent également être enregistrés sur l'ordinateur cible avant d'être utilisés. Les programmes d'installation comme InstallShield Express automatisent le processus d'enregistrement. Pour enregistrer manuellement un contrôle ActiveX, utilisez l'application exempleTRegSvr ou l'utilitaire Microsoft REGSRV32.EXE (qui n'est pas inclus dans toutes les versions de Windows).

Les fichiers DLL gérant un contrôle ActiveX doivent également être distribués avec une application.

Applications complémentaires

Les applications complémentaires sont des programmes distincts en l'absence desquels votre application Delphi fonctionnerait de manière incomplète ou ne fonctionnerait pas du tout. Les applications complémentaires peuvent être celles fournies avec le système d'exploitation, par Borland ou des produits tiers. Le programme utilitaire Server Manager de InterBase est un exemple de programme complémentaire qui permet de gérer les utilisateurs et la sécurité des bases de données InterBase.

Si une application dépend d'un programme complémentaire, assurez-vous de le déployer avec votre application, si c'est possible. La distribution des programmes complémentaires peut être limitée par des accords de licence de distribution. Consultez la documentation d'un programme complémentaire pour des informations spécifiques.

Emplacement des DLL

Vous pouvez installer les fichiers .dll utilisés par une seule application dans le même répertoire que l'application. Les DLL utilisées par plusieurs applications doivent être installées de manière à être partagées par ces applications. La convention courante veut qu'on installe ces fichiers DLL dans les répertoires Windows ou Windows\System. Une autre méthode consiste à créer un répertoire spécifique pour un groupe de fichiers dll associés comme le fait l'installation du moteur de bases de données Borland.

Déploiement d'applications CLX

Si vous écrivez des applications multiplates-formes qui seront déployées à la fois sur Windows et sur Linux, vous devez compiler et déployer les applications sur les deux plates-formes. Les étapes du déploiement des applications CLX sont les mêmes que pour les applications généralistes. Pour des informations sur le déploiement d'applications généralistes, voir "Déploiement d'applications généralistes" à la page 13-1. Pour davantage d'informations sur l'installation d'applications CLX de bases de données, voir "Déploiement d'applications de bases de données" à la page 13-7.

Remarque Quand vous déployez des applications CLX sous Windows, vous devez inclure qtintf.dll avec l'application pour inclure le runtime CLX. Si vous déployez des paquets avec des applications CLX, vous devez inclure clx60.bpl au lieu de vcl60.bpl.

Voir chapitre 10, "Utilisation de CLX pour le développement multiplate-forme", pour des informations sur l'écriture des applications CLX.

Déploiement d'applications de bases de données

Les applications accédant à des bases de données présentent des caractéristiques d'installation propres au-delà de la copie du fichier exécutable de l'application sur l'ordinateur cible. Le plus souvent, l'accès aux bases de données est géré par un moteur de bases de données distinct dont les fichiers ne peuvent être liés au fichier exécutable de l'application. Les fichiers de données, lorsqu'ils n'ont pas été créés au préalable, doivent être rendus accessibles à l'application. Les applications de bases de données multiniveaux nécessitent une gestion encore plus spécialisée de l'installation, car les fichiers constituant l'application doivent être installés sur plusieurs ordinateurs.

Différentes technologies de bases de données (ADO, BDE, dbExpress et InterBase Express) sont supportées et les exigences du déploiement diffèrent pour chacune. Indépendamment de celle que vous utilisez, vous devez vous assurer que le logiciel client est installé sur le système où vous prévoyez d'exécuter l'application de bases de données. BDE, ADO et dbExpress nécessitent également des pilotes pour interagir avec le logiciel client de la base de données. InterBase n'exige aucun pilote car les composants IBX communiquent directement avec la base de données.

Des informations spécifiques sur la façon de déployer les applications de bases de données dbExpress, BDE et multiniveaux sont décrites dans les sections suivantes :

- Déploiement d'applications de bases de données dbExpress
- Déploiement d'applications BDE
- Déploiement d'applications de bases de données multiniveaux (DataSnap)

Les applications de bases de données qui utilisent des ensembles de données client, comme *TClientDataSet* ou *TSQLClientDataSet*, ou des fournisseurs d'ensembles de données nécessitent d'inclure libmidas.dcu et crtl.dcu (pour la liaison statique quand un exécutable autonome est fourni) ; si vous conditionnez votre application (avec l'exécutable et les éventuelles DLL requises), vous devez inclure Midas.dll.

Si vous déployez des applications de bases de données qui utilisent ADO, vous devez vous assurer que MDAC version 2.1 ou ultérieure est installé sur le système où vous prévoyez d'exécuter l'application. MDAC est installé automatiquement avec des logiciels comme Windows 2000 et Internet Explorer version 5 ou ultérieure. Vous devez aussi vous assurer que les pilotes pour le serveur de base de données auquel vous voulez vous connecter sont installés sur le client. Aucune autre étape de déploiement n'est requise.

Si vous déployez des applications de bases de données qui utilisent InterBase Express, vous devez vous assurer que le client InterBase est installé sur le système où vous prévoyez d'exécuter l'application. InterBase nécessite que gd32.dll et interbase.msg soient situés dans un répertoire accessible. Aucune autre étape de déploiement n'est requise. Les composants InterBase Express communiquant directement avec la base de données, aucun pilote

supplémentaire n'est requis. Pour plus d'informations, reportez-vous au Embedded Installation Guide situé sur le site web de Borland.

Outre les technologies décrites ici, vous pouvez utiliser des moteurs de bases de données fournis par des tiers pour gérer l'accès aux bases de données des applications Delphi. Consultez la documentation ou le vendeur du moteur de bases de données pour ce qui concerne les problèmes de droit, d'installation et de configuration du moteur.

Déploiement d'applications de bases de données dbExpress

dbExpress est un ensemble de pilotes qui offrent un accès rapide aux informations des bases de données. Etant également disponibles sous Linux, les composants dbExpress autorisent le développement multiplate-forme. Pour plus d'informations sur l'utilisation des composants dbExpress, voir chapitre 22, "Utilisation d'ensembles de données unidirectionnels".

Vous pouvez déployer des applications dbExpress sous forme d'un fichier exécutable autonome ou sous forme d'un fichier exécutable contenant les DLL des pilotes dbExpress associés.

Pour déployer les applications dbExpress sous forme de fichiers exécutables autonomes, les fichiers objet dbExpress doivent être liés de façon statique dans votre exécutable. Vous faites cela en incluant les DCU suivantes, situées dans le répertoire lib :

Tableau 13.3 Déploiement dbExpress sous forme d'exécutable autonome

Unité de base de données	Quand l'inclure
dbExpInt	Applications se connectant aux bases de données InterBase
dbExpOra	Applications se connectant aux bases de données Oracle
dbExpDb2	Applications se connectant aux bases de données DB2
dbExpMy	Applications se connectant aux bases de données MySQL
Crtl, MidasLib	Requises par les exécutables dbExpress qui utilisent des ensembles de données client comme *TSQLClientDataSet*

Si vous ne déployez pas un exécutable autonome, vous pouvez déployer avec votre exécutable les pilotes dbExpress et les DLL DataSnap associés. Le tableau suivant énumère les DLL appropriées et indique quand il faut les inclure :

Tableau 13.4 Déploiement dbExpress avec les DLL des pilotes

DLL de bases de données	Quand les déployer
dbexpint.dll	Applications se connectant aux bases de données InterBase
dbexpora.dll	Applications se connectant aux bases de données Oracle
dbexpdb2.dll.	Applications se connectant aux bases de données DB2

Tableau 13.4 Déploiement dbExpress avec les DLL des pilotes (suite)

DLL de bases de données	Quand les déployer
dbexpmy.dll	Applications se connectant aux bases de données MySQL
Midas.dll	Requis par les applications de bases de données qui utilisent des ensembles de données client

Déploiement d'applications BDE

Le moteur de bases de données Borland (BDE) définit une API importante pour l'interaction avec les bases de données. De tous les mécanismes d'accès aux données, le BDE gère le plus large éventail de fonctions et offre les meilleurs utilitaires. Il représente le meilleur support de manipulation des données dans les tables Paradox ou dBASE.

L'accès aux bases de données dans une application se fait par le biais de divers moteurs de bases de données. Une application peut utiliser le BDE ou un moteur fourni par un tiers. SQL Links est fourni (mais pas avec toutes les éditions) pour permettre un accès natif aux systèmes de bases de données SQL. Les sections suivantes décrivent l'installation des éléments d'accès aux bases de données d'une application :

- Le moteur de bases de données Borland
- SQL Links

Le moteur de bases de données Borland

Pour utiliser les composants de données standard Delphi ayant un accès aux bases de données, le moteur de bases de données Borland (BDE) doit être présent et disponible. Voir le document BDEDEPLOY pour connaître les droits et restrictions s'appliquant à la distribution du BDE.

Borland recommande l'utilisation de InstallShield Express (ou d'un autre programme d'installation certifié) pour l'installation du BDE. InstallShield Express crée les entrées de registre nécessaires et définit les alias nécessaires à l'application. Il est important d'utiliser un programme certifié pour déployer les fichiers BDE car :

- Une installation incorrecte du BDE ou des sous-ensembles BDE peut empêcher le fonctionnement d'autres applications utilisant le BDE. Ces applications sont des produits Borland, mais également des programmes tiers utilisant le BDE.
- Sous Windows 9x et Windows NT, les informations de configuration BDE sont stockées dans les registres Windows et non pas dans des fichiers .INI, comme c'était le cas avec Windows 16 bits. La création ou la suppression de ces entrées lors de l'installation ou de la désinstallation est une tâche complexe.

Il est possible de n'installer que la partie du BDE nécessaire à une application. Si, par exemple, une application n'utilise que des tables Paradox, il est seulement nécessaire d'installer la partie du BDE indispensable à l'accès aux tables Paradox. Cela réduit l'espace disque nécessaire à une application. Les programmes

d'installation certifiés, comme InstallShield Express, sont capables d'effectuer une installation partielle du BDE. Il faut prendre garde à laisser intacts les fichiers système BDE inutilisés par l'application installée mais nécessaires à d'autres programmes déjà installés.

SQL Links

SQL Links propose les pilotes permettant de connecter une application au logiciel client d'une base de données SQL (via le moteur de bases de données Borland). Voir le document DEPLOY pour connaître les droits et restrictions s'appliquant à la distribution de SQL Links. Comme pour le moteur de bases de données Borland, SQL Links doit être déployé en utilisant InstallShield Express (ou tout autre programme certifié).

Remarque SQL Links connecte le BDE au logiciel client et pas directement à la base de données SQL même. Il est donc toujours nécessaire d'installer le programme client du système de bases de données SQL utilisé. Reportez-vous à la documentation de votre système SQL ou consultez le vendeur pour davantage d'informations sur l'installation et la configuration du logiciel client.

Le tableau suivant présente les noms des fichiers de pilote et de configuration utilisés par SQL Links pour se connecter aux différents systèmes de base de données SQL. Ces fichiers sont fournis avec SQL Links et sont redistribuables en accord avec la licence Delphi.

Tableau 13.5 Fichiers des logiciels client des bases de données SQL

Vendeur	Fichiers redistribuables
Oracle 7	SQLORA32.DLL et SQL_ORA.CNF
Oracle8	SQLORA8.DLL et SQL_ORA8.CNF
Sybase Db-Lib	SQLSYB32.DLL et SQL_SYB.CNF
Sybase Ct-Lib	SQLSSC32.DLL et SQL_SSC.CNF
Microsoft SQL Server	SQLMSS32.DLL et SQL_MSS.CNF
Informix 7	SQLINF32.DLL et SQL_INF.CNF
Informix 9	SQLINF9.DLL et SQL_INF9.CNF
DB/2	SQLDB232.DLL et SQL_DB2.CNF
InterBase	SQLINT32.DLL et SQL_INT.CNF

Installez SQL Links en utilisant InstallShield Express ou tout autre programme d'installation certifié. Pour des informations spécifiques concernant l'installation et la configuration de SQL Links, voir le fichier d'aide SQLLNK32.HLP qui est installé, par défaut, dans le répertoire principal du BDE.

Déploiement d'applications de bases de données multiniveaux (DataSnap)

DataSnap fournit des fonctionnalités de bases de données multiniveaux aux applications Delphi en permettant aux applications client de se connecter aux fournisseurs d'un serveur d'applications.

Installez DataSnap avec une application multiniveau au moyen d'InstallShield Express (ou d'un autre utilitaire de script d'installation certifié Borland). Consultez le document DEPLOY (situé dans le répertoire Delphi principal) pour de plus amples détails sur les fichiers qui nécessitent d'être redistribués avec une application. Reportez-vous aussi au document REMOTE pour obtenir des informations sur les fichiers DataSnap qui peuvent être redistribués et de quelle façon.

Déploiement d'applications Web

Certaines applications Delphi sont conçues pour être exécutées sur le Web, sous la forme de DLL d'extension côté serveur (ISAPI et Apache), d'applications CGI ou de fiches ActiveForm.

Les étapes du déploiement d'applications Web sont identiques à celles des applications généralistes à cette différence que les fichiers de l'application sont déployés sur le serveur Web. Pour des informations sur l'installation de programmes standard, voir "Déploiement d'applications généralistes" à la page 13-1. Pour davantage d'informations sur le déploiement d'applications de bases de données Web, voir "Déploiement d'applications de bases de données" à la page 13-7.

Les considérations suivantes sont spécifiques au déploiement d'applications Web :

- Pour les applications de bases de données BDE, le moteur de bases de données Borland (ou tout autre moteur de bases de données) est installé avec les fichiers de l'application sur le serveur Web.
- Pour les applications dbExpress, les DLL dbExpress doivent être incluses dans le chemin d'accès. S'il est inclus, le pilote *dbExpress* doit être installé avec les fichiers de l'application sur le serveur Web.
- La sécurité pour les répertoires doit être définie de sorte que l'application puisse accéder à tous les fichiers nécessaires de la base de données.
- Le répertoire contenant une application doit avoir des attributs de lecture et d'exécution.
- L'application ne doit pas utiliser de chemins d'accès codés "en dur" pour accéder aux bases de données et aux autres fichiers.
- L'emplacement d'un contrôle ActiveX est indiqué par le paramètre CODEBASE de la balise HTML <OBJECT>.

Le déploiement sur Apache est décrit dans la section suivante.

Déploiement pour Apache

WebBroker supporte Apache version 1.3.9 et ultérieure pour les DLL et les applications CGI. Apache est configuré par fichiers dans le répertoire conf.

Si vous créez des DLL Apache, vous devez vous assurer de définir les directives appropriées dans le fichier de configuration du serveur Apache, nommé httpd.conf. La DLL doit être physiquement située dans le sous-répertoire Modules du logiciel Apache.

Si vous créez des applications CGI, l'option ExecCGI du répertoire physique (spécifié dans la directive Directory du fichier httpd.conf) doit être définie pour permettre l'exécution de programmes, afin que le script CGI puisse être exécuté. Pour vous assurer que les permissions ont été correctement définies, vous devez utiliser la directive ScriptAlias ou bien activer Options ExecCGI.

La directive ScriptAlias crée un répertoire virtuel sur votre serveur et marque le répertoire de destination comme contenant des scripts CGI. Par exemple, ajoutez la ligne suivante dans votre fichier httpd.conf :

```
ScriptAlias /cgi-bin "c:\inetpub\cgi-bin"
```

Cela permettra de satisfaire les demandes comme as /cgi-bin/mycgi grâce à l'exécution du script c:\inetpub\cgi-bin\mycgi.

Vous pouvez aussi définir Options par All ou par ExecCGI en utilisant la directive Directory dans httpd.conf. La directive Options contrôle les fonctionnalités du serveur qui seront disponibles dans un répertoire particulier. Les directives Directory sont utilisées pour entourer un ensemble de directives qui s'appliquent au répertoire indiqué et à ses sous-répertoires. Voici un exemple de la directive Directory :

```
<Directory <apache-root-dir>\cgi-bin>
    AllowOverride None
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>
```

Dans cet exemple, Options est définie par ExecCGI ce qui permet l'exécution des scripts CGI dans le répertoire cgi-bin.

Remarque Apache s'exécute en local sur le serveur sous le compte spécifié par la directive User du fichier httpd.conf. Assurez-vous que l'utilisateur a bien les droits appropriés pour accéder aux ressources nécessaires à l'application.

Des informations sur le déploiement du logiciel Apache se trouvent dans le fichier LICENSE d'Apache, livré avec Apache. Vous trouverez aussi des informations de configuration sur le site web d'Apache, www.apache.org.

Programmation pour des environnements hôtes hétérogènes

En raison des caractéristiques des divers environnements des systèmes d'exploitation, certains éléments peuvent varier selon les préférences de l'utilisateur ou la configuration. Les points suivants peuvent affecter le déploiement d'une application sur un autre ordinateur :

- Résolution d'écran et profondeur de couleurs
- Fontes
- Versions des systèmes d'exploitation
- Applications complémentaires
- Emplacement des DLL

Résolution d'écran et profondeur de couleurs

La taille du bureau et le nombre de couleurs disponibles sur un ordinateur sont configurable et dépendent du matériel installé. Il est probable que ces caractéristiques ne sont pas identiques sur les systèmes utilisés pour le développement et ceux sur lesquels l'application est déployée.

L'aspect d'une application (fenêtres, objets et taille des fontes) sur des ordinateurs utilisant des résolutions différentes peut être géré de différentes manières :

- Concevez l'application avec la plus basse résolution employée par les utilisateurs (généralement, 640x480). Il n'y a rien à faire dans ce cas pour redimensionner les objets dynamiquement afin de les rendre proportionnels à la taille d'affichage de l'écran du système hôte. Visuellement, plus la résolution est importante et plus les objets apparaissent petits.
- Effectuez la conception en utilisant la résolution du système employé pour effectuer le développement et, à l'exécution, redimensionnez dynamiquement toutes les fiches et les objets proportionnellement à la différence de résolution écran entre le système de développement et le système hôte (en utilisant un coefficient de variation entre les résolutions écran).
- Effectuez la conception en utilisant une résolution du système de développement et, à l'exécution, redimensionnez dynamiquement les fiches de l'application. Selon la position des contrôles visuels dans les fiches, cette option peut nécessiter que les fiches disposent de barres de défilement pour que l'utilisateur puisse accéder à tous les contrôles des fiches.

Si vous n'utilisez pas de redimensionnement dynamique

Si les fiches et les contrôles visuels constituant une application ne sont pas redimensionnés dynamiquement à l'exécution, concevez les éléments de l'application en utilisant la résolution la plus basse. Sinon, les fiches d'une application exécutée sur un ordinateur utilisant une résolution d'écran plus faible que celle utilisée pour le système de développement risquent de déborder de l'écran.

Si par exemple, le système de développement est configuré avec une résolution écran de 1024x768 et qu'une fiche est conçue avec une largeur de 700 pixels, une partie de cette fiche ne sera pas visible sur le bureau d'un ordinateur configuré avec une résolution de 640x480.

Si vous redimensionnez dynamiquement les fiches et les contrôles

Si les fiches et les contrôles visuels d'une application sont dynamiquement redimensionnés, adaptez tous les aspects du processus de redimensionnement pour garantir un aspect optimal de l'application pour toutes les résolutions écran possibles. Voici quelques facteurs à considérer lorsque vous redimensionnez dynamiquement les éléments visuels d'une application :

- Le redimensionnement des fiches et des contrôles visuels est effectué en utilisant un ratio calculé en comparant la résolution écran du système de développement à celle du système sur lequel l'application est installée. Utilisez une constante pour représenter une dimension de la résolution écran du système de développement : la hauteur ou la largeur exprimée en pixels. Récupérez à l'exécution la même dimension pour le système de l'utilisateur en utilisant la propriété *TScreen.Height* ou *TScreen.Width* . Divisez la valeur pour le système de développement par la valeur pour le système de l'utilisateur afin d'en dériver le ratio entre les résolutions écran des deux systèmes.
- Redimensionnez les éléments visuels de l'application (fiches et contrôles) en réduisant ou en augmentant la taille des éléments et leur position dans les fiches. Ce redimensionnement est proportionnel à la différence entre les résolutions écran des systèmes du développeur et de l'utilisateur. Redimensionnez et repositionnez automatiquement les contrôles visuels des fiches en définissant la propriété *CustomForm.Scaled* par *True* et en appelant la méthode *TWinControl.ScaleBy* (*TWidgetControl.ScaleBy* pour les applications multiplates-formes). La méthode *ScaleBy* ne modifie pas la hauteur ou la largeur de la fiche. Il faut effectuer cette opération manuellement en multipliant les valeurs en cours des propriétés *Height* et *Width* par le ratio de différence des résolutions écran.
- Les contrôles d'une fiche peuvent être redimensionnés manuellement au lieu d'utiliser la méthode *TWinControl.ScaleBy* (*TWidgetControl.ScaleBy* pour les applications multiplates-formes), en faisant référence à chaque contrôle dans une boucle et en affectant ses dimensions et sa position. La valeur des propriétés *Height* et *Width* des contrôles visuels est multipliée par le ratio de différence des résolutions écran. Repositionnez les contrôles visuels en fonction de la résolution écran en multipliant la valeur des propriétés *Top* et *Left* par le même ratio.
- Si une application a été conçue sur un ordinateur configuré pour une résolution écran supérieure à celle de l'utilisateur, les tailles de fontes seront réduites dans le processus de redimensionnement des contrôles visuels. Si la taille de la fonte lors de la conception est petite, la fonte redimensionnée à l'exécution risque d'être trop petite pour être lisible. Par exemple, supposons que la taille de fonte par défaut d'une fiche est 8. Avec un système de développement ayant une résolution écran de 1024x768 et celui de l'utilisateur une résolution 640x480, les contrôles visuels seront réduits d'un facteur 0,625

(640 / 1024 = 0,625). La taille de fonte d'origine de 8 est réduite à 5 (8 * 0,625 = 5). Le texte de l'application apparaît irrégulier et illisible quand il s'affiche l'affiche dans la fonte réduite.

- Certains contrôles visuels comme *TLabel* et *TEdit* se redimensionnent dynamiquement quand la taille de la fonte du contrôle change. Cela peut affecter les applications déployées quand les fiches et les contrôles sont redimensionnés dynamiquement. Le redimensionnement du contrôle provoqué par la modification de taille de la fonte se cumule à la modification de taille due au redimensionnement proportionnel aux résolutions écran. Cet effet indésirable est neutralisé en définissant la propriété *AutoSize* de ces contrôles par *False*.
- Il faut éviter d'utiliser des coordonnées en pixel explicites, par exemple pour écrire directement dans un canevas. Il faut à la place modifier les coordonnées en leur appliquant un ratio proportionnel au ratio de différence des résolutions écran entre le système de développement et celui d'utilisation. Si, par exemple, l'application dessine un rectangle dans le canevas de dix pixels de haut sur vingt pixels de large, multipliez les valeurs dix et vingt par le ratio de différence de résolution. Ainsi, vous êtes certain que le rectangle apparaît visuellement de la même taille pour différentes résolutions écran.

Adaptation à des profondeurs de couleurs variables

Pour prendre en compte le fait que tous les ordinateurs sur lesquels l'application est déployée ne sont pas configurés avec les mêmes possibilités de couleurs, la solution la plus simple consiste à n'utiliser que des graphiques avec le plus petit nombre possible de couleurs. Cela s'applique particulièrement aux glyphes des contrôles qui doivent utiliser des graphiques en 16 couleurs. Pour l'affichage d'images, vous pouvez soit proposer plusieurs copies de l'image dans différentes résolutions et niveaux de couleur ou indiquer dans l'application la résolution minimale et le nombre de couleurs nécessaires à l'application.

Fontes

Les systèmes d'exploitation Windows et Linux disposent d'un jeu standard de fontes. Quand vous concevez une application devant être déployée sur d'autres ordinateurs, tenez compte du fait que tous les ordinateurs n'ont pas nécessairement de fontes en-dehors des jeux standard.

Les composants texte utilisés dans l'application ne doivent utiliser que des fontes qui sont très probablement disponibles sur les ordinateurs cible.

Quand l'utilisation d'une fonte non standard est absolument nécessaire dans une application, vous devez distribuer cette fonte avec l'application. Soit le programme d'installation, soit l'application même doit installer la fonte sur l'ordinateur cible. La distribution de fontes créées par des tiers peut être sujette à des restrictions imposées par leurs créateurs.

Windows dispose d'une protection contre l'utilisation d'une fonte inexistante sur un système. Il lui substitue une fonte existante, la plus proche possible. Bien que

cela empêche les erreurs dues à des fontes manquantes, le résultat final peut dégrader l'aspect visuel de l'application. Il est préférable de prévoir cette éventualité à la conception.

Pour mettre à la disposition d'une application Windows une fonte non standard, utilisez les fonctions *AddFontResource* et *DeleteFontResource* de l'API Windows. Déployez les fichiers .fot des fontes non-standard avec l'application.

Versions des systèmes d'exploitation

Quand vous utilisez des fonctions de l'API du système d'exploitation ou accédez à des zones du système d'exploitation depuis une application, il y a le risque que cette fonction, cette opération ou cette zone ne soit pas disponible sur des ordinateurs utilisant une version du système différente.

Pour prendre cette possibilité en compte, vous avez différentes possibilités :

- Spécifiez dans les spécifications logicielles de l'application les versions du système sous lesquelles l'application peut s'exécuter. C'est alors à l'utilisateur de n'installer et de n'utiliser l'application que dans des versions compatibles du système d'exploitation.
- Testez la version du système d'exploitation lors de l'installation de l'application. Si une version incompatible du système d'exploitation est détectée, arrêtez le processus d'installation ou prévenez l'utilisateur du problème.
- Testez la version du système d'exploitation à l'exécution, juste avant d'exécuter une opération qui n'est pas applicable à toutes les versions. Si une version incompatible du système d'exploitation est détectée, abandonnez l'opération et informez l'utilisateur. Vous pouvez aussi utiliser du code différent pour les différentes versions du système d'exploitation. Par exemple, certaines opérations sont effectuées de façons différentes sous Windows 95/98 et sous Windows NT/2000. Utilisez la fonction *GetVersionEx* de l'API Windows pour déterminer la version de Windows.

Termes du contrat de licence logicielle

La distribution de certains des fichiers associés aux applications Delphi est sujette à des limitations ou est purement et simplement interdite. Les documents suivants décrivent les stipulations légales concernant la redistribution de ces fichiers :

- DEPLOY
- README
- Contrat de licence
- Documentation de produits vendus par un tiers

DEPLOY

DEPLOY aborde certains aspects légaux de la distribution de divers composants et utilitaires et autres produits pouvant faire partie ou être associés à une application Delphi. DEPLOY est un document installé dans le répertoire principal de Delphi. Il aborde les sujets suivants :

- fichiers .exe, .dll et .bpl
- Les composants et les paquets de conception
- Le moteur de bases de données Borland (BDE)
- contrôles ActiveX
- Les images exemple
- SQL Links

README

README contient des informations de dernière minute Delphi ; il peut donc contenir des informations pouvant affecter les droits de redistribution des composants, utilitaires ou autres éléments. README est un document installé dans le répertoire principal de Delphi.

Contrat de licence

Le contrat de licence Delphi est un document imprimé qui traite des droits et obligations légales concernant Delphi.

Documentation de produits vendus par un tiers

Les droits de redistribution des composants, utilitaires, applications utilitaires, moteurs de bases de données ou autres logiciels provenant d'un tiers sont régis par le vendeur fournissant le produit. Consultez la documentation du produit ou le vendeur pour des informations concernant la redistribution du produit avec une application Delphi avant de le distribuer.

Partie

II

Développement d'applications de bases de données

Les chapitres de cette partie présentent les concepts et les connaissances nécessaires à la création d'applications de bases de données Delphi.

Remarque Vous avez besoin de l'édition Professionnelle ou Entreprise de Delphi pour développer des applications de bases de données. Pour implémenter des bases de données Client/Serveur plus évoluées, vous avez besoin des caractéristiques de Delphi disponibles dans l'édition Entreprise.

Chapitre

14

Conception d'applications de bases de données

Les applications de bases de données permettent aux utilisateurs d'interagir avec les informations stockées dans les bases de données. Les bases de données permettent de structurer les informations et de les partager entre plusieurs applications.

Delphi permet de gérer les applications de bases de données relationnelles. Les bases de données relationnelles organisent les informations en tables, qui contiennent des lignes (enregistrements) et des colonnes (champs). Ces tables peuvent être manipulées par des opérations simples appelées calculs relationnels.

Lorsque vous concevez une application de bases de données, vous devez comprendre comment les données sont structurées. A partir de cette structure, vous pouvez concevoir une interface utilisateur pour afficher les données et permettre à l'utilisateur d'entrer de nouvelles informations et de modifier les données existantes.

Ce chapitre présente certains aspects courants de la conception d'une application de bases de données et les décisions inhérentes à la conception d'une interface utilisateur.

Utilisation des bases de données

Delphi comprend de nombreux composants permettant d'accéder aux bases de données et de représenter les informations qu'elles contiennent. Ils sont regroupés en fonction du mécanisme d'accès aux données :

- La page BDE de la palette de composants contient les composants qui utilisent le moteur de bases de données Borland (BDE, Borland Database Engine). Le BDE définit une API importante pour l'interaction avec les bases de données. De tous les mécanismes d'accès aux données, le BDE gère le plus large

éventail de fonctions et offre les meilleurs utilitaires. Il représente le meilleur support de manipulation des données dans les tables Paradox ou dBASE. Toutefois, c'est le mécanisme le plus compliqué à déployer. Pour plus d'informations sur l'utilisation des composants BDE, voir chapitre 20, "Utilisation du moteur de bases de données Borland".

- La page ADO de la palette de composants contient les composants qui utilisent ActiveX Data Objects (ADO) pour accéder aux informations de bases de données via OLEDB. ADO est un standard Microsoft. De nombreux pilotes ADO permettent de se connecter à différents serveurs de bases de données. Grâce aux composants ADO, vous pouvez intégrer votre application dans un environnement ADO (par exemple, en recourant à des serveurs d'applications ADO). Pour plus d'informations sur l'utilisation des composants ADO, voir chapitre 21, "Utilisation des composants ADO".
- La page dbExpress de la palette de composants contient les composants qui utilisent dbExpress pour accéder aux informations de bases de données. dbExpress est un ensemble de pilotes légers qui offrent l'accès le plus rapide aux informations de bases de données. En outre, étant également disponibles sous Linux, les composants dbExpress autorisent le développement multiplate-forme. Toutefois, les composants base de données dbExpress prennent en charge l'éventail le plus réduit de fonctions de manipulation de données. Pour plus d'informations sur l'utilisation des composants dbExpress, voir chapitre 22, "Utilisation d'ensembles de données unidirectionnels".
- La page InterBase de la palette de composants contient les composants qui accèdent directement aux bases de données InterBase, sans passer par une couche moteur distincte.
- La page AccèsBD de la palette de composants contient les composants utilisables avec tout mécanisme d'accès aux données. Cette page comprend *TClientDataset*, qui peut utiliser les données stockées sur disque ou, par le biais du composant *TDataSetProvider* de cette même page, les composants d'un des autres groupes. Pour plus d'informations sur l'utilisation des ensembles de données client, voir chapitre 23, "Utilisation d'ensembles de données client". Pour plus d'informations sur *TDataSetProvider*, voir chapitre 24, "Utilisation des composants fournisseur".

Remarque Chaque version de Delphi comprend ses propres pilotes d'accès aux serveurs de bases de données par le biais de BDE, ADO ou dbExpress.

Lorsque vous concevez une application de base de données, vous devez choisir l'ensemble de composants à utiliser. Chaque mécanisme d'accès aux données diffère par l'éventail des fonctions prises en charge, la facilité de déploiement et la capacité des pilotes à gérer divers serveurs de bases de données.

Outre un mécanisme d'accès aux données, vous devez choisir un serveur de bases de données. Il existe différents types de bases de données et vous devez prendre en considération les avantages et les inconvénients de chaque type avant de choisir un serveur de bases de données.

Bien que tous ces types de bases de données contiennent des tables qui stockent des informations, certains présentent certaines caractéristiques telles que :

- Sécurité des bases de données
- Transactions
- Intégrité référentielle, procédures stockées et déclencheurs

Types de bases de données

Les serveurs de bases de données relationnelles diffèrent par la façon dont ils stockent les informations et par celle dont ils permettent à plusieurs utilisateurs d'y accéder simultanément. Delphi prend en charge deux types de serveurs de bases de données relationnelles :

- Les **serveurs de bases de données distants** résident sur des machines distinctes. Parfois, les données d'un serveur de bases de données distant ne résident pas sur une seule machine mais sont réparties entre plusieurs serveurs. Bien que les serveurs de bases de données distants ne stockent pas les informations de la même manière, ils fournissent une interface logique commune aux clients, en l'occurrence SQL (Structured Query Language). Le fait que vous y accédiez à l'aide de SQL leur vaut parfois d'être appelés serveurs SQL (ils sont aussi appelés système de gestion de bases de données distant.) Outre les commandes courantes qui composent SQL, la plupart des serveurs de bases de données distants gèrent une variante unique du langage SQL. InterBase, Oracle, Sybase, Informix, Microsoft SQL Server et DB2 sont des exemples de serveurs SQL.
- Les **bases de données locales** résident sur votre disque local ou sur un réseau local. Elles disposent d'interfaces de programmation d'applications propriétaires pour accéder aux données. Lorsqu'elles sont partagées par plusieurs utilisateurs, elles utilisent des mécanismes de verrouillage de fichiers. C'est pourquoi elles sont parfois appelées bases de données à base de fichiers. Paradox, dBASE, FoxPro et Access sont des exemples de bases de données locales.

Les applications qui utilisent des bases de données locales sont appelées **applications à niveau unique** car l'application et la base de données partagent un système de fichiers unique. Les applications qui utilisent des serveurs de bases de données distants sont appelées **applications à niveau double** ou **applications multiniveaux** car l'application et la base de données fonctionnent sur des systèmes (ou niveaux) indépendants.

Le choix du type de base de données à utiliser dépend de plusieurs facteurs. Par exemple, il se peut que vos données soient déjà stockées dans une base de données existante. Si vous créez les tables de bases de données qu'utilise votre application, les points suivants vous intéressent.

- Combien d'utilisateurs partageront ces tables ? Les serveurs de bases de données distants sont conçus pour permettre à plusieurs utilisateurs d'accéder simultanément aux informations. Ils peuvent prendre en charge plusieurs utilisateurs grâce à un mécanisme appelé transactions. Certaines bases de

données locales (telles que Local InterBase) offrent également un support de transaction mais bon nombre ne proposent que des mécanismes de verrouillage de fichiers tandis que certaines ne présentent aucun support multi-utilisateur (tels que les fichiers d'ensembles de données client).

- Quelle quantité de données les tables contiendront-elles ? Les serveurs de bases de données distants peuvent contenir davantage de données que les bases de données locales. Certains serveurs de bases de données distants sont conçus pour stocker des quantités volumineuses de données tandis que d'autres répondent à des impératifs différents (tels que la rapidité des mises à jour).
- Quel type de performance (vitesse) attendez-vous de la base de données ? Les bases de données locales sont généralement plus rapides que les serveurs de bases de données distants car elles résident sur le même système. Chaque serveur de base de données distant étant conçu pour un type d'opération particulier, vous pouvez prendre en compte la question de la performance dans le choix du serveur.
- Quel est le type de support qui sera disponible pour l'administration des bases de données ? Les bases de données locales ne nécessitent pas autant de support que les serveurs de bases de données distants. Généralement, leur coût de fonctionnement est inférieur car elles ne nécessitent pas de serveurs indépendants ni de licences de site onéreuses.

Sécurité des bases de données

Les bases de données contiennent souvent des informations sensibles. Différentes bases de données offrent des schémas de sécurité pour protéger ces informations. Certaines bases de données, comme Paradox et dBASE, n'offrent une protection qu'au niveau des tables ou des champs. Lorsque les utilisateurs essaient d'accéder aux tables protégées, ils doivent fournir un mot de passe. Une fois identifiés, ils ne peuvent visualiser que les champs (colonnes) pour lesquels ils disposent d'une permission.

La plupart des serveurs SQL requièrent un mot de passe et un nom d'utilisateur pour être utilisés. Une fois que l'utilisateur est connecté à la base de données, le nom d'utilisateur et le mot de passe déterminent les tables qu'il peut utiliser. Pour plus d'informations sur l'attribution de mots de passe pour accéder aux serveurs SQL, voir "Contrôle de la connexion au serveur" à la page 17-4.

Lorsque vous concevez des applications de bases de données, vous devez envisager le type d'authentification requis par votre serveur de base de données. Les applications sont souvent conçues de telle sorte que la connexion de base de données explicite soit masquée, si bien que les utilisateurs ont uniquement besoin de se connecter aux applications. Si vous ne souhaitez pas que vos utilisateurs aient besoin de fournir un mot de passe, vous devez soit utiliser une base de données qui n'en requiert pas, soit fournir le mot de passe et le nom d'utilisateur au serveur par programmation. Lorsque vous fournissez le mot de passe par programmation, vous devez veiller à ce que la sécurité ne soit pas violée par lecture du mot de passe à partir de l'application.

Si vous obligez les utilisateurs à fournir un mot de passe, vous devez déterminer à quel moment ce dernier est requis. Si vous utilisez une base de données locale mais envisagez de passer à un serveur SQL plus important, vous pouvez inviter l'utilisateur à fournir son mot de passe au moment où il se connecte à la base de données SQL plutôt qu'à l'ouverture des différentes tables.

Si votre application requiert plusieurs mots de passe pour la connexion à plusieurs bases de données ou systèmes protégés, vous pouvez demander aux utilisateurs de fournir un mot de passe maître unique qui permet d'accéder à une table de mots de passe requis par ces systèmes. L'application fournit alors les mots de passe par programmation, sans que les utilisateurs aient besoin de fournir plusieurs mots de passe.

Dans les applications multiniveaux, vous pouvez utiliser un modèle de sécurité différent. Vous pouvez utiliser HTTPs, CORBA ou COM+ pour contrôler l'accès aux niveaux intermédiaires et laisser ces derniers gérer tous les détails relatifs à l'accès aux serveurs de bases de données.

Transactions

Une transaction est un groupe d'actions qui doivent être menées avec succès sur une ou plusieurs tables dans une base de données avant d'être validées (rendues définitives). Si l'une des actions du groupe échoue, toutes les actions sont abandonnées (annulées).

Les transactions garantissent ce qui suit :

- Toutes les mises à jour d'une même transaction sont soit validées, soit annulées et ramenées à leur état précédent. C'est ce que nous appelons **atomicité**.
- Une transaction est une transformation valide de l'état d'un système, en maintenant les constantes de cet état. C'est ce que nous appelons **cohérence**.
- Les transactions simultanées ne voient pas les résultats partiels et non validés les unes des autres afin de ne pas créer d'incohérences dans l'état de l'application. C'est ce que nous appelons **isolation**.
- Les mises à jour validées des enregistrements survivent aux pannes, y compris les pannes de communication, les pannes de processus et les pannes système des serveurs. C'est ce que nous appelons **durabilité**.

Les transactions protègent ainsi contre les défaillances matérielles qui se produisent au milieu d'une commande de base de données ou d'un ensemble de commandes. L'ouverture d'une transaction vous permet de bénéficier d'un état durable après les pannes survenues sur les supports disque. Les transactions constituent aussi la base du contrôle simultané de plusieurs utilisateurs sur les serveurs SQL. Lorsque tous les utilisateurs interagissent avec la base de données par le biais de transactions, les commandes d'un utilisateur ne peuvent pas altérer l'unité d'une transaction d'un autre utilisateur ; le serveur SQL planifie les transactions entrantes, qui réussissent ou échouent en bloc.

Bien que le support des transactions ne fasse pas partie de la plupart des bases de données locales, il est fourni par InterBase local. En outre, les pilotes du moteur de bases de données Borland offrent pour certaines un support des transactions limité. Le support des transactions de bases de données est fourni par le composant qui représente la connexion à la base de données. Pour plus de détails sur la gestion des transactions à l'aide d'un composant connexion de base de données, voir "Gestion des transactions" à la page 17-6.

Dans les applications multiniveaux, vous pouvez créer des transactions qui comprennent des actions autres que des opérations de bases de données ou qui englobent plusieurs bases de données. Pour plus de détails sur l'utilisation des transactions dans les applications multiniveaux, voir "Gestion des transactions dans les applications multiniveaux" à la page 25-21.

Intégrité référentielle, procédures stockées et déclencheurs

Toutes les bases de données relationnelles présentent certaines caractéristiques communes qui permettent aux applications de stocker et de manipuler les données. En outre, les bases de données offrent souvent des fonctionnalités qui leur sont propres et qui s'avèrent utiles pour garantir la cohérence des relations entre les tables d'une base de données. C'est-à-dire :

- **Intégrité référentielle.** L'intégrité référentielle offre un mécanisme permettant d'éviter la cassure des relations maître/détail entre les tables. Lorsque l'utilisateur essaie de supprimer un champ de la table maître, pouvant aboutir à la création d'enregistrements détail orphelins, les règles de l'intégrité référentielle évitent la suppression ou suppriment automatiquement les enregistrements détail orphelins.
- **Procédures stockées.** Les procédures stockées sont des jeux d'instructions SQL nommés et enregistrés sur un serveur SQL. Les procédures stockées réalisent généralement des tâches de bases de données courantes sur le serveur et renvoient parfois des ensembles d'enregistrements (ensembles de données).
- **Déclencheurs.** Les déclencheurs sont des ensembles d'instructions SQL automatiquement créées en réponse à une commande.

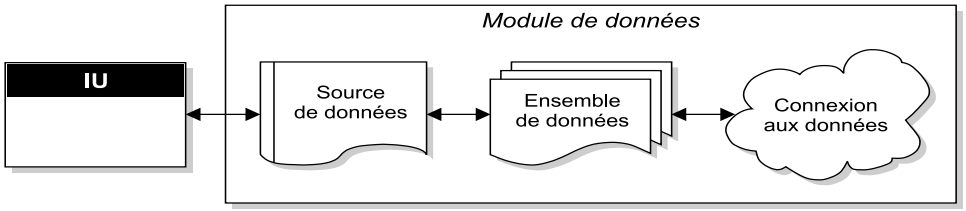
Architecture des bases de données

Les applications de bases de données sont construites à partir d'éléments d'interface utilisateur, de composants qui représentent les informations de bases de données (ensembles de données) et de composants qui connectent ceux-ci les uns aux autres et à la source des informations de bases de données. L'architecture de votre application de base de données représente l'organisation de tous ces éléments.

Structure générale

Bien qu'il existe de nombreuses façons d'organiser les composants d'une application de base de données, la plupart d'entre elles suivent le schéma général illustré par la figure suivante :

Figure 14.1 Architecture de base de données générique



Fiche interface utilisateur

Il est conseillé d'isoler l'interface utilisateur sur une fiche complètement indépendante du reste de l'application. Cela présente plusieurs avantages. L'isolation de l'interface utilisateur des composants qui représentent les informations de bases de données vous apporte une plus grande flexibilité conceptuelle : les modifications que vous apportez à la gestion des informations de bases de données n'imposent pas la réécriture de l'interface utilisateur, tandis que celles que vous apportez à l'interface utilisateur ne vous obligent pas à modifier la partie de l'application qui utilise la base de données. En outre, ce type d'isolation vous permet de développer des fiches communes à diverses applications, ce qui garantit la cohérence de l'interface utilisateur. De plus, si le référentiel d'objets contient des liens vers des fiches convenablement conçues, vous -même et les autres développeurs disposez d'une base de travail et n'êtes pas obligés de commencer chaque nouveau projet à partir d'aucun élément. Enfin, le partage des fiches vous permet de développer des standards d'entreprise pour les interfaces des applications. Pour plus d'informations sur la création de l'interface utilisateur d'une application de base de données, voir "Conception de l'interface utilisateur" à la page 14-17.

Module de données

Si vous avez isolé votre interface utilisateur dans sa propre fiche, vous pouvez utiliser un module de données afin d'y placer les composants qui représentent les informations de bases de données (ensembles de données), et les composants qui connectent ces ensembles de données aux autres éléments de votre application. Comme les fiches de l'interface utilisateur, les modules de données peuvent figurer dans le référentiel d'objets en vue d'être réutilisés ou partagés par les applications.

Source de données

Le premier élément du module de données est une source de données. La source de données relie l'interface utilisateur à un ensemble de données qui représente les informations d'une base de données. Plusieurs contrôles orientés données disposés sur une fiche peuvent partager une même source de données. Dans ce

cas, le contenu de chaque contrôle est synchronisé : lorsque l'utilisateur parcourt les enregistrements, les valeurs figurant dans les différents champs de l'enregistrement actif sont affichées dans les contrôles correspondants.

Ensemble de données

L'ensemble de données constitue le cœur de votre application de base de données. Ce composant représente un ensemble d'enregistrements de la base de données sous-jacente. Ces enregistrements peuvent être les données d'une seule table de base de données, un sous-ensemble des champs ou des enregistrements d'une table ou des informations émanant de plusieurs tables jointes en une vue unique. L'utilisation d'ensembles de données protège la logique de votre application de la restructuration des tables physiques de la base de données. Lorsque la base de données sous-jacente change, vous pouvez être amené à modifier la façon dont le composant ensemble de données spécifie les données qu'il contient, mais le reste de votre application peut continuer à fonctionner sans subir de modifications. Pour plus d'informations sur les propriétés et méthodes courantes des ensembles de données, voir chapitre 18, "Présentation des ensembles de données".

Connexion des données

Différents types d'ensembles de données utilisent différents mécanismes de connexion aux informations de la base de données sous-jacente. Ces différents mécanismes déterminent les variantes majeures de l'architecture des applications de bases de données que vous créez. Il existe essentiellement quatre mécanismes de connexion aux données :

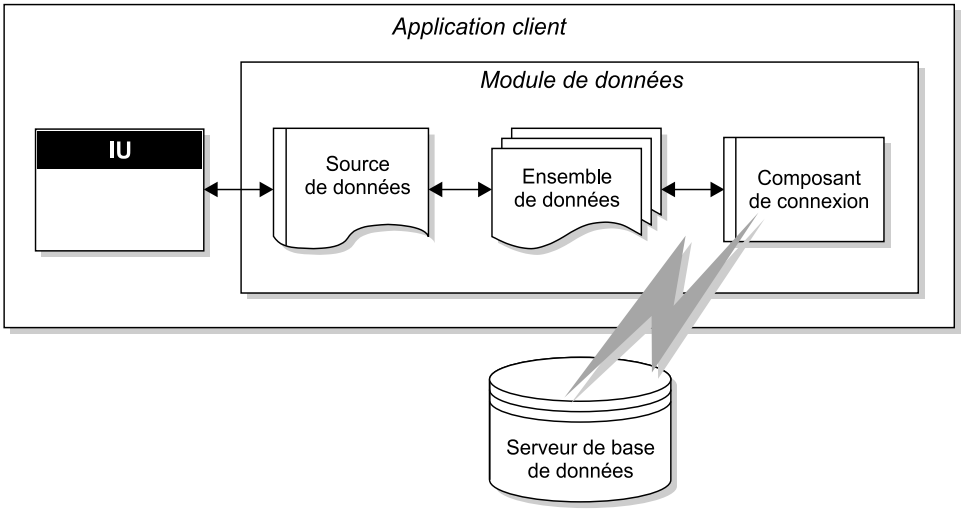
- Connexion directe à un serveur de bases de données. La plupart des ensembles de données utilisent un descendant de *TCustomConnection* pour représenter la connexion à un serveur de bases de données.
- Utilisation d'un fichier dédié sur disque. Les ensembles de données client permettent d'utiliser un fichier dédié sur disque. Aucun composant connexion séparé n'est requis lors de l'utilisation d'un fichier dédié car l'ensemble de données client est en mesure de lire et d'écrire dans le fichier.
- Connexion à un autre ensemble de données. Les ensembles de données client peuvent utiliser les données fournies par un autre ensemble de données. Un composant *TDataSetProvider* fait office d'intermédiaire entre l'ensemble de données client et son ensemble de données source. Ce fournisseur d'ensemble de données peut résider dans le même module de données que l'ensemble de données client ou faire partie d'un serveur d'application exécuté sur une autre machine. Si le fournisseur fait partie d'un serveur d'application, vous devez utiliser un descendant spécial de *TCustomConnection* pour représenter la connexion au serveur.
- Obtention des données à partir d'un objet DataSpace RDS. Les ensembles de données ADO peuvent utiliser un composant *TRDSConnection* pour rassembler les données des applications de bases de données multiniveaux élaborées à l'aide de serveurs d'applications ADO.

Parfois, ces mécanismes peuvent être combinés en une même application.

Connexion directe à un serveur de bases de données

L'architecture de base de données la plus courante est celle dans laquelle l'ensemble de données utilise un composant connexion pour établir une connexion à serveur de bases de données. L'ensemble de données peut alors directement lire les données du serveur et y envoyer les modifications. Cette architecture est illustrée par la figure suivante :

Figure 14.2 Connexion directe au serveur de bases de données



Chaque type d'ensemble de données utilise son propre type de composant connexion, qui représente un mécanisme d'accès aux données unique :

- Si l'ensemble de données est un ensemble de données BDE tel que *TTable*, *TQuery* ou *TStoredProc*, le composant connexion est un objet *TDataBase.* Vous connectez l'ensemble de données au composant base de données en définissant sa propriété *Database.* Vous n'avez pas besoin d'ajouter explicitement un composant base de données lorsque vous utilisez un ensemble de données BDE. Si vous définissez la propriété *DatabaseName* de l'ensemble de données, un composant base de données est automatiquement créé à l'exécution.
- Si l'ensemble de données est un ensemble de données ADO tel que *TADODataSet*, *TADOTable*, *TADOQuery* ou *TADOStoredProc*, le composant connexion est un objet *TADOConnection.* Vous connectez l'ensemble de données au composant connexion ADO en définissant sa propriété *ADOConnection.* Comme pour les ensembles de données BDE, vous n'avez pas besoin d'ajouter explicitement le composant connexion : par contre, vous pouvez définir la propriété *ConnectionString* de l'ensemble de données.

- Si l'ensemble de données est un ensemble de données dbExpress tel que *TSQLDataSet*, *TSQLTable*, *TSQLQuery* ou *TSQLStoredProc*, le composant connexion est un objet *TSQLConnection.* Vous connectez l'ensemble de données au composant connexion SQL en définissant sa propriété *SQLConnection.* Lorsque vous utilisez des ensembles de données dbExpress, vous devez explicitement ajouter le composant connexion. En outre, les ensembles de données dbExpress présentent la particularité d'être toujours unidirectionnels et accessibles en lecture seule : cela signifie que vous pouvez uniquement parcourir les enregistrements dans l'ordre et que vous ne pouvez pas utiliser les méthodes d'édition des ensembles de données.
- Si l'ensemble de données est un ensemble de données InterBase Express tel que *TIBDataSet*, *TIBTable*, *TIBQuery* ou *TIBStoredProc,* le composant connexion est un objet *TIBDatabase.* Vous connectez l'ensemble de données au composant base de données InterBase en définissant sa propriété *Database.* Comme dans le cas des ensembles de données dbExpress, vous devez explicitement ajouter le composant connexion.

Outre les composants précédemment cités, vous pouvez utiliser un ensemble de données client spécialisé tel que *TBDEClientDataSet, TSQLClientDataSet* ou *TIBClientDataSet* avec un composant connexion de base de données. Lorsque vous utilisez l'un de ces ensembles de données client, spécifiez le type approprié de composant connexion comme valeur de la propriété *DBConnection*.

Bien que chaque type d'ensemble de données utilise un composant connexion différent, ils effectuent tous la plupart des mêmes tâches et mettent à disposition la plupart des mêmes propriétés, méthodes et événements. Pour plus d'informations sur les points communs des différents composants connexion de base de données, voir chapitre 17, "Connexion aux bases de données".

Cette architecture représente une application à niveau unique ou une application à niveau double, selon que le serveur de base de données est une base de données locale ou un serveur de base de données distant. La logique qui manipule les informations de bases de données figure dans l'application qui implémente l'interface utilisateur, tout en étant confinée dans un module de données.

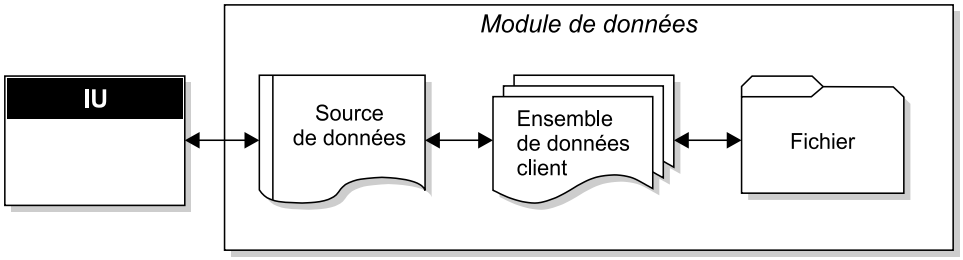
Remarque Toutes les versions de Delphi ne proposent pas les pilotes ou composants connexion requis pour créer des applications à niveau double.

Utilisation d'un fichier dédié sur disque

Dans sa forme la plus simple, une application de base de données que vous écrivez ne recourt à aucun serveur de bases de données. Par contre, elle utilise MyBase, exploite la possibilité qu'ont les ensembles de données client de s'enregistrer eux-mêmes dans un fichier puis d'en extraire les données.

Cette architecture est illustrée par la figure suivante :

Figure 14.3 Application de base de données à base de fichiers



Lorsque vous utilisez cette approche à base de fichiers, votre application écrit les modifications sur disque à l'aide de la méthode *SaveToFile* de l'ensemble de données client. *SaveToFile* accepte un paramètre, le nom du fichier qui est créé (ou écrasé) et qui contient la table. Lorsque vous souhaitez lire une table précédemment écrite à l'aide de la méthode *SaveToFile*, utilisez la méthode *LoadFromFile*. *LoadFromFile* accepte aussi un paramètre, le nom du fichier contenant la table.

Si vous chargez les données toujours à partir du même fichier et les y enregistrez toujours, vous pouvez utiliser la propriété *FileName* au lieu des méthodes *SaveToFile* et *LoadFromFile*. Lorsque *FileName* a pour valeur un nom de fichier valide, les données sont automatiquement chargées à partir du fichier à l'ouverture de l'ensemble de données client et enregistrées dans le fichier à la fermeture de l'ensemble de données client.

Cette architecture simple de type fichier est une application à niveau unique. La logique qui manipule les informations de bases de données figure dans l'application qui implémente l'interface utilisateur, tout en étant confinée dans un module de données.

L'approche de type fichier présente l'avantage de la simplicité. Aucun serveur de bases de données ne doit être installé, configuré ou déployé (si vous n'effectuez pas de liaison statique dans midaslib.dcu, l'ensemble de données client requiert midas.dll). Aucune licence de site ni administration de base de données n'est nécessaire.

En outre, certaines versions de Delphi vous permettent de réaliser des conversions entre des documents XML arbitraires et les paquets de données utilisés par un ensemble de données client. Par conséquent, l'approche à base de fichiers permet d'utiliser des documents XML ainsi que des ensembles de données dédiés. Pour plus d'informations sur la conversion entre des documents XML et des paquets de données d'ensemble de données client, voir chapitre 26, "Utilisation de XML dans les applications de bases de données".

L'approche à base de fichiers ne gère pas les situations multi-utilisateurs. L'ensemble de données doit être totalement dédié à l'application. Les données sont enregistrées dans des fichiers sur disque puis chargées ultérieurement, mais aucune protection intégrée n'empêche un utilisateur d'écraser les fichiers de données d'un autre utilisateur.

Pour plus d'informations sur l'utilisation d'un ensemble de données client dont les données sont stockées sur disque, voir "Utilisation d'un ensemble de données client avec des données basées sur des fichiers" à la page 23-39.

Connexion à un autre ensemble de données

Certains ensembles de données client spécialisés utilisent le BDE ou *dbExpress* pour se connecter à un serveur de bases de données. Ces ensembles de données client spécialisés sont, de fait, des composants composites qui utilisent de manière interne un autre ensemble de données pour accéder aux données et un composant fournisseur pour empaqueter les données de l'ensemble de données source et appliquer les mises à jour au serveur de bases de données. Ces composants composites requièrent des ressources système supplémentaires mais présentent certains avantages :

- Les ensembles de données client offrent le support le plus robuste pour la manipulation des mises à jour placées en mémoire cache. Par défaut, les autres types d'ensembles de données envoient directement les modifications au serveur de bases de données. Vous pouvez réduire le trafic réseau en utilisant un ensemble de données qui place les mises à jour en mémoire cache localement puis les applique toutes en une seule transaction. Pour plus d'informations sur les avantages de l'utilisation d'ensembles de données client pour placer les mises à jour en mémoire cache, voir "Utilisation d'un ensemble de données client pour mettre en cache les mises à jour" à la page 23-18.
- Les ensembles de données client peuvent appliquer les modifications directement au serveur de bases de données lorsque l'ensemble de données est accessible en lecture seule. Lorsque vous utilisez *dbExpress*, outre que c'est le seul moyen de parcourir librement les données, c'est la seule façon de modifier les données dans l'ensemble de données. Même lorsque vous n'utilisez pas *dbExpress*, le résultat de certaines requêtes et de toutes les procédures stockées est accessible en lecture seule. L'utilisation d'un ensemble de données client offre un procédé standard pour rendre ces données modifiables.
- Un ensemble de données client pouvant directement utiliser des fichiers dédiés sur disque, son utilisation peut être combinée avec un modèle à base de fichiers afin d'offrir une application "briefcase" souple. Pour plus d'informations sur le modèle "briefcase", voir "Combinaison des approches" à la page 14-16.

Outre ces ensembles de données client spécialisés, il existe un ensemble de données client générique (*TClientDataSet*), qui ne comprend pas de fournisseur d'ensemble de données ni d'ensemble de données interne. Bien que *TClientDataSet* ne possède pas de mécanisme interne d'accès aux bases de données, vous pouvez le connecter à un autre ensemble de données externe à partir duquel il peut lire les données et vers lequel il peut envoyer les mises à jour.

Malgré son caractère quelque peu complexe, cette approche s'avère parfois opportune :

- L'ensemble de données source et le fournisseur d'ensemble de données étant externes, vous pouvez plus facilement contrôler la façon dont ils lisent les données et appliquent les mises à jour. Par exemple, le composant fournisseur met à disposition une série d'événements qui ne sont pas disponibles lorsque vous utilisez un ensemble de données client spécialisé pour accéder aux données.
- Lorsque l'ensemble de données source est externe, vous pouvez le lier à un autre ensemble de données dans une relation maître/détail. Un fournisseur externe convertit automatiquement cette organisation en un seul ensemble de données comprenant des détails imbriqués. Lorsque l'ensemble de données source est interne, vous ne pouvez pas créer d'ensembles détail imbriqués de cette façon.
- La connexion d'un ensemble de données client à un ensemble de données externe est une architecture qui peut facilement évoluer vers une architecture multiniveau. Etant donné que le processus de développement est d'autant plus coûteux et consommateur d'énergie que le nombre de niveaux augmente, vous pouvez commencer à développer votre application en tant qu'application à niveau unique ou double. A mesure qu'augmenteront la quantité de données, le nombre d'utilisateurs et le nombre des différentes applications accédant aux données, vous serez éventuellement amené à adopter une architecture multiniveau. Si vous pensez utiliser à terme une architecture multiniveau, il peut être judicieux de commencer en utilisant un ensemble de données client avec un ensemble de données source externe. Ainsi, lorsque vous transférez vers un niveau intermédiaire la logique de l'accès et de la manipulation des données, le développement effectué est protégé car le code est réutilisable à mesure que l'application prend de l'ampleur.
- *TClientDataSet* peut être lié à tout ensemble de données source. Cela signifie que vous pouvez utiliser des ensembles de données personnalisés (composants tierces parties) pour lesquels n'existe aucun ensemble de données client spécialisé correspondant. Certaines versions de Delphi incluent même des composants fournisseur spéciaux qui connectent un ensemble de données client à un document XML plutôt qu'à un autre ensemble de données. (Cela fonctionne de la même façon que la connexion d'un ensemble de données client à un autre ensemble de données (source), à la différence que le fournisseur XML utilise un document XML à la place d'un ensemble de données. Pour plus d'informations sur ces fournisseurs XML, voir "Utilisation d'un document XML comme source pour un fournisseur" à la page 26-9.)

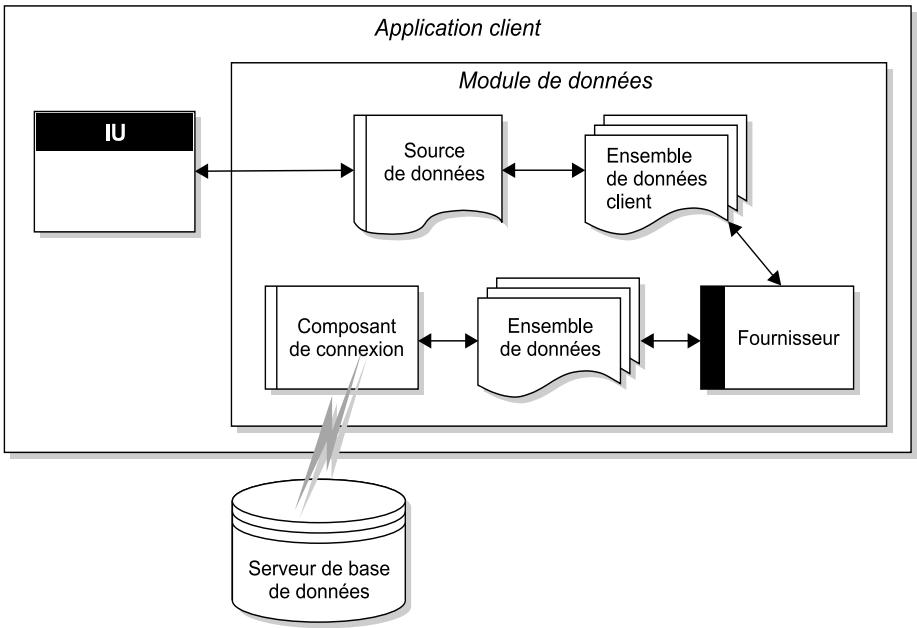
L'architecture qui connecte un ensemble de données client à un ensemble de données externe se présente sous deux versions :

- Connexion d'un ensemble de données client à un autre ensemble de données dans la même application.
- Utilisation d'une architecture multiniveau.

Connexion d'un ensemble de données client à un autre ensemble de données dans la même application

Le fait d'utiliser un composant fournisseur vous permet de vous connecter *TClientDataSet* à un autre ensemble de données (source). Le fournisseur assemble les informations de bases de données en paquets de données transportables (utilisables par les ensembles de données client) et applique à un serveur de base de données les mises à jour reçues dans les paquets delta (que les ensembles de données client créent). Cette architecture est illustrée par la figure suivante :

Figure 14.4 Architecture combinant un ensemble de données client et un autre ensemble de données



Cette architecture représente une application à niveau unique ou une application à niveau double, selon que le serveur de base de données est une base de données locale ou un serveur de base de données distant. La logique qui manipule les informations de bases de données figure dans l'application qui implémente l'interface utilisateur, tout en étant confinée dans un module de données.

Pour lier l'ensemble de données client au fournisseur, attribuez à sa propriété *ProviderName*le nom du composant fournisseur. Le fournisseur doit se trouver dans le même module de données que l'ensemble de données client. Pour lier le fournisseur à l'ensemble de données source, définissez sa propriété *DataSet*.

Une fois que l'ensemble de données client est lié au fournisseur et que celui-ci est lié à l'ensemble de données source, ces composants gèrent automatiquement tous les aspects inhérents à la lecture, à l'affichage et à l'exploration des enregistrements de bases de données (sous réserve que l'ensemble de données source soit connecté à une base de données). Pour appliquer les modifications

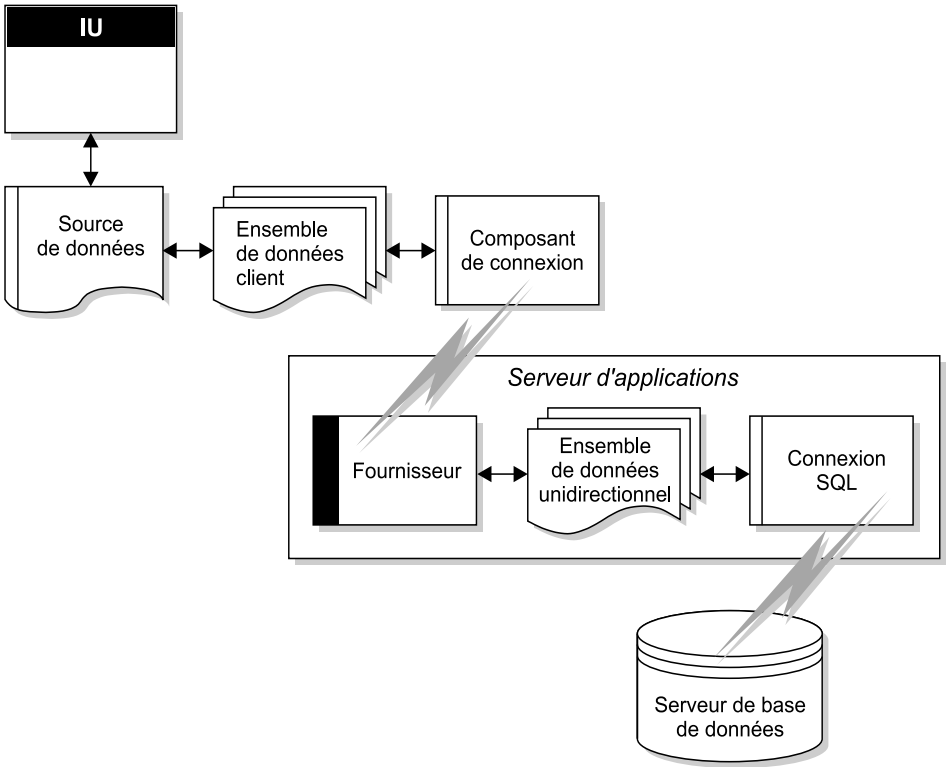
utilisateur à la base de données, vous devez uniquement appeler la méthode *ApplyUpdates* de l'ensemble de données client.

Pour plus d'informations sur l'utilisation d'un ensemble de données client avec un fournisseur, voir "Utilisation d'un ensemble de données client avec un fournisseur" à la page 23-29.

Utilisation d'une architecture multiniveau

Lorsque les informations de bases de données comprennent des relations complexes entre plusieurs tables ou que le nombre de clients s'accroît, vous pouvez utiliser une application multiniveau. Les applications multiniveaux comprennent des niveaux intermédiaires entre l'application cliente et le serveur de base de données. Cette architecture est illustrée par la figure suivante :

Figure 14.5 Architecture de base de données multiniveau



La figure précédente représente une application à trois niveaux. La logique qui manipule les informations de bases de données se trouve sur un système indépendant, ou niveau. Ce niveau intermédiaire centralise la logique qui gouverne les interactions avec votre base de données et offre ainsi un contrôle centralisé des relations entre les données. Cela permet à différentes applications clientes d'utiliser les mêmes données tout en garantissant l'homogénéité de la

logique des données. Les applications multiniveaux autorisent aussi les applications clientes de taille réduite car la majeure partie du traitement est déplacée vers le niveau intermédiaire. Ces applications clientes de taille réduite sont plus faciles à installer, à configurer et à gérer. Les applications multiniveaux peuvent aussi améliorer les performances en répartissant le traitement des données sur plusieurs systèmes.

L'architecture multiniveau s'apparente beaucoup au modèle précédent. Elle s'en distingue essentiellement par le fait que l'ensemble de données source, qui se connecte au serveur de base de données, et le fournisseur, qui fait office d'intermédiaire entre cet ensemble de données source et l'ensemble de données client, ont tous les deux été déplacés vers une application séparée. Cette application est appelée serveur d'applications (ou parfois "courtier ou agent des données distantes").

Étant donné que le fournisseur a été déplacé vers une application séparée, l'ensemble de données client ne peut plus se connecter à l'ensemble de données source en définissant simplement sa propriété *ProviderName*. En outre, il doit utiliser un type de composant connexion pour rechercher le serveur d'applications et s'y connecter.

Plusieurs types de composants connexion peuvent connecter un ensemble de données client à un serveur d'applications. Ils dérivent tous de *TCustomRemoteServer* et diffèrent essentiellement par le protocole de communication utilisé (TCP/IP, HTTP, DCOM, SOAP, ou CORBA). Liez l'ensemble de données client à son composant connexion en définissant la propriété *RemoteServer*.

Le composant connexion établit une connexion au serveur d'applications et renvoie une interface que l'ensemble de données client utilise pour appeler le fournisseur spécifié par sa propriété *ProviderName*. Chaque fois que l'ensemble de données client appelle le serveur d'applications, il transmet la valeur de *ProviderName* puis le serveur d'applications transmet l'appel au fournisseur.

Pour plus d'informations sur la connexion d'un ensemble de données client à un serveur d'applications, voir chapitre 25, "Création d'applications multiniveaux".

Combinaison des approches

Les sections précédentes décrivent différentes architectures utilisables lors de l'écriture d'applications de bases de données. Rien ne vous empêche, toutefois, de combiner deux ou plusieurs des architectures disponibles dans une même application. De fait, certaines combinaisons peuvent s'avérer très puissantes.

Par exemple, vous pouvez combiner l'architecture de type disque décrite dans "Utilisation d'un fichier dédié sur disque" à la page 14-10, avec une approche telle que celles décrites dans "Connexion d'un ensemble de données client à un autre ensemble de données dans la même application" à la page 14-14 ou "Utilisation d'une architecture multiniveau" à la page 14-15. Ces combinaisons sont simples car tous les modèles utilisent un ensemble de données client pour représenter les données qui apparaissent dans l'interface utilisateur. Le résultat

est appelé modèle "briefcase" (ou parfois modèle "déconnecté" ou informatique nomade).

Le modèle "briefcase" peut, par exemple, s'avérer utile dans la situation suivante : la base de données sur site d'une entreprise contient des informations de contacts clients que les représentants peuvent utiliser et mettre à jour à l'extérieur de l'entreprise. Ils peuvent télécharger des informations à partir de la base de données, exploiter ces données sur leurs ordinateurs portables pendant qu'ils sont en déplacement dans tout le pays, et même mettre à jour des enregistrements sur des sites existants ou de nouveaux sites. Quand les représentants reviennent dans l'entreprise, ils doivent charger leurs changements de données dans la base de données de l'entreprise pour les mettre à la disposition de tous.

Lors d'une opération sur site, l'ensemble de données client d'une application de modèle "briefcase" obtient ses données d'un fournisseur. L'ensemble de données client est ensuite connecté au serveur de base de données et peut, par le biais du fournisseur, récupérer des données du serveur et lui communiquer des mises à jour. Avant de se déconnecter du fournisseur, l'ensemble de données client enregistre sa capture instantanée des informations dans un fichier sur disque. Hors site, l'ensemble de données client charge ses données à partir du fichier et enregistre toute modification dans ce fichier. Dans une dernière phase, une fois de nouveau sur site, l'ensemble de données client se reconnecte au fournisseur afin d'appliquer ses mises à jour au serveur de base de données ou d'actualiser sa capture instantanée des données.

Conception de l'interface utilisateur

La page ContrôleBD de la palette des composants offre un ensemble de contrôles orientés données qui représentent les données de champs d'un enregistrement de base de données et qui permettent aux utilisateurs de modifier ces données et de répercuter ces modifications dans la base de données. L'utilisation de contrôles orientés données vous permet de construire l'interface utilisateur votre application de base de données de sorte que ces informations soient visibles et accessibles aux utilisateurs. Pour plus d'informations sur les contrôles orientés données, voir chapitre 15, "Utilisation de contrôles de données"

Outre les contrôles de données élémentaires, vous pouvez également intégrer d'autres éléments dans votre interface utilisateur :

- Votre application peut analyser les données contenues dans une base de données. Les applications qui analysent les données d'une base de données ne se contentent pas de les afficher ; elles les résument sous forme conviviale afin que les utilisateurs en saisissent l'impact.
- Vous pouvez imprimer des rapports afin d'obtenir une copie imprimée des informations affichées dans votre interface utilisateur.
- Vous pouvez créer une interface utilisateur qui peut être visualisée à l'aide de navigateurs Web. Les applications de bases de données Web les plus simples sont décrites dans "Utilisation des bases de données dans les réponses" à la

page 28-18. En outre, vous pouvez combiner l'approche Web et l'architecture multiniveau, comme décrit dans "Ecriture des applications client Web" à la page 25-36.

Analyse des données

Certaines applications de bases de données ne présentent pas les informations de bases de données directement à l'utilisateur mais analysent et résument les informations des bases de données pour permettre aux utilisateurs de tirer des conclusions à partir des données.

Le composant *TDBChart* de la page ContrôleBD de la palette des composants vous permet de présenter les informations de bases de données sous forme graphique afin que les utilisateurs puissent rapidement saisir l'importance des informations de bases de données.

En outre, certaines versions de Delphi proposent une page Decision Cube sur la palette des composants. Elle contient six composants qui vous permettent d'analyser les données et de réaliser des références croisées sur les données lors de la construction d'applications d'aide à la décision. Pour plus d'informations sur l'utilisation des composants Decision Cube, voir chapitre 16, "Utilisation de composants d'aide à la décision".

Si vous souhaitez construire vos propres composants d'affichage de résumés de données en fonction de divers critères de regroupement, vous pouvez utiliser des agrégats maintenus avec un ensemble de données client. Pour plus d'informations sur l'utilisation des agrégats maintenus, voir "Utilisation des agrégats maintenus" à la page 23-13.

Ecriture de rapports

Si vous souhaitez que les utilisateurs puissent imprimer les informations de bases de données émanant des ensembles de données de votre application, vous pouvez utiliser les composants rapport de la page QReport de la palette des composants. L'utilisation de ces composants vous permet de construire visuellement des rapports à bandes pour présenter et résumer les informations contenues dans vos tables de bases de données. Vous pouvez ajouter des résumés aux en-têtes et aux pieds de page de groupe pour analyser les données en fonction de critères de regroupement.

Démarrez un rapport pour votre application en sélectionnant l'icône QuickReport dans la boîte de dialogue Nouveaux éléments. Sélectionnez Fichier | Nouveau dans le menu principal et allez à la page Affaires. Double-cliquez sur l'icône Expert QuickReport pour lancer l'expert.

Remarque Pour un exemple d'utilisation des composants de la page QReport, reportez-vous au programme exemple QuickReport livré avec Delphi.

Chapitre

15

Utilisation de contrôles de données

La page ContrôleBD de la palette des composants offre un ensemble de contrôles orientés données qui représentent les données de champs d'un enregistrement de base de données et qui permettent aux utilisateurs de modifier ces données et de répercuter ces modifications dans la base de données si l'ensemble de données le permet. L'utilisation de contrôles de données sur les fiches de votre application de base de données vous permet de construire l'interface utilisateur de celle-ci de sorte que ces informations soient visibles et accessibles aux utilisateurs.

Les contrôles orientés données que vous ajoutez à votre interface utilisateur dépendent notamment des facteurs suivants :

- Le type de données que vous affichez. Vous avez le choix entre les contrôles qui permettent d'afficher et de modifier du texte brut, ceux conçus pour le texte formaté, ceux destinés aux éléments multimédia ou aux graphiques, etc. Les contrôles qui affichent différents types d'informations sont décrits dans "Affichage d'un seul enregistrement" à la page 15-8.
- L'organisation des informations. Vous pouvez afficher les informations d'un seul enregistrement ou, à l'aide d'une grille, répertorier les données émanant de plusieurs enregistrements. Le paragraphe "Choix de l'organisation des données" à la page 15-8 décrit certaines possibilités.
- Le type d'ensemble de données qui fournit les données aux contrôles. Vous pouvez utiliser des contrôles qui reflètent les limites de l'ensemble de données sous-jacent. Par exemple, l'utilisation d'une grille avec un ensemble de données unidirectionnel ne serait pas fondée car les ensembles de données unidirectionnels ne peuvent fournir qu'un enregistrement à la fois.
- Eventuellement, la façon dont vous souhaitez que les utilisateurs puissent naviguer dans les enregistrements ou les ensembles de données pour y ajouter ou y modifier des données. Vous pouvez ajouter vos propres contrôles ou mécanismes de navigation et de modification ou utiliser un contrôle intégré tel qu'un navigateur de données. Pour plus d'informations sur l'utilisation d'un

navigateur de données, voir “Navigation et manipulation d’enregistrements” à la page 15-33.

Remarque Des contrôles orientés données d’aide à la décision plus complexes sont présentés dans le chapitre 16, “Utilisation de composants d’aide à la décision”.

Quels que soient les contrôles orientés données que vous choisissez d’ajouter à votre interface, ils présentent certaines fonctionnalités communes, décrites ci-après.

Fonctionnalités communes des contrôles de données

Les tâches suivantes sont communes à la plupart des contrôles de données :

- Association d’un contrôle de données à un ensemble de données
- Edition et mise à jour des données
- Activation et désactivation de l’affichage des données
- Rafraîchissement de l’affichage des données
- Activation des événements souris, clavier et timer

Les contrôles orientés données vous permettent d’afficher et d’éditer des champs associés à l’enregistrement en cours d’un ensemble de données. Le tableau suivant dresse la liste des contrôles de données apparaissant sur la page ContrôleBD de la palette des composants.

Tableau 15.1 Contrôles de données

Contrôles de données	Description
TDBGrid	Affiche les informations issues d’une source de données dans un format tabulaire. Les colonnes de la grille correspondent à celles de la table sous-jacente ou de l’ensemble de données de la requête. Chaque ligne de la grille représente un enregistrement.
TDBNavigator	Permet la navigation dans les enregistrements d’un ensemble de données, leur mise à jour, leur validation, leur suppression, l’annulation des opérations d’édition et le rafraîchissement de l’affichage.
TDBText	Affiche les données d’un champ sous forme de libellé.
TDBEdit	Affiche les données d’un champ dans une zone de saisie.
TDBMemo	Affiche les données d’un champ mémo ou d’un champ BLOB dans une zone de saisie multiligne.
TDBImage	Affiche des images graphiques d’un champ de données dans une zone graphique.
TDBListBox	Affiche une liste d’éléments à partir desquels vous pouvez mettre à jour un champ de l’enregistrement en cours.
TDBComboBox	Affiche une liste déroulante d’éléments à partir desquels vous pouvez mettre à jour un champ et permet la saisie directe de texte comme dans une zone de saisie standard.
TDBCheckBox	Affiche une case à cocher indiquant la valeur d’un champ booléen.

Tableau 15.1 Contrôles de données (suite)

Contrôles de données	Description
TDBRadioGroup	Affiche un ensemble d'options mutuellement exclusives pour un champ.
TDBLookupListBox	Affiche une liste d'éléments référencés dans un autre ensemble de données en fonction de la valeur d'un champ.
TDBLookupComboBox	Affiche une liste d'éléments référencés dans un autre ensemble de données en fonction de la valeur d'un champ et permet la saisie directe de texte comme dans une zone de saisie orientée données standard.
TDBCtrlGrid	Affiche un ensemble de contrôles orientés données configurable et récurrent dans une grille.
TDBRichEdit	Affiche les données formatées d'un champ dans une zone de saisie.

Les contrôles sont orientés données au moment de la conception. Lorsque vous associez le contrôle orienté données à un ensemble de données actif lors de la construction d'une application, vous pouvez voir immédiatement les données réelles dans le contrôle. En phase de conception, vous pouvez utiliser l'éditeur de champs pour parcourir un ensemble de données afin de vérifier que votre application affiche les données correctement sans qu'il soit nécessaire de la compiler ou de l'exécuter. Pour plus d'informations sur l'éditeur de champs, voir la section "Création de champs persistants" à la page 19-4.

Lors de l'exécution, les contrôles orientés données affichent également des données et permettent leur édition si le contrôle, l'application et l'ensemble de données auxquels votre application est connectée l'autorisent.

Association d'un contrôle de données à un ensemble de données

Les contrôles de données se connectent aux ensembles de données en utilisant une source de données. Un composant source de données (*TDataSource*) agit comme une voie de communication entre le contrôle et un ensemble de données contenant des données. Chaque contrôle orienté données doit être associé à un composant source de données afin de pouvoir afficher et manipuler des données. De même, tous les ensembles de données doivent être associés à un composant source de données afin que leurs données puissent être affichées et manipulées dans les contrôles orientés données d'une fiche.

Remarque Les composants source de données sont également nécessaires pour lier les ensembles de données non imbriqués dans les relations maître-détail.

Pour associer un contrôle de données à un ensemble de données,

1. Placez un ensemble de données dans un module de données ou sur une fiche et définissez ses propriétés.
2. Placez une source de données dans le même module de données ou sur la même fiche. A l'aide de l'inspecteur d'objets, attribuez à sa propriété *DataSet* l'ensemble de données placé à l'étape 1.

3 Depuis l'onglet AccèsBD de la palette des composants, placez un contrôle de données sur une fiche.

4 A l'aide de l'inspecteur d'objets, attribuez à la propriété *DataSource* du contrôle le composant source de données placé à l'étape 2.

5 Donnez à la propriété *DataField* du contrôle le nom du champ à afficher, ou bien sélectionnez un champ dans la liste déroulante. Cette étape ne s'applique pas aux contrôles *TDBGrid, TDBCtrlGrid* et *TDBNavigator*, car ils accèdent à tous les champs disponibles dans un ensemble de données.

6 Pour afficher des données dans le contrôle, mettez la propriété *Active* de l'ensemble de données à *True*.

Modification de l'ensemble de données associé à l'exécution

Dans l'exemple précédent, la source de données a été associée à son ensemble de données en définissant sa propriété *DataSet* à la conception. A l'exécution, si nécessaire, vous pouvez modifier l'ensemble de données d'un composant source de données. Par exemple, le code suivant attribue à l'ensemble de données du composant source de données *CustSource* l'un des composants ensemble de données *Customers* ou *Orders* :

```
with CustSource do begin
  if (DataSet = Customers) then
    DataSet := Orders
  else
    DataSet := Customers;
end;
```

Vous pouvez également attribuer à la propriété *DataSet* un ensemble de données appartenant à une autre fiche afin de synchroniser les contrôles de données des deux fiches. Par exemple :

```
procedure TForm2.FormCreate (Sender : TObject);
begin
  DataSource1.Dataset := Form1.Table1;
end;
```

Activation et désactivation de la source de données

La source de données possède une propriété *Enabled* qui détermine si elle est connectée à son ensemble de données. Lorsque *Enabled* vaut *True*, la source de données est connectée à un ensemble de données.

Vous pouvez temporairement déconnecter une source de données unique de son ensemble de données en attribuant à *Enabled* la valeur *False*. Lorsque *Enabled* vaut *False*, tous les contrôles de données attachés au composant source de données deviennent vierges et inactifs jusqu'à ce que *Enabled* prenne la valeur *True*. Il est toutefois recommandé de contrôler l'accès à un ensemble de données par le biais des méthodes *DisableControls* et *EnableControls* d'un composant ensemble de données car elles affectent toutes les sources de données attachées.

Réponse aux modifications effectuées par le biais de la source de données

Étant donné que la source de données relie le contrôle de données à son ensemble de données, elle véhicule toute la communication qui intervient entre eux deux. Généralement, le contrôle orienté données répond automatiquement aux modifications apportées dans l'ensemble de données. Toutefois, si votre interface utilisateur utilise des contrôles non orientés données, vous pouvez utiliser les événements d'un composant source de données pour fournir manuellement le même type de réponse.

L'événement *OnDataChange* se produit chaque fois que les données d'un enregistrement sont susceptibles d'avoir évolué, notamment lorsque le contenu des champs est modifié ou que le curseur est positionné sur un autre enregistrement. Cet événement, déclenché par toute modification, garantit que le contrôle reflète les valeurs de champ en cours dans l'ensemble de données. Généralement, un gestionnaire d'événement *OnDataChange* rafraîchit la valeur d'un contrôle non orienté données qui affiche des données de champ.

L'événement *OnUpdateData* se produit lorsque les données de l'enregistrement en cours sont sur le point d'être validées. Par exemple, un événement *OnUpdateData* se produit après l'appel de *Post*, mais avant la validation effective des données dans le cache local ou le serveur de bases de données sous-jacent.

L'événement *OnStateChange* se produit lorsque l'état de l'ensemble de données change. Lorsque cet événement est déclenché, vous pouvez examiner la propriété *State* de l'ensemble de données afin de déterminer son état en cours.

Par exemple, le gestionnaire d'événement *OnStateChange* suivant active ou désactive les boutons ou les éléments de menu en fonction de l'état en cours :

```
procedure Form1.DataSource1.StateChange(Sender: TObject);
begin
  CustTableEditBtn.Enabled := (CustTable.State = dsBrowse);
  CustTableCancelBtn.Enabled := CustTable.State in [dsInsert, dsEdit, dsSetKey];
  CustTableActivateBtn.Enabled := CustTable.State in [dsInactive];
  ⋮
end;
```

Remarque Pour plus d'informations sur les états des ensembles de données, voir "Détermination des états d'un ensemble de données" à la page 18-3.

Edition et mise à jour des données

A l'exception du navigateur, tous les contrôles affichent les données d'un champ de base de données. De plus, vous pouvez les utiliser pour éditer et mettre à jour les données, si l'ensemble de données sous-jacent le permet.

Remarque Les ensembles de données unidirectionnels ne permettent jamais aux utilisateurs d'éditer et de mettre à jour les données.

Activation de l'édition des contrôles lors d'une saisie utilisateur

Un ensemble de données doit être en mode *dsEdit* pour autoriser l'édition de son contenu. Si la propriété *AutoEdit* de la source de données vaut *True* (valeur par défaut), le contrôle de données place l'ensemble de données en mode *dsEdit* dès que l'utilisateur essaie de modifier son contenu.

Si *AutoEdit* vaut *False*, vous devez fournir un mécanisme permettant de placer l'ensemble de données en mode édition. Ce mécanisme peut consister en un contrôle *TDBNavigator* doté d'un bouton *d'édition*, grâce auquel l'utilisateur peut explicitement placer l'ensemble de données en mode édition. Pour plus d'informations sur *TDBNavigator*, voir "Navigation et manipulation d'enregistrements" à la page 15-33. Vous avez également la possibilité d'écrire du code qui appelle la méthode *Edit* de l'ensemble de données lorsque vous souhaitez placer celui-ci en mode édition.

Edition des données affichées dans un contrôle

Un contrôle de données ne peut valider les modifications dans son ensemble de données associé que si la propriété *CanModify* de l'ensemble de données vaut *True*. *CanModify* vaut toujours *False* pour les ensembles de données unidirectionnels. Certains ensembles de données possèdent une propriété *ReadOnly* qui permet de spécifier si *CanModify* vaut *True*.

Remarque La possibilité pour un ensemble de données de mettre à jour des données dépend de la table de base de données sous-jacent.

Même si la propriété *CanModify* de l'ensemble de données vaut *True*, la propriété *Enabled* de la source de données qui connecte l'ensemble de données au contrôle doit également valoir *True* afin que le contrôle puisse valider les mises à jour dans la table de base de données. La propriété *Enabled* de la source de données détermine si le contrôle peut afficher les valeurs de champ à partir de l'ensemble de données et, par conséquent, si un utilisateur peut modifier et valider les valeurs. Si *Enabled* vaut *True* (valeur par défaut), les contrôles peuvent afficher les valeurs de champ.

Enfin, vous pouvez même déterminer si l'utilisateur peut apporter des modifications aux données affichées dans le contrôle. La propriété *ReadOnly* du contrôle de données détermine si un utilisateur peut modifier les données affichées par le contrôle. Si elle vaut *False* (valeur par défaut), les utilisateurs peuvent modifier les données. Il est logique d'appliquer d'office à la propriété *ReadOnly* du contrôle la valeur *True* lorsque la propriété *CanModify* de l'ensemble de données a pour valeur *False*. Sinon, vous donnez aux utilisateurs la fausse impression qu'ils peuvent affecter les données de la table de base de données sous-jacente.

Dans tous les contrôles orientés données, à l'exception de *TDBGrid*, les modifications d'un champ sont copiées dans l'ensemble de données sous-jacent quand vous appuyez sur ***Tab*** depuis le contrôle. Si vous appuyez sur la touche ***Echap*** avant d'appuyer sur ***Tab*** dans un champ, les modifications sont abandonnées et le champ reprend sa valeur initiale.

Avec le composant *TDBGrid*, les modifications sont validées au moment où vous passez à un enregistrement différent ; vous pouvez appuyer sur *Echap* dans n'importe quel champ d'un enregistrement avant de passer à un autre enregistrement pour annuler une modification.

Lorsqu'un enregistrement est écrit, Delphi vérifie si le statut des contrôles orientés données associés à l'ensemble de données a été modifié. En cas de problème de mise à jour d'un champ contenant des données modifiées, Delphi provoque une exception et aucune modification n'est apportée à l'enregistrement.

Remarque si votre application place les mises à jour en mémoire cache (par exemple, à l'aide d'un ensemble de données client), toutes les modifications sont écrites dans un cache interne. Ces modifications ne sont pas appliquées à la table de base de données sous-jacente tant que vous n'appelez pas la méthode *ApplyUpdates* de l'ensemble de données.

Activation et désactivation de l'affichage des données

Lorsque votre application parcourt un ensemble de données ou effectue une recherche, il est préférable d'interdire temporairement le rafraîchissement des valeurs affichées dans les contrôles orientés données à chaque changement d'enregistrement. Cela a pour effet d'accélérer l'itération ou la recherche et permet d'éviter que l'image ne "saute" à l'écran.

DisableControls est une méthode qui désactive l'affichage dans tous les contrôles orientés données liés à un ensemble de données. Dès que l'itération ou la recherche est terminée, votre application doit immédiatement faire appel à la méthode *EnableControls* pour réactiver l'affichage des contrôles.

En règle générale, vous devez désactiver les contrôles avant de commencer à parcourir les enregistrements. Ce processus doit prendre place dans une instruction **try...finally** pour que vous puissiez réactiver les contrôles même si une exception est provoquée. La clause **finally** doit faire appel à *EnableControls*. Le code ci-dessous montre comment *DisableControls* et *EnableControls* peuvent être utilisées dans ce but :

```
CustTable.DisableControls;
try
  CustTable.First; { Aller sur le premier enregistrement qui définit EOF à False }
  while not CustTable.EOF do { Boucle jusqu'à ce qu'EOF soit à True }
  begin
    { Traiter chaque enregistrement ici }
    ⋮
    CustTable.Next; { EOF à False en cas de succès ; EOF à True quand Next échoue sur le
dernier enregistrement }
  end;
finally
  CustTable.EnableControls;
end;
```

Rafraîchissement de l'affichage des données

La méthode *Refresh* d'un ensemble de données réinitialise les tampons locaux et extrait à nouveau les données d'un ensemble ouvert. Vous pouvez utiliser cette méthode pour mettre à jour l'affichage dans les contrôles orientés données si vous pensez que les données sous-jacentes ont changé du fait que d'autres applications y accèdent en même temps. Si vous utilisez les mises à jour en mémoire cache, vous devez, avant d'actualiser l'ensemble de données, appliquer les mises à jour éventuelles que celui-ci détient en mémoire cache.

Le rafraîchissement donne parfois des résultats inattendus (par exemple, lorsqu'un utilisateur est en train de visualiser un enregistrement supprimé par une autre application, puis que cet enregistrement disparaît dès que l'application fait appel à *Refresh*). Les données peuvent également changer à l'affichage si un autre utilisateur modifie un enregistrement après que vous ayez extrait les données et avant d'avoir fait appel à *Refresh*.

Activation des événements souris, clavier et timer

La propriété *Enabled* d'un contrôle de données détermine s'il doit réagir aux événements souris, clavier ou timer et transmettre des informations à sa source de données. La valeur par défaut de cette propriété est *True*.

Pour empêcher les événements souris, clavier ou timer d'atteindre un contrôle de données, vous devez définir sa propriété *Enabled* à *False*. Quand *Enabled* a la valeur *False*, la source de données qui connecte le contrôle à son ensemble de données ne reçoit pas d'informations du contrôle de données. Celui-ci affichera toujours les données, mais le texte n'apparaîtra pas en surbrillance.

Choix de l'organisation des données

Lorsque vous créez l'interface utilisateur de votre application de base de données, vous devez définir l'organisation de l'affichage des informations et des contrôles qui les manipulent.

Vous devez d'abord déterminer si vous souhaitez afficher un seul enregistrement ou plusieurs enregistrements à la fois.

En outre, vous pouvez ajouter des contrôles pour explorer et manipuler les enregistrements. Le contrôle *TDBNavigator* prend en charge de nombreuses fonctions susceptibles de vous intéresser.

Affichage d'un seul enregistrement

Dans de nombreuses applications, il est souhaitable de n'afficher simultanément que les informations relatives à un seul enregistrement de données. Par exemple, une application de saisie de commandes peut afficher les informations relatives à une seule commande sans indiquer les autres commandes consignées. Ces

informations peuvent provenir d'un seul enregistrement d'un ensemble de commandes.

Les applications qui affichent un seul enregistrement sont généralement faciles à lire et à comprendre car toutes les informations de bases de données concernent le même élément (la même commande dans le cas précédent). Les contrôles orientés données contenus dans ces interfaces utilisateur représentent un seul champ d'un enregistrement de base de données. La page ContrôleBD de la palette des composants offre un large choix de contrôles pour la représentation des différents types de champs. Ces contrôles sont généralement les versions orientées données de contrôles disponibles sur la palette des composants. Par exemple, le contrôle *TDBEdit* est une version orientée données du contrôle *TEdit* standard qui permet aux utilisateurs de visualiser et d'éditer une chaîne texte.

Le type de données contenu dans le champ (texte, texte formaté, graphique, informations booléennes, etc.) détermine le contrôle utilisé.

Affichage de données en tant que libellés

TDBText est un contrôle en lecture seulement semblable au composant *TLabel* de la page Standard de la palette des composants. Le contrôle *TDBText* est utile quand vous voulez placer des données en affichage seulement sur une fiche permettant à l'utilisateur d'entrer des données dans d'autres contrôles. Supposons, par exemple, qu'une fiche soit créée à partir des champs d'une table clients et qu'une fois que l'utilisateur entre les informations sur la rue, la ville et le département dans la fiche, vous utilisiez une référence dynamique pour déterminer automatiquement le contenu du champ code postal depuis une table distincte. Un composant *TDBText* relié à la table des codes postaux permettra d'afficher le champ code postal correspondant à l'adresse entrée par l'utilisateur.

TDBText extrait le texte affiché du champ spécifié dans l'enregistrement en cours d'un ensemble de données. Puisque *TDBText* obtient son texte d'un ensemble de données, le texte affiché est dynamique, c'est-à-dire qu'il change au fur et à mesure que l'utilisateur navigue dans la table. Pour cette raison, il n'est pas possible de spécifier le texte d'affichage de *TDBText* au moment de la conception comme c'est le cas pour le composant *TLabel*.

Remarque Lorsque vous placez un composant *TDBText* sur une fiche, vérifiez que sa propriété *AutoSize* est à *True* (la valeur par défaut) ; vous garantissez ainsi que le contrôle se redimensionne automatiquement pour afficher des données de largeur variable. Si *AutoSize* vaut *False*, et le contrôle trop étroit, les données affichées seront tronquées.

Affichage et édition de champs dans une zone de saisie

TDBEdit est la version orientée données du composant zone de saisie. *TDBEdit* affiche la valeur actuelle d'un champ de données auquel il est lié et lui permet d'être édité comme dans n'importe quelle zone de saisie.

Par exemple, supposons que *CustomersSource* est un composant *TDataSource* actif et lié à un objet *TClientDataSet* ouvert appelé *CustomersTable*. Vous pouvez placer un composant *TDBEdit* sur une fiche et définir ses propriétés comme suit :

- *DataSource*: CustomersSource
- *DataField*: CustNo

Le composant zone de saisie orientée données affiche immédiatement la valeur de la colonne *CustNo* de la ligne en cours dans l'ensemble de données *CustomersTable*, tant au moment de la conception que de l'exécution.

Affichage et édition de texte dans un contrôle mémo

TDBMemo est un composant orienté données, semblable au composant *TMemo* standard, permettant d'afficher des données de texte long. *TDBMemo* sert à afficher et saisir du texte multiligne. Vous pouvez utiliser les contrôles *TDBMemo* pour afficher les champs de texte volumineux ou les données textuelles contenues dans les champs BLOB (binary large object).

Par défaut, *TDBMemo* permet à un utilisateur d'éditer du texte mémo. Pour empêcher l'édition, mettez la propriété *ReadOnly* du contrôle mémo à *True*. Pour afficher les tabulations et permettre aux utilisateurs d'entrer des tabulations dans le mémo, mettez la propriété *WantTabs* à *True*. Pour limiter le nombre de caractères pouvant être saisis par les utilisateurs dans un mémo de base de données, utilisez la propriété *MaxLength*. Par défaut, *MaxLength* vaut 0, ce qui signifie qu'il n'y a aucune limite, en dehors de celles du système d'exploitation, au nombre de caractères pouvant être contenus dans le contrôle.

Plusieurs propriétés affectent l'aspect du mémo de base de données et la façon d'entrer le texte. Vous pouvez fournir des barres de défilement à l'aide de la propriété *ScrollBars*. Pour interdire le retour automatique en fin de ligne, définissez la propriété *WordWrap* à *False*. La propriété *Alignment* détermine l'alignement du texte dans le contrôle. Les options possibles sont *taLeftJustify* (valeur par défaut), *taCenter* et *taRightJustify*. Pour changer la fonte du texte, utilisez la propriété *Font*.

Au moment de l'exécution, l'utilisateur peut couper, copier et coller du texte vers un contrôle mémo de base de données ou depuis celui-ci. Vous pouvez également effectuer cette tâche par programmation avec les méthodes *CutToClipboard*, *CopyToClipboard* et *PasteFromClipboard*.

Comme *TDBMemo* peut afficher de grandes quantités de données, ce champ peut être relativement long à remplir lors de l'exécution. Pour accélérer le défilement des enregistrements, *TDBMemo* a une propriété *AutoDisplay* qui contrôle si les données auxquelles il accède doivent être affichées automatiquement. Si vous définissez *AutoDisplay* à *False*, *TDBMemo* affiche le nom du champ plutôt que les données elles-mêmes. Pour visualiser les données, il suffit de double-cliquer à l'intérieur du contrôle.

Affichage et édition dans un contrôle mémo de texte formaté

TDBRichEdit est un composant orienté données, semblable au composant *TRichEdit* standard, qui peut afficher le texte formaté d'un BLOB (binary large object). *TDBRichEdit* affiche du texte formaté, multiligne, et permet à l'utilisateur d'entrer du texte multiligne formaté.

Remarque *TDBRichEdit* est doté de propriétés et de méthodes permettant d'entrer et de manipuler du texte formaté. Toutefois, il n'offre pas une interface utilisateur mettant ces options de formatage à la disposition de l'utilisateur. C'est votre application qui doit implémenter l'interface utilisateur permettant d'accéder aux fonctions de texte formaté.

Par défaut, *TDBRichEdit* permet à l'utilisateur d'éditer du texte mémo formaté. Pour empêcher l'édition, mettez la propriété *ReadOnly* du contrôle éditeur de texte formaté à *True*. Pour afficher les tabulations et permettre aux utilisateurs d'entrer des tabulations dans le mémo, mettez la propriété *WantTabs* à *True*. Pour limiter le nombre de caractères pouvant être saisis par les utilisateurs dans un mémo de base de données, utilisez la propriété *MaxLength*. Par défaut, MaxLength vaut 0, ce qui signifie qu'il n'y a aucune limite, en dehors de celles du système d'exploitation, au nombre de caractères pouvant être saisi.

Comme *TDBRichEdit* peut contenir de grands volumes de données, il faut parfois beaucoup de temps pour afficher les données à l'exécution. Pour accélérer le défilement des enregistrements, *TDBRichEdit* a une propriété *AutoDisplay* qui contrôle si les données auxquelles il accède doivent être affichées automatiquement. Si *AutoDisplay* est à *False*, *TDBRichEdit* affiche le nom du champ à la place des données. Pour visualiser les données, il suffit de double-cliquer à l'intérieur du contrôle.

Affichage et édition de champs graphiques dans un contrôle image

TDBImage est un contrôle orienté données qui permet l'affichage de graphiques contenus dans des champs BLOB.

Par défaut, *TDBImage* permet à l'utilisateur d'éditer une image par couper-coller via le Presse-papiers en utilisant les méthodes *CutToClipboard*, *CopyToClipboard* et *PasteFromClipboard*. Vous pouvez également fournir vos propres méthodes d'édition si vous le souhaitez.

Par défaut, seule la partie du graphique pouvant tenir dans le contrôle image apparaît et l'image est tronquée si elle est trop grande. Vous pouvez attribuer à la propriété *Stretch* la valeur *True* pour redimensionner le graphique afin qu'il remplisse le contrôle image et que sa taille soit ajustée conformément aux redimensionnements du contrôle.

Comme le composant *TDBImage* peut afficher de grandes quantités de données, l'affichage peut prendre un certain temps à l'exécution. Pour accélérer le défilement des enregistrements, *TDBImage* a une propriété *AutoDisplay* qui contrôle si les données auxquelles il accède doivent s'afficher automatiquement. Si vous mettez *AutoDisplay* à *False*, *TDBImage* affiche le nom du champ au lieu des données. Pour visualiser les données, il suffit de double-cliquer à l'intérieur du contrôle.

Affichage de données dans des boîtes liste et des boîtes à options

Quatre contrôles de données fournissent aux utilisateurs un groupe de valeurs par défaut parmi lesquelles il est possible d'effectuer un choix à l'exécution. Ces contrôles présentent une version orientée données des contrôles boîte liste et boîte à options standard :

- *TDBListBox*. Ce contrôle affiche une liste d'éléments qu'il est possible de faire défiler. L'utilisateur peut aussi choisir l'un de ces éléments pour entrer dans un champ de données. Une boîte liste orientée données affiche la valeur en cours pour un champ de l'enregistrement en cours et met l'entrée correspondante en surbrillance dans la liste. Si la valeur de champ de la ligne en cours ne figure pas dans la liste, aucune valeur n'est mise en surbrillance dans la boîte liste. Lorsqu'il sélectionne un élément dans la liste, la valeur de champ correspondante est remplacée dans l'ensemble de données sous-jacent.
- *TDBComboBox*. Ce contrôle combine les fonctionnalités d'un contrôle de saisie orienté données et d'une liste déroulante. A l'exécution, les utilisateurs ont la possibilité d'afficher une liste déroulante dans laquelle ils peuvent sélectionner un ensemble de valeurs prédéfinies ou de saisir une valeur différente.
- *TDBLookupListBox*. Ce composant se comporte comme *TDBListBox* à la différence que la liste des éléments d'affichage est recherchée dans un autre ensemble de données.
- *TDBLookupComboBox*. Ce composant se comporte comme *TDBComboBox* à la différence que la liste des éléments d'affichage est recherchée dans un autre ensemble de données.

Remarque Au moment de l'exécution, les utilisateurs peuvent lancer une recherche incrémentale pour trouver des éléments dans une boîte liste. Lorsque le contrôle a la focalisation, le fait qu'un utilisateur tape 'ROB' par exemple, provoque la sélection du premier élément de la boîte liste commençant par ces lettres. S'il ajoute un 'E', le premier élément commençant par les lettres ROBE est sélectionné, par exemple 'Robert Johnson'. La recherche n'effectue pas de distinction majuscules/minuscules. Les touches *Retour arrière* et *Echap* permettent d'annuler la recherche de la chaîne (mais laissent la sélection intacte), à l'instar d'une pause de deux secondes entre deux frappes.

Utilisation de TDBListBox et de TDBComboBox

Lorsque vous utilisez *TDBListBox* ou *TDBComboBox*, vous devez recourir à l'éditeur de liste de chaînes lors de la conception afin de créer la liste des éléments à afficher. Pour activer l'éditeur de liste de chaînes, cliquez sur le bouton à points de suspension situé en regard de la propriété *Items* dans l'inspecteur d'objets. Ensuite, tapez les éléments devant apparaître dans la liste. A l'exécution, utilisez les méthodes de la propriété *Items* pour manipuler sa liste de chaînes.

Quand un contrôle *TDBListBox* ou *TDBComboBox* est lié à un champ par sa propriété *DataField*, la valeur du champ est sélectionnée dans la liste. Si la valeur en cours ne figure pas dans la liste, aucun élément n'est sélectionné. Toutefois, *TDBComboBox* affiche la valeur en cours du champ dans sa zone de saisie, qu'elle apparaisse ou non dans la liste *Items*.

Pour *TDBListBox*, la propriété *Height* détermine le nombre d'éléments simultanément visibles dans la boîte liste. La propriété *IntegralHeight* contrôle l'affichage du dernier élément. Si *IntegralHeight* vaut *False* (valeur par défaut), la base de la boîte liste est déterminée par la propriété *ItemHeight* et le dernier élément est susceptible de n'apparaître que partiellement. Si *IntegralHeight* a pour valeur *True*, le dernier élément visible de la boîte liste s'affiche entièrement.

Pour *TDBComboBox*, la propriété *Style* détermine l'interaction de l'utilisateur avec le contrôle. Par défaut, *Style* est à *csDropDown*, ce qui signifie que l'utilisateur peut entrer des valeurs au clavier ou bien choisir un élément dans la liste déroulante. Les propriétés suivantes déterminent l'affichage de la liste *Items* à l'exécution :

- *Style* détermine le style d'affichage du composante la boîte à options :
  - *csDropDown* (valeur par défaut) : affiche une liste déroulante avec une zone de saisie où l'utilisateur peut entrer du texte. Tous les éléments sont des chaînes de même hauteur.
  - *csSimple* : combine un contrôle de saisie à une liste d'éléments de taille fixe systématiquement affichée. Quand *Style* a la valeur *csSimple*, assurez-vous d'augmenter la propriété *Height* afin de pouvoir afficher la liste.
  - *csDropDownList* : affiche une boîte liste et une zone de saisie, mais l'utilisateur ne peut ni saisir ni modifier les valeurs qui ne figurent pas dans la liste déroulante au moment de l'exécution.
  - *csOwnerDrawFixed* et *csOwnerDrawVariable* : permettent à la liste des éléments d'afficher des valeurs autres que des chaînes (par exemple, des images bitmap) ou d'utiliser des fontes différentes pour les éléments de la liste.
- *DropDownCount* : nombre maximum d'éléments affichés dans la liste. Si le nombre d'éléments (*Items*) est supérieur à *DropDownCount*, l'utilisateur peut faire défiler la liste. Si le nombre d'éléments (*Items*) est inférieur à *DropDownCount*, la liste est suffisamment grande pour afficher tous les éléments (Items).
- *ItemHeight* : hauteur de chaque élément lorsque le style a pour valeur *csOwnerDrawFixed*.
- *Sorted* : si la valeur est *True*, la liste des éléments (*Items*) est affichée dans l'ordre alphabétique.

Affichage dans une boîte liste de référence et une boîte à options de référence

Les boîtes liste de référence et les boîtes à options de référence (*TDBLookupListBox* et *TDBLookupComboBox*) proposent à l'utilisateur une liste limitée de choix à partir desquels il peut définir une valeur de champ correcte. Lorsqu'il sélectionne un élément dans la liste, la valeur de champ correspondante est remplacée dans l'ensemble de données sous-jacent.

Prenons l'exemple d'un bon de commande dont les champs sont liés à *OrdersTable*. *OrdersTable* contient un champ *CustNo* correspondant à un numéro d'ID client, mais *OrdersTable* ne contient aucune autre information sur le client.

CustomersTable, en revanche, contient un champ *CustNo* correspondant à un numéro d'ID client ainsi que des informations supplémentaires, telles que l'entreprise et l'adresse du client. Lors de la facturation, il serait pratique que le bon de commande permette à un employé de sélectionner un client d'après le nom de l'entreprise au lieu de l'ID client. Un composant *TDBLookupListBox* affichant le nom de toutes les entreprises dans *CustomersTable* permettra à un utilisateur de sélectionner le nom de l'entreprise dans la liste et de placer *CustNo* sur le bon de commande.

Ces contrôles de référence dérivent la liste d'éléments d'affichage depuis l'une des deux sources suivantes :

- **Un champ de référence défini pour un ensemble de données.** Pour spécifier les éléments d'une boîte liste à l'aide d'un champ de référence, l'ensemble de données auquel vous liez le contrôle doit déjà définir un champ de référence. (Ce processus est décrit dans "Définition d'un champ de référence" à la page 19-10). Pour spécifier le champ de référence pour les éléments d'une boîte liste,

  1 Affectez à la propriété *DataSource* de la boîte liste la source de données de l'ensemble de données contenant le champ de référence à utiliser.

  2 Choisissez le champ de référence à utiliser dans la liste déroulante de la propriété *DataField*.

  Lorsque vous activez une table associée à un contrôle de référence, il identifie son champ de données en tant que champ de référence et affiche les valeurs appropriées.

- **Une clé, un champ de données et une source de données secondaires**. Si vous n'avez pas défini de champ de référence pour un ensemble de données, vous pouvez établir une relation de même type à l'aide d'une source de données secondaire, d'une valeur de champ à rechercher dans la source de données secondaire et d'une valeur de champ à renvoyer comme élément de liste. Pour spécifier une source de données secondaire pour les éléments d'une boîte liste,

  3 Affectez à la propriété *DataSource* de la boîte liste la source de données du contrôle.

  4 Choisissez un champ pour insérer les valeurs de référence depuis la liste déroulante de la propriété *DataField.* Le champ choisi ne doit pas être un champ de référence.

  5 Affectez à la propriété *ListSource* de la boîte liste la source de l'ensemble de données contenant le champ dont vous voulez référencer les valeurs.

  6 Pour la propriété *KeyField*, choisissez dans la liste déroulante un champ à utiliser comme clé de référence. La liste déroulante affiche les champs de l'ensemble de données associé à la source de données spécifiée à l'étape 3. Le champ choisi n'a pas besoin de faire partie d'un index, mais si c'est le cas, les performances en seront améliorées.

  7 Dans la liste déroulante de la propriété *ListField*, choisissez un champ dont les valeurs sont à renvoyer. Cette liste déroulante affiche les champs de

l'ensemble de données associé à la source que vous avez spécifiée à l'étape 3.

Lorsque vous activez une table associée à un contrôle de référence, il reconnaît que les éléments de sa liste sont dérivés d'une source secondaire et affiche des valeurs issues de celle-ci.

Pour définir le nombre d'éléments simultanément affichés dans un contrôle *TDBLookupListBox*, utilisez la propriété *RowCount*. La hauteur de la boîte liste s'adapte au nombre de lignes défini.

Pour définir le nombre d'éléments affichés dans la liste déroulante de *TDBLookupComboBox*, utilisez la propriété *DropDownRows*.

Remarque Vous pouvez également définir une colonne de grille de données pour qu'elle fonctionne comme une boîte à options de référence. Pour plus d'informations sur la manière de procéder, voir "Définition d'une colonne de liste de référence" à la page 15-24.

Manipulation de champs booléens avec des cases à cocher

TDBCheckBox est un contrôle case à cocher orientée données. Vous pouvez l'utiliser pour définir les valeurs de champs booléens dans un ensemble de données. Par exemple, un formulaire de facture peut comporter une case qui, quand elle est cochée, indique que le client est exonéré d'impôt et, quand elle ne l'est pas, indique qu'il est imposable.

Le contrôle case à cocher orienté données gère son propre état (activation ou désactivation) en comparant le contenu du champ au contenu des propriétés*ValueChecked* et *ValueUnchecked*. Si la valeur de la propriété *ValueChecked* correspond à la valeur du champ, le contrôle est activé. Si la valeur de la propriété *ValueUnchecked*, correspond à la valeur du champ, le contrôle est désactivé.

Remarque Les valeurs des propriétés *ValueChecked* et *ValueUnchecked* ne peuvent pas être identiques.

Donnez à la propriété *ValueChecked* une valeur que le contrôle doit écrire dans la base de données s'il est activé lorsque l'utilisateur passe à un autre enregistrement. Par défaut, cette valeur est à "true", mais vous pouvez la changer en valeur alphanumérique en fonction de vos besoins. Vous pouvez également entrer une liste d'éléments délimités par des points-virgules comme valeur de *ValueChecked*. Si l'un de ces éléments correspond au contenu de ce champ dans l'enregistrement en cours, la case est activée. Par exemple, vous pouvez spécifier une chaîne *ValueChecked* comme :

```
DBCheckBox1.ValueChecked := 'True;Yes;On';
```

Si le champ de l'enregistrement en cours contient des valeurs "true", "Yes" ou "On", c'est que la case est activée. Aucune distinction majuscules/minuscules n'est effectuée pendant la comparaison du champ avec les chaînes de *ValueChecked*. Si un utilisateur active une case à cocher pour laquelle il y a plusieurs chaînes *ValueChecked*, la première est la valeur écrite dans la base de données.

Donnez à la propriété *ValueUnchecked* une valeur que le contrôle doit écrire dans la base de données s'il n'est pas activé lorsque l'utilisateur passe à un autre enregistrement. Par défaut, cette valeur est définie comme "false", mais vous pouvez lui donner n'importe quelle valeur alphanumérique en fonction de vos besoins. Vous pouvez également entrer une liste d'éléments délimités par des points-virgules en tant que valeur de *ValueUnchecked*. Si l'un des éléments correspond au contenu de ce champ dans l'enregistrement en cours, la case à cocher n'est pas activée.

Une case à cocher orientée données est désactivée à chaque fois que le champ de l'enregistrement en cours ne contient pas l'une des valeurs de la liste des propriétés *ValueChecked* ou *ValueUnchecked*.

Si le champ associé à la case à cocher est un champ logique, elle est toujours activée lorsque le contenu du champ est à *True* et désactivée s'il est à *False*. Dans ce cas, les chaînes entrées dans les propriétés *ValueChecked* et *ValueUnchecked* sont sans effet sur les champs logiques.

Limitation de valeurs de champ avec des boutons radio

TDBRadioGroup est une version orientée données du composant groupe de boutons radio. Il vous permet de définir la valeur d'un champ de données à l'aide d'un bouton radio où le nombre de valeurs possibles est limité. Le groupe de boutons radio présente un bouton radio pour chaque valeur qu'un champ peut accepter. Les utilisateurs peuvent définir la valeur d'un champ en sélectionnant le bouton radio voulu.

La propriété *Items*détermine les boutons radio devant figurer dans le groupe. *Items* est une liste de chaînes. Un bouton radio est affiché pour chaque chaîne de la propriété *Items* et chaque chaîne apparaît à droite du bouton radio en tant que libellé.

Si la valeur actuelle d'un champ associé à un groupe de boutons radio correspond à l'une des chaînes de la propriété *Items*, ce bouton radio est sélectionné. Par exemple, si trois chaînes, "Rouge", "Jaune" et "Bleu" apparaissent dans la liste *Items* et si le champ de l'enregistrement en cours contient la valeur "Bleu", le troisième bouton du groupe est sélectionné.

Remarque Si le champ ne correspond à aucune des chaînes de la liste *Items*, il est toujours possible de sélectionner un bouton radio lorsque le champ correspond à une chaîne de la propriété *Values*. Si le champ de l'enregistrement en cours ne correspond à aucune des chaînes d'*Items* ou de *Values*, aucun bouton radio n'est sélectionné.

La propriété *Values* peut contenir une liste facultative de chaînes qu'il est possible de renvoyer à l'ensemble de données quand l'utilisateur sélectionne un bouton radio et écrit un enregistrement dans la base de données. Les chaînes sont associées aux boutons dans un ordre numérique. La première chaîne est associée au premier bouton, la seconde au second bouton et ainsi de suite. Par exemple, supposons que la propriété *Items* contienne "Rouge", "Jaune" et "Bleu" et que *Values* contienne "Magenta", "Jaune" et "Cyan". Si un utilisateur sélectionne le bouton libellé "Rouge", "Magenta" est écrit dans la base de données.

Si aucune chaîne n'a été fournie pour *Values*, la chaîne *Item* d'un bouton radio sélectionné est renvoyée à la base de données au moment où un enregistrement y est écrit.

Affichage de plusieurs enregistrements

Il se peut que vous souhaitiez afficher de nombreux enregistrements dans la même fiche. Par exemple, une application de facturation peut afficher sur la même fiche toutes les commandes passées par un même client.

Pour afficher plusieurs enregistrements, utilisez un contrôle grille. Les contrôles grille offrent un affichage à champs et enregistrements multiples des données qui peut rendre l'interface utilisateur de votre application plus attrayante et plus efficace. Les contrôles grille sont présentés dans "Visualisation et édition des données avec un contrôle TDBGrid" à la page 15-18 et "Création d'une grille qui contient d'autres contrôles orientés données" à la page 15-32.

Remarque Vous ne pouvez pas afficher plusieurs enregistrements lorsque vous utilisez un ensemble de données unidirectionnel.

Vous pouvez concevoir une interface utilisateur qui affiche à la fois les champs d'un même enregistrement et les grilles qui représentent plusieurs enregistrements. Voici deux modèles qui combinent ces deux approches :

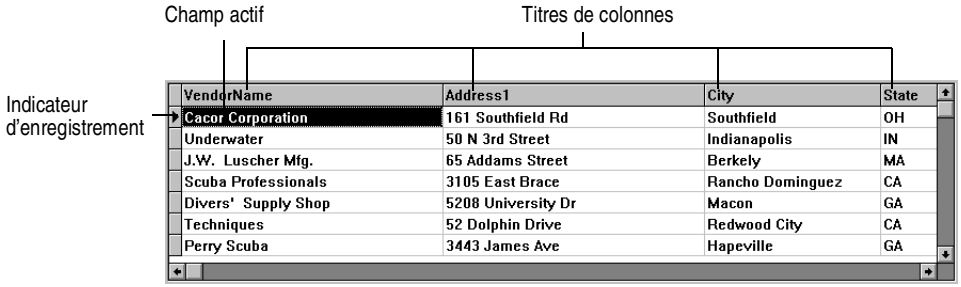
- **Fiches maître-détail :** Vous pouvez représenter les informations à la fois d'une table maître et d'une table détail en incluant à la fois les contrôles qui affichent un champ unique et les contrôles grille. Par exemple, vous pouvez afficher les informations sur un client unique et une grille détail qui affiche les commandes passées par ce client. Pour plus d'informations sur la liaison de tables sous-jacentes dans une fiche maître-détail, voir "Création de relations maître/détail" à la page 18-40 et "Etablissement de relations maître/détail en utilisant des paramètres" à la page 18-55.
- **Fiches détail** : dans une fiche qui affiche plusieurs enregistrements, vous pouvez inclure des contrôles champ unique qui affichent les informations détaillées de l'enregistrement sélectionné. Cette approche est particulièrement utile lorsque les enregistrements incluent de long mémos ou des informations graphiques. Lorsque l'utilisateur passe en revue les enregistrements de la grille, le mémo ou le graphique se met à jour pour représenter la valeur de l'enregistrement sélectionné. La mise en place de ce dispositif est très facile. La synchronisation entre les deux affichages est automatique si la grille et le contrôle mémo ou image partagent une source de données commune.

Astuce Il est généralement préférable de ne pas combiner ces deux approches sur une même fiche. Il est généralement délicat pour les utilisateurs de comprendre les relations entre les données de telles fiches.

Visualisation et édition des données avec un contrôle TDBGrid

Un contrôle *TDBGrid* permet de visualiser et de modifier les enregistrements d'un ensemble de données dans un format de type grille tabulaire.

Figure 15.1 Contrôle TDBGrid



Trois facteurs affectent l'aspect des enregistrements affichés dans un contrôle grille :

- L'existence d'objets colonne persistante définis pour la grille à l'aide de l'éditeur de colonnes. Les objets colonne persistante offrent une grande flexibilité pour configurer des grilles et définir l'aspect des données. Pour plus d'informations sur l'utilisation de colonnes persistantes, voir "Création d'une grille personnalisée" à la page 15-19.
- La création de composants champs persistants pour l'ensemble de données affiché dans la grille. Pour plus d'informations sur la création de ces composants avec l'éditeur de champs, voir chapitre 19, "Manipulation des composants champ".
- La définition du paramètre *ObjectView* de l'ensemble de données pour l'affichage des champs ADT et tableau dans la grille. Voir "Affichage des champs ADT et tableau" à la page 15-26.

Le contrôle grille dispose d'une propriété *Columns* qui est l'enveloppe d'un objet *TDBGridColumns*. *TDBGridColumns* contient une collection d'objets *TColumn* représentant toutes les colonnes du contrôle grille. Vous pouvez utiliser l'éditeur de colonnes pour configurer les attributs de colonnes lors de la phase de conception, ou bien utiliser la propriété *Columns* pour accéder aux méthodes, événements et propriétés de *TDBGridColumns* à l'exécution.

Utilisation d'un contrôle grille à son état par défaut

La propriété *State* de la propriété *Columns* d'une grille indique si des objets colonne persistante existent. *Columns.State* est une propriété d'exécution et est automatiquement définie pour la grille. Sa valeur par défaut est *csDefault*, ce qui signifie que les objets colonne persistante n'existent pas. Dans ce cas, l'affichage des données dans la grille est essentiellement déterminé par les propriétés des

champs de l'ensemble de données de la grille ou, en l'absence de composants champs persistants, par les valeurs d'affichage par défaut.

Lorsque la propriété *Columns.State* de la grille vaut *csDefault*, les colonnes des grilles sont générées dynamiquement à partir des champs visibles de l'ensemble de données, et leur ordre dans la grille correspond à celui des champs dans l'ensemble de données. Chaque colonne de la grille est associée à un composant champ. Les modifications des propriétés des composants champ sont immédiatement reflétées dans la grille.

Il peut être utile d'utiliser un contrôle grille ayant des colonnes dynamiquement générées pour afficher et modifier le contenu de certaines tables sélectionnées à l'exécution. La structure de la grille n'étant pas définie, elle peut changer en temps réel pour afficher différents ensembles de données. Une grille unique avec des colonnes dynamiquement générées peut afficher une table Paradox à un certain moment, puis les résultats d'une requête SQL lorsque sa propriété *DataSource* est modifiée, ou lorsque la propriété *DataSet* de la source des données est modifiée.

L'apparence d'une colonne dynamique peut être modifiée lors de la conception ou à l'exécution ; dans les deux cas, ce sont les propriétés correspondantes du composant champ affiché dans la colonne que vous modifiez. La durée des propriétés des colonnes dynamiques correspond à la période où une colonne est associée à un champ particulier dans un ensemble de données unique. Par exemple, la modification de la propriété *Width* d'une colonne change la propriété *DisplayWidth* du champ associé à cette colonne. Les modifications de propriétés de colonnes non basées sur des propriétés de champs, telles que la propriété *Font*, ne sont valides que pendant l'existence de ces colonnes.

Si l'ensemble de données d'une grille consiste en un ensemble de composants champs dynamiques, les champs sont détruits dès la fermeture de l'ensemble de données. Lorsque les composants champs sont détruits, les colonnes qui leur sont associées sont également détruites. Si l'ensemble de données d'une grille consiste en un ensemble de composants champs persistants, les composants champs existent même si l'ensemble de données est fermé, de telle sorte que toutes les colonnes associées à ces champs gardent leurs propriétés à la fermeture de l'ensemble de données.

Remarque Si la propriété *Columns.State* d'une grille est mise à *csDefault* à l'exécution, tous les objets colonnes de la grille sont supprimés (même les colonnes persistantes) et les colonnes dynamiques sont reconstruites à partir des champs visibles dans l'ensemble de données de la grille.

Création d'une grille personnalisée

Une grille personnalisée est un contrôle pour lequel vous définissez des objets colonne persistante décrivant l'aspect d'une colonne et la méthode d'affichage des données dans la colonne. Une grille personnalisée vous permet de configurer plusieurs grilles pour qu'elles affichent diverses vues d'un même ensemble de données (divers ordres de colonnes, divers choix de champs et diverses couleurs et fontes, par exemple). Une grille personnalisée permet aussi aux utilisateurs de

modifier l'aspect de la grille à l'exécution sans affecter les champs utilisés par la grille, ou l'ordre des champs de l'ensemble de données.

L'utilisation des grilles personnalisées est conseillée avec les ensembles de données dont la structure est connue pendant la phase de conception. Comme elles s'attendent à ce que les noms des champs définis à la conception existent dans l'ensemble de données, les grilles personnalisées ne sont pas adaptées si vous devez utiliser des tables arbitraires sélectionnées à l'exécution.

Présentation des colonnes persistantes

Lorsque vous créez des objets colonne persistante pour une grille, ils ne sont associés aux champs sous-jacents de l'ensemble de données de la grille que de façon temporaire. Les valeurs par défaut des propriétés des colonnes persistantes sont lues dynamiquement dans la source par défaut (la grille ou le champ associé, par exemple) jusqu'à ce qu'une valeur soit affectée à la propriété de la colonne. Tant que vous n'avez pas affecté une valeur à une propriété de colonne, sa valeur change si sa source par défaut change. Dès que vous affectez une valeur à une propriété de colonne, elle ne change plus même si sa source par défaut change.

Par exemple, la source par défaut d'un libellé de colonne est la propriété *DisplayLabel* du champ associé. Si vous modifiez la propriété *DisplayLabel*, le titre de colonne reflète immédiatement ce changement. Si vous affectez ensuite une chaîne au libellé de la colonne, le titre de la colonne devient indépendant de la propriété *DisplayLabel* du champ associé. Les modifications de la propriété *DisplayLabel* du champ ne sont plus reportées dans le titre de la colonne.

Les colonnes persistantes sont indépendantes des composants champ qui leur sont associés. De plus, il n'est pas nécessaire d'associer les colonnes persistantes à des objets champ. Si la propriété *FieldName* d'une colonne persistante est vide, ou si le nom du champ ne correspond à aucun champ dans l'ensemble de données actif dans la grille, la propriété *Field* de la colonne est NULL et la colonne est dessinée avec des cellules vides. Si vous surchargez la méthode de dessin par défaut de la cellule, vous pouvez afficher vos propres informations dans les cellules vides. Par exemple, vous pouvez utiliser une colonne vide pour afficher des valeurs synthétisées sur le dernier enregistrement d'un groupe d'enregistrements récapitulé par l'agrégat. Vous pouvez aussi afficher un bitmap ou un histogramme représentant les données de l'enregistrement.

Il est possible d'associer une ou plusieurs colonnes persistantes au même champ d'un ensemble de données. Par exemple, vous pouvez afficher un champ contenant une référence d'article à droite et à gauche d'une large grille pour faciliter la recherche d'une référence en évitant à l'utilisateur de faire défiler la grille.

Remarque Etant donné que les colonnes persistantes n'ont pas besoin d'être associées à un champ d'un ensemble de données, et étant donné que plusieurs colonnes peuvent référencer le même champ, la valeur de la propriété *FieldCount* d'une grille personnalisée peut être inférieure ou égale au nombre de colonnes d'une grille. Notez également que si la colonne sélectionnée dans la grille personnalisée

n'est pas associée à un champ, la propriété *SelectedField* de la grille est NULL et la propriété *SelectedIndex* est à –1.

Les colonnes persistantes peuvent être configurées pour afficher des cellules de grille sous forme de liste déroulante dans une boîte à options de valeurs de référence provenant d'un autre ensemble de données ou d'une liste de choix statique, ou sous forme d'un bouton à points de suspension (…) dans une cellule, sur laquelle l'utilisateur peut cliquer pour lancer des visionneurs de données spéciaux ou des boîtes de dialogue en relation avec la cellule en cours.

Création de colonnes persistantes

Pour personnaliser l'aspect d'une grille lors de la phase de conception, appelez l'éditeur de colonnes pour créer un ensemble d'objets colonne persistante pour la grille. La propriété *State* d'une grille comportant des objets colonne persistante est automatiquement mise à *csCustomized*.

Pour créer des colonnes persistantes pour un contrôle de grille,

1. Sélectionnez le composant grille dans la fiche.
2. Appelez l'éditeur de colonnes en double-cliquant sur la propriété *Columns* de la grille dans l'inspecteur d'objets.

La boîte liste des colonnes affiche les colonnes persistantes définies pour la grille sélectionnée. Lorsque vous ouvrez l'éditeur de colonnes pour la première fois, cette liste est vide car la grille est dans son mode par défaut et ne contient que des colonnes dynamiques.

Vous pouvez créer des colonnes pour tous les champs d'un ensemble de données en une seule fois, ou vous pouvez créer des colonnes persistantes sur une base individuelle. Pour créer des colonnes persistantes pour tous les champs :

1. Cliquez avec le bouton droit de la souris pour appeler le menu contextuel et choisissez la commande Ajouter tous les champs. Si la grille n'est associée à aucune source de données, Ajouter tous les champs est désactivée. Associez la grille avec une source de données ayant un ensemble de données actif avant de choisir cette commande.
2. Si la grille contient déjà des colonnes persistantes, une boîte de dialogue vous demande si vous souhaitez supprimer les colonnes existantes. Si vous répondez Oui, toutes les informations existantes sur les champs persistants sont supprimées et les champs de l'ensemble de données en cours sont insérés selon leur ordre dans l'ensemble de données. Si vous répondez Non, les champs persistants existants restent intacts, et les nouvelles informations de colonnes basées sur les champs supplémentaires de l'ensemble de données sont ajoutées à l'ensemble de données.
3. Cliquez sur la case de fermeture pour appliquer les colonnes persistantes à la grille et fermer la boîte de dialogue.

Pour créer des colonnes persistantes individuelles :

1. Choisissez le bouton Ajouter dans l'éditeur de colonnes. La nouvelle colonne sera sélectionnée dans la boîte liste. Elle porte un numéro et un nom par défaut (comme, 0 - TColumn).
2. Pour associer un champ à cette nouvelle colonne, définissez la propriété *FieldName* dans l'inspecteur d'objets.
3. Pour définir le titre de la nouvelle colonne, développez la propriété *Title* dans l'inspecteur d'objets et définissez son option *Caption*.
4. Fermez l'éditeur de colonnes pour appliquer les colonnes persistantes à la grille et fermer la boîte de dialogue.

Lors de l'exécution, vous pouvez passer aux colonnes persistantes en attribuant *csCustomized* à la propriété *Columns.State*. Toutes les colonnes existantes de la grille sont détruites et de nouvelles colonnes persistantes sont construites pour chaque champ de l'ensemble de données de la grille. Vous pouvez ensuite ajouter une colonne persistante lors de l'exécution en appelant la méthode *Add* de la liste de colonnes :

```
DBGrid1.Columns.Add;
```

Suppression de colonnes persistantes

Il peut être utile de supprimer une colonne persistante d'une grille pour éliminer les champs que vous ne souhaitez pas afficher. Pour supprimer une colonne persistante :

1. Double-cliquez sur la grille afin d'afficher l'éditeur de colonnes.
2. Sélectionnez le champ à supprimer dans la boîte liste des colonnes.
3. Cliquez sur Supprimer (vous pouvez aussi utiliser le menu contextuel ou la touche *Suppr*).

Remarque Si vous supprimez toutes les colonnes d'une grille, la propriété *Columns.State* revient à l'état *csDefault* et crée automatiquement des colonnes dynamiques pour chaque champ de l'ensemble de données.

Pour supprimer une colonne persistante lors de l'exécution, il vous suffit de libérer l'objet colonne :

```
DBGrid1.Columns[5].Free;
```

Modification de l'ordre des colonnes persistantes

Les colonnes apparaissent dans l'éditeur de colonnes dans le même ordre que dans la grille. Vous pouvez modifier l'ordre des colonnes en faisant un glisser-déplacer depuis la boîte liste des colonnes.

Pour modifier la position d'une colonne :

1. Sélectionnez la colonne dans la boîte liste des colonnes.
2. Faites-la glisser vers son nouvel emplacement dans la boîte liste.

Vous pouvez aussi modifier l'ordre des colonnes en cliquant sur le titre d'une colonne dans la grille elle-même puis en faisant glisser la colonne jusqu'à une nouvelle position, comme vous le feriez à l'exécution.

Remarque Le déplacement des champs persistants dans l'éditeur de champs a pour effet de déplacer les colonnes dans une grille par défaut mais pas dans une grille personnalisée.

Important Vous ne pouvez pas, en phase de conception, déplacer des colonnes dans les grilles contenant des colonnes et des champs dynamiques, car aucun élément persistant ne peut enregistrer la modification.

L'utilisateur peut, à l'exécution, utiliser la souris pour faire glisser une colonne à un nouvel emplacement de la grille si sa propriété *DragMode*vaut *dmManual*. Le changement de l'ordre des colonnes d'une grille pour lesquelles la propriété *State* vaut *csDefault*, provoque également la modification de l'ordre des composants champs dans l'ensemble de données sous-jacent. L'ordre des champs dans la table elle-même n'est pas affecté. Pour empêcher le changement de l'ordre des colonnes à l'exécution, il faut mettre la propriété *DragMode* à *dmAutomatic*.

A l'exécution, l'événement *OnColumnMoved* d'une grille se déclenche après le déplacement d'une colonne.

Définition des propriétés de colonne en mode conception

Les propriétés d'une colonne déterminent l'affichage des données dans ses cellules. La plupart des propriétés de colonnes héritent leurs valeurs par défaut des propriétés associées à un autre composant (appelé la *source par défaut*), tel qu'une grille ou un composant champ associé.

Pour définir les propriétés d'une colonne, sélectionnez la colonne dans l'éditeur de colonnes et définissez ses propriétés dans l'inspecteur d'objets. Le tableau suivant dresse la liste des propriétés de colonne pouvant être définies.

Tableau 15.2 Propriétés de colonne

Propriété	Utilisation
Alignment	Aligne (à gauche ou à droite) ou centre les données du champ dans la colonne. Source par défaut : *TField.Alignment*.
ButtonStyle	*cbsAuto* : (valeur par défaut) affiche une liste déroulante si le champ associé est un champ de référence, ou si la propriété *PickList* de la colonne contient des données. *cbsEllipsis* : affiche un bouton à points de suspension (...) à droite de la cellule. Un clic sur ce bouton déclenche l'événement *OnEditButtonClick* de la grille. *cbsNone* : la colonne utilise le contrôle de saisie normal pour modifier les données contenues dans la colonne.
Color	Spécifie la couleur de fond des cellules de la colonne. Source par défaut : *TDBGrid.Color. (*Pour définir la couleur de fond du texte, utilisez la propriété Font.)
DropDownRows	Nombre de lignes de texte affichées dans la liste déroulante. Par défaut : 7.
Expanded	Spécifie si la colonne est étendue. S'applique uniquement aux colonnes représentant des champs ADT ou tableau.

Tableau 15.2 Propriétés de colonne (suite)

Propriété	Utilisation
FieldName	Spécifie le nom du champ associé à cette colonne (peut être vierge).
ReadOnly	*True* : les données dans la colonne ne sont pas modifiables par l'utilisateur. *False* : (valeur par défaut) les données dans la colonne sont modifiables.
Width	Spécifie la largeur de la colonne en pixels. Source par défaut : *TField.DisplayWidth*.
Font	Spécifie la fonte, la taille et la couleur du texte dans la colonne. Source par défaut : *TDBGrid.Font*.
PickList	Contient une liste de valeurs à afficher dans une liste déroulante dans la colonne.
Title	Définit les propriétés du titre de la colonne sélectionnée.

Le tableau ci-dessous résume les propriétés pouvant être définies pour la propriété *Title*.

Tableau 15.3 Propriétés disponibles pour la propriété Title

Propriété	Utilisation
Alignment	Aligne à gauche (défaut), à droite, ou centre le libellé dans le titre de colonne.
Caption	Spécifie le texte à afficher dans le titre de la colonne. Source par défaut : *TField.DisplayLabel*.
Color	Spécifie la couleur de fond de la cellule de titre de colonne. Source par défaut : *TDBGrid.FixedColor*.
Font	Spécifie la fonte, la taille et la couleur du texte dans le titre de la colonne. Source par défaut : *TDBGrid.TitleFont*.

Définition d'une colonne de liste de référence

Vous pouvez créer une colonne qui affiche une liste déroulante de valeurs, à l'image d'un contrôle boîte à options de référence. Pour que la colonne fasse office de boîte à options, attribuez à sa propriété *ButtonStyle* la valeur *cbsAuto*. Une fois que vous aurez rempli la liste de valeurs, la grille affiche automatiquement un bouton de boîte à options lorsqu'une cellule de cette colonne est en mode Edition.

Pour remplir cette liste de valeurs parmi lesquelles l'utilisateur pourra effectuer son choix, deux possibilités s'offrent à vous :

- Vous pouvez récupérer les valeurs à partir d'une table de référence. Si vous voulez qu'une colonne affiche une liste déroulante de valeurs extraites d'une table de référence distincte, vous devez définir un champ de référence dans l'ensemble de données. Pour plus d'informations sur la création de champs de référence, voir "Définition d'un champ de référence" à la page 19-10. Lorsque le champ de référence est défini, affectez son nom à la propriété *FieldName* de la colonne. La liste déroulante est automatiquement remplie avec les valeurs de référence définies dans le champ de référence.
- Vous pouvez spécifier une liste de valeurs explicitement lors de la conception. Pour entrer les valeurs de la liste à la conception, double-cliquez sur la

propriété *PickList* de la colonne dans l'inspecteur d'objets. Cela active l'éditeur de liste de chaînes, dans lequel vous pouvez entrer les valeurs qui constituent la liste de sélection de la colonne.

Par défaut, la liste déroulante affiche 7 valeurs. Vous pouvez modifier la longueur de cette liste en définissant la propriété *DropDownRows*.

Remarque Pour restaurer le comportement par défaut d'une colonne contenant une liste de sélection explicite, supprimez tout le texte de la liste de sélection à l'aide de l'éditeur de liste de chaîne.

Insertion d'un bouton dans une colonne

Une colonne peut contenir un bouton à points de suspension (…), placé à droite de l'éditeur de cellules standard. En cliquant sur la souris ou en appuyant sur *Ctrl+Entrée*, vous déclenchez l'événement *OnEditButtonClick* de la grille. Vous pouvez utiliser le bouton à points de suspension pour ouvrir des fiches contenant des vues plus détaillées des données de la colonne. Par exemple, dans une table affichant la synthèse des factures émises, vous pouvez intégrer un bouton à points de suspension dans la colonne des totaux permettant d'afficher une fiche contenant les éléments d'une facture précise ou le montant de la TVA, etc. Pour les champs graphiques, vous pouvez utiliser ce bouton pour afficher une fiche contenant l'image.

Pour doter une colonne d'un bouton à points de suspension :

1. Sélectionnez la colonne dans la boîte liste *Colonnes*.
2. Choisissez *cbsEllipsis*comme valeur de la propriété *ButtonStyle*.
3. Ecrivez un gestionnaire d'événement *OnEditButtonClick*.

Restauration des valeurs par défaut d'une colonne

A l'exécution, vous pouvez tester la propriété *AssignedValues* d'une colonne pour savoir si une propriété de colonne a été explicitement attribuée. Les valeurs non explicitement définies sont dynamiquement basées sur les valeurs par défaut de la grille ou du champ associé.

Vous pouvez annuler les modifications de propriétés apportées à une ou plusieurs colonnes. Dans l'éditeur de colonnes, sélectionnez la ou les colonnes à restaurer, puis choisissez Restaurer défauts dans le menu contextuel. Cette commande provoque l'annulation des paramètres de propriétés définis et restaure les propriétés d'une colonne aux valeurs des propriétés dérivées du composant champ sous-jacent.

A l'exécution, Il est possible de redéfinir toutes les propriétés par défaut d'une colonne en appelant la méthode *RestoreDefaults* de cette colonne. Vous pouvez aussi redéfinir les propriétés par défaut de toutes les colonnes d'une grille en appelant la méthode *RestoreDefaults* de la liste de colonnes :

```
DBGrid1.Columns.RestoreDefaults;
```

Affichage des champs ADT et tableau

Parfois, les champs de l'ensemble de données de la grille ne représentent pas de simples valeurs, telles que du texte, des graphiques, des valeurs numériques ou autres. Certains serveurs de bases de données autorisent les champs qui englobent des types de données simples, tels que les champs ADT ou tableau.

Une grille peut afficher les champs composites de deux façons :

- Elle peut aplanir le champ afin que chacun des types simples qui composent le champ apparaisse en tant que champ séparé dans l'ensemble de données. Lorsqu'un champ composite est aplani, ses composants apparaissent sous forme de champs séparés qui reflètent leur source commune uniquement si chaque nom de champ est précédé du nom du champ parent commun dans la table de base de données sous-jacente.

  Pour afficher les champs composites comme s'ils étaient aplanis, attribuez la valeur *False* à la propriété *ObjectView* de l'ensemble de données. L'ensemble de données stocke les champs composites sous la forme d'un ensemble de champs séparés et la grille reflète cette organisation en attribuant une colonne à chaque composante.

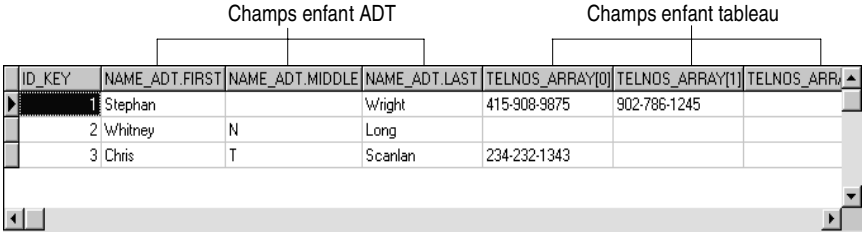
- Elle peut afficher des champs composites dans une seule colonne, indiquant ainsi qu'ils constituent un seul champ. Vous pouvez alors étendre et réduire la colonne en cliquant sur la flèche dans la barre de titre du champ ou en définissant la propriété *Expanded* de la colonne :

  - Lorsqu'une colonne est étendue, chaque champ enfant apparaît dans sa propre sous-colonne et une barre de titre correspondante apparaît sous la barre de titre du champ parent. La hauteur de la barre de titre de la grille augmente : la première ligne fournit le nom du champ composite et la seconde indique ses différentes composantes. Les champs qui ne sont pas composites apparaissent avec des barres de titre extra hautes. Cette extension se poursuit pour les composantes qui sont elles-mêmes des champs composites (par exemple, une table détail imbriquée dans une autre) et la hauteur de la barre de titre augmente en conséquence.
  - Lorsque le champ est réduit, seule la colonne apparaît avec une chaîne non modifiable des champs enfants, séparés par des virgules.

  Pour afficher un champ composite dans une colonne étendue ou réduite, attribuez la valeur *True* à la propriété *ObjectView* de l'ensemble de données. L'ensemble de données stocke le champ composite sous la forme d'un champ composant unique qui contient un ensemble de sous-champs imbriqués. La grille reflète cette organisation dans une colonne qui peut être étendue ou réduite.

La figure suivante montre une grille contenant un champ ADT et un champ tableau. La propriété *ObjectView* de l'ensemble de données a pour valeur *False* afin que chaque champ enfant dispose d'une colonne.

Figure 15.2 Contrôle TDBGrid avec ObjectView défini à False



Les figures suivantes montrent la grille dotée d'un champ ADT et d'un champ tableau. La première figure montre les champs réduits. Cet état ne permet pas de les modifier. La seconde figure montre les champs étendus. Pour étendre et réduire les champs, cliquez sur la flèche dans leur barre de titre.

Figure 15.3 Contrôle TDBGrid avec Expanded défini à False

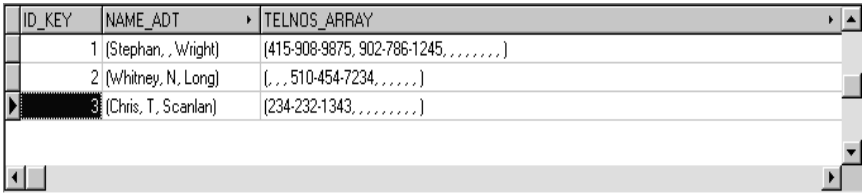
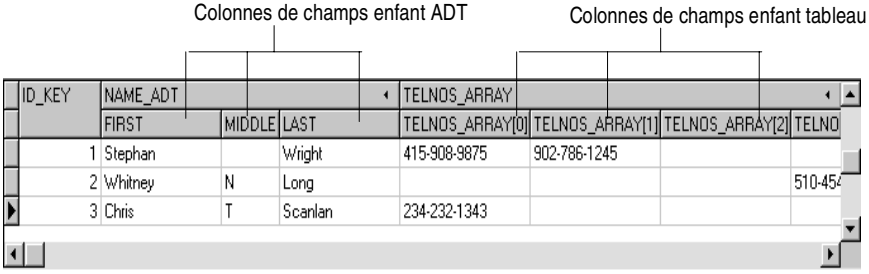


Figure 15.4 Contrôle TDBGrid avec Expanded défini à True



Le tableau suivant énumère les propriétés qui affectent l'affichage des champs ADT et tableau dans un objet *TDBGrid* :

Tableau 15.4 Propriétés relatives à l'affichage des champs composites

Propriété	Objet	Utilisation
Expandable	TColumn	Indique si la colonne peut être étendue pour afficher les champs enfants dans des colonnes séparées et modifiables. (lecture seule)
Expanded	TColumn	Spécifie si la colonne est étendue.
MaxTitleRows	TDBGrid	Spécifie le nombre maximum de lignes de titre pouvant apparaître dans la grille.

Tableau 15.4 Propriétés relatives à l'affichage des champs composites (suite)

Propriété	Objet	Utilisation
ObjectView	TDataSet	Spécifie si les champs apparaissent aplanis, ou dans un mode objet, où chaque champ objet peut être développé et réduit.
ParentColumn	TColumn	Fait référence à l'objet TColumn auquel appartient la colonne du champ enfant.

Remarque Outre des champs ADT et array, certains ensembles de données comprennent des champs qui font référence à un autre ensemble de données (champs ensemble de données) ou à un enregistrement appartenant à d'autres champs ensemble de données (référence). Les grilles orientées données affichent respectivement ces champs sous les formes Ensemble de données ou Référence. A l'exécution, un bouton à points de suspension apparaît à droite. Le fait de cliquer sur ce bouton active une nouvelle fiche dont la grille affiche le contenu du champ. Dans le cas d'un champ ensemble de données, cette grille affiche l'ensemble de données correspondant à la valeur du champ. Dans le cas d'un champ référence, cette grille contient une seule ligne qui affiche l'enregistrement d'un autre ensemble de données.

Définition des options de la grille

La propriété *Options* d'une grille permet de contrôler, à la conception, le comportement et l'aspect de la grille à l'exécution. Quand un composant grille est placé dans une fiche à la conception, la propriété *Options* de l'inspecteur d'objets apparaît avec un signe + (plus) indiquant qu'elle peut être développée pour afficher une série de propriétés booléennes définies individuellement. Pour visualiser et définir ces propriétés, cliquez sur le signe +. La liste des options apparaît dans l'inspecteur d'objets sous la propriété *Options*. Le signe + devient un signe - (moins), sur lequel vous pouvez cliquer pour réduire la liste.

Le tableau suivant liste les propriétés *Options* pouvant être définies et décrit leur effet sur la grille à l'exécution.

Tableau 15.5 Options détaillées des propriétés Options du composant TDBGrid

Option	Utilisation
dgEditing	*True* : (valeur par défaut) interdit la modification, l'insertion et la suppression d'enregistrements dans la grille. *False* : interdit la modification, l'insertion et la suppression d'enregistrements dans la grille.
dgAlwaysShowEditor	*True* : quand un champ est sélectionné, il est en mode Edition. *False* : (valeur par défaut) un champ n'est pas automatiquement en mode Edition s'il est sélectionné.
dgTitles	*True* : (valeur par défaut) affiche les noms de champs en haut de la grille. *False* : l'affichage des noms de champs est désactivé.

Tableau 15.5 Options détaillées des propriétés Options du composant TDBGrid (suite)

Option	Utilisation
dgIndicator	*True* : (valeur par défaut) la colonne d'indicateur est affichée sur la gauche de la grille et l'indicateur d'enregistrement en cours (une flèche à gauche de la grille) est activé. A l'insertion, la flèche se transforme en astérisque et se transforme en I à la modification.
	False : la colonne d'indicateur n'apparaît pas.
dgColumnResize	*True* : (valeur par défaut) les colonnes peuvent être redimensionnées en faisant glisser les lignes de colonnes dans la zone de titre. Le redimensionnement modifie la largeur correspondante du composant *TField* sous-jacent.
	False : les colonnes de la grille ne peuvent être redimensionnées.
dgColLines	*True* : (valeur par défaut) affiche une ligne verticale de séparation entre les colonnes.
	False : n'affiche pas de ligne verticale de séparation entre les colonnes.
dgRowLines	*True* : (valeur par défaut) affiche une ligne horizontale de séparation entre les enregistrements.
	False : n'affiche pas de ligne horizontale de séparation entre les enregistrements.
dgTabs	*True* : (valeur par défaut) autorise la tabulation entre les champs des enregistrements.
	False : la tabulation fait sortir du contrôle grille.
dgRowSelect	*True* : la barre de sélection occupe toute la largeur de la grille.
	False : (valeur par défaut) la sélection d'un champ d'un enregistrement ne sélectionne que ce champ.
dgAlwaysShowSelection	*True* : (valeur par défaut) la barre de sélection de la grille est toujours visible, même si un autre contrôle a la focalisation.
	False : la barre de sélection n'apparaît dans la grille que si la grille a la focalisation.
dgConfirmDelete	*True* : (valeur par défaut) demande la confirmation de la suppression d'enregistrements (*Ctrl+Suppr*).
	False : supprime les enregistrements sans confirmation.
dgCancelOnExit	*True* : (valeur par défaut) annule une insertion en attente lorsque la focalisation quitte la grille. Cela permet d'éviter des validations involontaires d'enregistrements vierges ou partiellement vierges.
	False : permet à une insertion en attente d'être validée.
dgMultiSelect	*True* : permet aux utilisateurs de sélectionner des lignes non contiguës en utilisant les touches *Ctrl+Maj* ou *Maj+flèche*.
	False : (valeur par défaut) ne permet pas à l'utilisateur de sélectionner plusieurs lignes.

Saisie de modifications dans la grille

Lors de l'exécution, vous pouvez utiliser une grille pour modifier les données existantes et pour entrer de nouveaux enregistrements si :

- La propriété *CanModify*de *l'ensemble de données* est à *True*.

- La propriété *ReadOnly* de la grille est à *False*.

Lorsqu'un utilisateur modifie un enregistrement de la grille, les modifications apportées à chaque champ sont émises dans le tampon interne des enregistrements, et ne sont pas émises (ou validées) tant que l'utilisateur n'est pas passé à un autre enregistrement de la grille. Même si vous utilisez la souris pour déplacer la focalisation sur un autre contrôle de la fiche, la grille n'écrit vos modifications que lorsque vous changez d'enregistrement en cours. Lorsqu'un enregistrement est écrit, le système vérifie si le statut des composants orientés données associés à l'ensemble de données a été modifié. En cas de problème de mise à jour d'un champ contenant des données modifiées, la grille provoque une exception et ne modifie pas l'enregistrement.

Remarque Si votre application place les mises à jour en mémoire cache, les modifications apportées aux enregistrements sont uniquement envoyées dans un cache interne. Elles ne sont répercutées dans la table de base de données sous-jacente que lorsque votre application applique les mises à jour.

Vous pouvez annuler toutes les modifications d'un enregistrement en appuyant sur *Echap* dans un champ avant d'activer un autre enregistrement.

Contrôle du dessin de la grille

Le premier niveau de contrôle du dessin des cellules de la grille consiste à définir les propriétés des colonnes. La grille utilise par défaut la fonte, l'alignement et la couleur d'une colonne pour dessiner les cellules de cette colonne. Le texte des champs de données est dessiné à l'aide des propriétés *DisplayFormat* et *EditFormat* du composant champ associé à la colonne.

Vous pouvez modifier la logique d'affichage de la grille en entrant du code dans l'événement *OnDrawColumnCell* d'une grille. Si la propriété *DefaultDrawing* de la grille est à *True*, le dessin normal est effectué avant que votre événement *OnDrawColumnCell* ne soit appelé. Votre code peut alors se superposer à l'affichage par défaut. Cette fonctionnalité vous sera utile si vous avez défini une colonne persistante vierge et désirez ajouter des graphiques dans les cellules de cette colonne.

Si vous souhaitez remplacer la logique de dessin de la grille, mettez *DefaultDrawing* à *False* et placez le code dans l'événement *OnDrawColumnCell* de la grille. Si vous désirez remplacer la logique de dessin pour certaines colonnes ou certains types de données, vous pouvez appeler *DefaultDrawColumnCell* depuis votre gestionnaire d'événement *OnDrawColumnCell* pour que la grille utilise son code de dessin normal pour les colonnes sélectionnées. Ceci réduit votre travail si vous ne voulez modifier que la façon dont les champs logiques sont dessinés, par exemple.

Comment répondre aux actions de l'utilisateur à l'exécution

Le comportement de la grille peut être modifié en écrivant des gestionnaires d'événements répondant à des actions spécifiques dans la grille. Une grille affichant en général plusieurs champs et enregistrements de façon simultanée, des besoins très spécifiques peuvent intervenir dans la réponse à des modifications de colonnes. Il est possible, par exemple, d'activer ou de désactiver un bouton de la fiche à chaque fois que l'utilisateur entre ou sort d'une colonne particulière.

Le tableau suivant liste les événements d'un contrôle grille disponibles dans l'inspecteur d'objets.

Tableau 15.6 Evénements d'un contrôle grille

Evénement	Utilisation
OnCellClick	Spécifie l'action à lancer lorsqu'un utilisateur clique sur une cellule de la grille.
OnColEnter	Spécifie l'action à lancer lorsque l'utilisateur se place dans une colonne de la grille.
OnColExit	Spécifie l'action à lancer lorsque l'utilisateur quitte une colonne de la grille.
OnColumnMoved	Appelé lorsque l'utilisateur déplace une colonne.
OnDblClick	Spécifie l'action à lancer lorsque l'utilisateur double-clique dans la grille.
OnDragDrop	Spécifie l'action à lancer lorsque l'utilisateur se sert du glisser-déplacer dans la grille.
OnDragOver	Spécifie l'action à lancer lorsque l'utilisateur fait glisser la sélection sur la grille.
OnDrawColumnCell	Appelé pour dessiner des cellules individuelles.
OnDrawDataCell	(obsolète) Dans les grilles pour lesquelles *State = csDefault*, appelé pour dessiner des cellules individuelles.
OnEditButtonClick	Appelé quand l'utilisateur clique sur un bouton à points de suspension (...) dans une colonne.
OnEndDrag	Spécifie l'action à lancer lorsque l'utilisateur arrête de faire glisser la sélection sur la grille.
OnEnter	Spécifie l'action à lancer lorsque la grille reçoit la focalisation.
OnExit	Spécifie l'action à lancer lorsque la grille perd la focalisation.
OnKeyDown	Spécifie l'action à lancer lorsque l'utilisateur appuie sur une touche ou une combinaison de touches du clavier dans la grille.
OnKeyPress	Spécifie l'action à lancer lorsque l'utilisateur appuie sur une touche alphanumérique du clavier dans la grille.
OnKeyUp	Spécifie l'action à lancer lorsque l'utilisateur relâche une touche du clavier dans la grille.
OnStartDrag	Spécifie l'action à lancer lorsque l'utilisateur démarre un glisser-déplacer sur la grille.
OnTitleClick	Spécifiez l'action à lancer lorsqu'un utilisateur clique sur le titre d'une colonne.

Il y a de nombreuses utilisations possibles pour ces événements. Par exemple, il est possible d'écrire pour l'événement *OnDblClick*, un gestionnaire qui fasse apparaître une liste surgissante dans laquelle l'utilisateur peut choisir la valeur à entrer dans la colonne. Un tel gestionnaire utiliserait la propriété *SelectedField* pour déterminer la ligne et la colonne actives.

Création d'une grille qui contient d'autres contrôles orientés données

Un contrôle *TDBCtrlGrid* affiche plusieurs champs de plusieurs enregistrements dans un format de type grille tabulaire. Chaque cellule de la grille affiche plusieurs champs d'une ligne. Pour utiliser une grille de contrôle de base de données :

1. Placez une grille de contrôle de base de données sur la fiche.
2. Donnez à la propriété *DataSource* le nom de la source de données.
3. Placez des contrôles de données dans la cellule de conception de la grille. (la cellule qui se trouve le plus haut ou le plus à gauche de la grille, la seule dans laquelle vous puissiez placer des contrôles).
4. Donnez à la propriété *DataField* de chaque contrôle de données le nom d'un champ. La source de données de ces contrôles est déjà définie comme la source de la grille de contrôle de base de données.
5. Disposez les contrôles dans la cellule comme vous l'entendez.

Lorsque vous compilez et exécutez une application contenant une grille de contrôle de base de données, la disposition des contrôles orientés données dans la cellule de conception (définie à l'exécution) est dupliquée dans chaque cellule de la grille. Chaque cellule affiche un enregistrement distinct d'un ensemble de données.

Figure 15.5 TDBCtrlGrid en mode conception

Le tableau ci-dessous résume les propriétés uniques des grilles de contrôle de base de données que vous pouvez définir en phase de conception :

Tableau 15.7 Propriétés d'une grille de contrôle de base de données

Propriété	Utilisation
AllowDelete	*True* (par défaut) : permet la suppression des enregistrements.
	False : interdit la suppression des enregistrements.
AllowInsert	*True* (par défaut) : permet l'insertion d'enregistrements.
	False : interdit l'insertion d'enregistrements.
ColCount	Définit le nombre de colonnes dans la grille (1 par défaut).
Orientation	*goVertical* (par défaut) : affiche les enregistrements de haut en bas.
	goHorizontal: affiche les enregistrements de gauche à droite.
PanelHeight	Définit la hauteur d'un volet (72 par défaut).
PanelWidth	Définit la largeur d'un volet (200 par défaut).
RowCount	Définit le nombre de volets à afficher (3 par défaut).
ShowFocus	*True* (par défaut) : affiche, à l'exécution, un rectangle de focalisation autour du volet de l'enregistrement en cours.
	False : n'affiche pas de rectangle de focalisation.

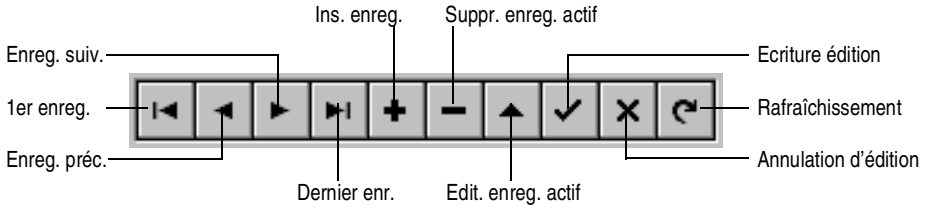
Pour plus d'informations sur les propriétés et les méthodes des grilles de contrôle de base de données, voir la *Référence VCL* en ligne.

Navigation et manipulation d'enregistrements

TDBNavigator est un contrôle simple permettant à l'utilisateur de naviguer parmi les enregistrements d'un ensemble de données et de les manipuler. Le navigateur se compose d'une série de boutons permettant à l'utilisateur de faire défiler vers l'avant ou vers l'arrière des enregistrements, un par un, d'aller directement au premier ou au dernier, d'insérer un nouvel enregistrement, de mettre à jour un enregistrement existant, d'écrire des modifications ou d'en annuler, de supprimer un enregistrement ou de rafraîchir l'affichage.

La figure suivante montre le navigateur tel qu'il apparaît par défaut lorsque vous le placez sur une fiche lors de la conception. Le navigateur se compose d'une série de boutons permettant à l'utilisateur de naviguer d'un enregistrement à l'autre dans un ensemble de données, et de modifier, supprimer, insérer et valider des enregistrements. La propriété *VisibleButtons*du navigateur vous permet d'afficher ou de cacher dynamiquement un sous-ensemble de ces boutons.

Figure 15.6 Boutons de TDBNavigator



Le tableau ci-dessous donne la description des boutons du navigateur.

Tableau 15.8 Boutons TDBNavigator

Bouton	Utilisation
Premier	Fait appel à la méthode *First* de l'ensemble de données pour que le premier enregistrement de l'ensemble de données devienne l'enregistrement en cours.
Précédent	Fait appel à la méthode *Prior* de l'ensemble de données pour que l'enregistrement précédent devienne l'enregistrement en cours.
Suivant	Fait appel à la méthode *Next* de l'ensemble de données pour que l'enregistrement suivant devienne l'enregistrement en cours.
Dernier	Fait appel à la méthode *Last* de l'ensemble de données pour que le dernier enregistrement devienne l'enregistrement en cours.
Insertion	Fait appel à la méthode *Insert* de l'ensemble de données pour insérer un nouvel enregistrement avant l'enregistrement en cours et placer l'ensemble de données en mode Insertion.
Suppression	Supprime l'enregistrement en cours. Si la propriété *ConfirmDelete* est à *True*, il est demandé confirmation avant la suppression.
Edition	Place l'ensemble de données en mode Edition afin de pouvoir modifier l'enregistrement en cours.
Ecriture	Ecrit les modifications dans l'enregistrement en cours dans la base de données.
Annulation	Annule l'édition de l'enregistrement en cours, et replace l'ensemble de données à l'état Visualisation.
Rafraîchissement	Vide les tampons d'affichage du contrôle orienté données, puis les rafraîchit à partir de la table ou de la requête physique. Utile si les données sous-jacentes ont pu être modifiées par une autre application.

Choix des boutons visibles

Lorsque vous placez un composant *TDBNavigator* sur une fiche, tous ses boutons sont visibles par défaut. Vous pouvez utiliser la propriété *VisibleButtons* pour les désactiver si vous ne comptez pas les utiliser sur une fiche. Par exemple, lorsque vous utilisez un ensemble de données unidirectionnel,seuls les boutons *Premier*, *Suivant* et *Rafraîchissement* sont pertinents. Il peut être utile de désactiver les boutons *Edition*, *Insertion*, *Suppression*, *Ecriture* et *Annulation* sur une fiche servant à parcourir des enregistrements plutôt qu'à les éditer.

Affichage et dissimulation des boutons en mode conception

La propriété *VisibleButtons*de l'inspecteur d'objets est suivie du signe + pour indiquer que sa liste d'options peut être développée. Vous pouvez choisir d'afficher une valeur booléenne pour chacun des boutons du navigateur. Pour voir ces valeurs et les définir, cliquez sur le signe +. La liste des boutons pouvant être activés ou désactivés apparaît dans l'inspecteur d'objets en dessous de la propriété *VisibleButtons*. Le signe + devient un signe - (moins), sur lequel vous pouvez cliquer pour réduire la liste des propriétés.

La visibilité d'un bouton est indiquée par l'état *Boolean* de sa valeur. Si cette valeur est à *True*, le bouton apparaît dans le composant *TDBNavigator*. Si elle est à *False*, le bouton est supprimé du navigateur lors de la conception et de l'exécution.

Remarque Quand la valeur d'un bouton est à *False*, il est supprimé du composant *TDBNavigator* et tous les autres boutons s'agrandissent pour occuper toute la largeur du contrôle. Vous pouvez faire glisser les poignées de celui-ci pour redimensionner les boutons.

Affichage et dissimulation des boutons à l'exécution

Lors de l'exécution, vous avez la possibilité de masquer ou d'afficher les boutons du navigateur en réponse aux actions de l'utilisateur ou aux états de l'application. Supposons, par exemple, que vous fournissiez un seul navigateur pour consulter deux ensembles de données, l'un permettant aux utilisateurs d'éditer les enregistrements, l'autre étant en lecture seulement. Lorsque vous passez de l'un à l'autre, vous devez masquer les boutons *Insertion, Suppression, Edition, Ecriture, Annulation* et *Rafraîchissement* pour le second ensemble de données et les afficher pour le premier.

Supposons que vous vouliez empêcher toute édition d'*OrdersTable* en masquant les boutons *Insertion, Suppression, Edition, Ecriture, Annulation* et *Rafraîchissement* du navigateur, mais que vous vouliez permettre en même temps l'édition de *CustomersTable*. La propriété *VisibleButtons* détermine quels sont les boutons affichés dans le navigateur. Voici comment ajouter le code nécessaire au gestionnaire d'événement*OnEnter*codé préalablement :

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
  begin
    DBNavigatorAll.DataSource := CustomerCompany.DataSource;
    DBNavigatorAll.VisibleButtons := [nbFirst,nbPrior,nbNext,nbLast];
  end
  else
  begin
    DBNavigatorAll.DataSource := OrderNum.DataSource;
    DBNavigatorAll.VisibleButtons := DBNavigatorAll.VisibleButtons + [nbInsert,
      nbDelete,nbEdit,nbPost,nbCancel,nbRefresh];
  end;
end;
```

Affichage de panneaux d'information

Pour afficher un panneau d'information pour chaque bouton du navigateur à l'exécution, attribuez à la propriété*ShowHint* la valeur *True*. Quand *ShowHint* a la valeur *True*, le navigateur affichera alors des panneaux d'information lorsque vous ferez passer le curseur sur les boutons. Par défaut *ShowHint* est à*False*.

La propriété *Hints* contrôle le texte des panneaux d'information des boutons. Par défaut, *Hints* est une liste de chaînes vide. Quand *Hints* est vide, un texte d'aide s'affiche par défaut pour chaque bouton. Pour personnaliser ces panneaux d'information, utilisez l'éditeur de liste de chaînes et saisissez une ligne de texte distincte pour chaque bouton dans la propriété *Hints*. Lorsqu'elles sont présentes, ces chaînes ont priorité sur les chaînes par défaut du navigateur.

Utilisation d'un navigateur pour plusieurs ensembles de données

Comme pour les autres contrôles orientés données, la propriété *DataSource* d'un navigateur spécifie la source de données qui lie le contrôle à un ensemble de données. En changeant la propriété *DataSource* d'un navigateur lors de l'exécution, vous pouvez faire en sorte qu'un même navigateur procure des fonctions de navigation pour plusieurs ensembles de données.

Supposons qu'une fiche contienne deux contrôles DBEdit liés aux ensembles de données *CustomerTable* et *OrdersTable* via respectivement les sources de données *CustomersSource* et *OrdersSource*. Lorsqu'un utilisateur accède au contrôle d'édition connecté à *CustomersSource*, le navigateur doit également utiliser *CustomersSource* ; et quand l'utilisateur accède au contrôle d'édition connecté à *OrdersSource*, le navigateur doit basculer à *OrdersSource*. Vous pouvez coder un gestionnaire d'événement *OnEnter* pour l'un des contrôles d'édition, puis le partager avec l'autre contrôle d'édition. Par exemple :

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
    DBNavigatorAll.DataSource := CustomerCompany.DataSource
  else
    DBNavigatorAll.DataSource := OrderNum.DataSource;
end;
```

Chapitre

16

Utilisation de composants d'aide à la décision

Les composants d'aide à la décision permettent de créer des graphes et des tableaux de références croisées pour visualiser et analyser des données selon différentes perspectives. Pour plus d'informations sur les références croisées, voir "Présentation des références croisées" à la page 16-2.

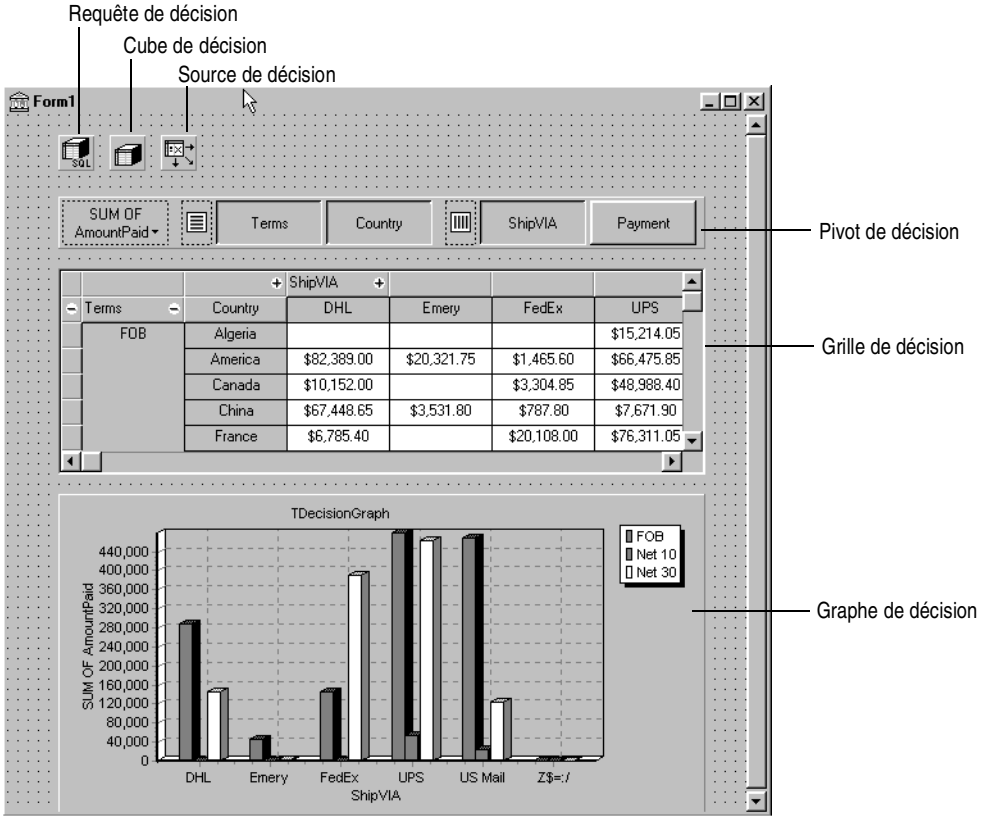
Présentation

Les composants d'aide à la décision apparaissent sur la page Decision Cube de la palette des composants :

- Le cube de décision, *TDecisionCube*, est un lieu de stockage de données multidimensionnelles.
- La source de décision, *TDecisionSource*, définit l'état actuel du pivot d'une grille ou d'un graphe de décision.
- La requête de décision, *TDecisionQuery*, est une forme spécialisée de *TQuery* utilisée pour définir les données d'un cube de décision.
- Le pivot de décision, *TDecisionPivot*, vous permet d'ouvrir et de fermer les dimensions, ou champs, d'un cube de décision, en appuyant sur des boutons.
- La grille de décision, *TDecisionGrid*, affiche des données unidimensionnelles ou multidimensionnelles sous forme d'un tableau.
- Le graphe de décision, *TDecisionGraph*, affiche les champs en provenance d'une grille de décision sous forme d'un graphe dynamique, qui change lorsque les dimensions des données sont modifiées.

La figure suivante montre tous les composants d'aide à la décision placés dans une fiche au moment de la conception.

Figure 16.1 Composants d'aide à la décision au moment de la conception



Présentation des références croisées

Les références croisées sont une façon de présenter des sous-ensembles de données afin de mettre en évidence des relations ou des tendances. Les champs des tables deviennent les dimensions de la référence croisée tandis que les valeurs des champs définissent les catégories et les calculs récapitulatifs dans chaque dimension.

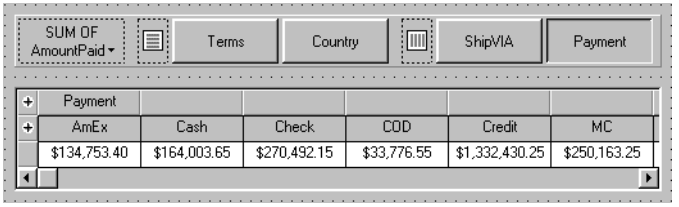
Vous pouvez utiliser les composants d'aide à la décision pour définir des références croisées dans les fiches. *TDecisionGrid* représente les données dans un tableau, tandis que *TDecisionGraph* les représente dans un graphe. *TDecisionPivot* possède des boutons pour afficher ou cacher des dimensions et pour les permuter entre lignes et colonnes.

Les références croisées peuvent avoir une ou plusieurs dimensions.

Références croisées à une dimension

Les références croisées à une dimension montrent une ligne (ou une colonne) récapitulative des catégories d'une seule dimension. Par exemple, si Payment est la dimension colonne choisie et si AmountPaid est le champ récapitulatif, la référence croisée de la figure suivante montre le montant payé avec chaque méthode.

Figure 16.2 Référence croisée à une seule dimension

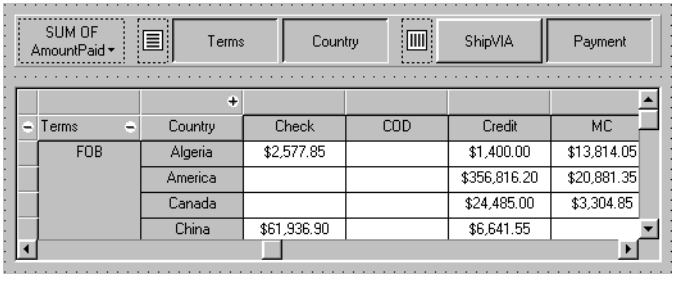


Références croisées à plusieurs dimensions

Les références croisées multidimensionnelles utilisent des dimensions supplémentaires pour les lignes et/ou les colonnes. Par exemple, une référence croisée à deux dimensions pourrait montrer le montant payé par mode de règlement pour chaque pays.

Une référence croisée à trois dimensions pourrait montrer le montant payé (AmountPaid) par mode de règlement (Payment) et par échéance (Terms) pour chaque pays (Country), comme le montre la figure suivante.

Figure 16.3 Référence croisée à trois dimensions



Instructions relatives à l'utilisation de composants d'aide à la décision

Les composants d'aide à la décision dont la liste est à la page 16-1 peuvent être utilisés ensemble pour présenter des données multidimensionnelles sous forme de tableaux et de graphes. Plusieurs grilles ou graphes peuvent être attachés à chacun des ensembles de données. Plusieurs instances de *TDecisionPivot* peuvent être utilisées pour afficher les données sous différents angles à l'exécution.

Pour créer une fiche avec des tableaux et des graphes de données multidimensionnelles, suivez ces étapes :

1. Créez une fiche.
2. Ajoutez ces composants à la fiche et utilisez l'inspecteur d'objets pour les lier comme indiqué :
   - un ensemble de données, habituellement *TDecisionQuery* (pour plus de détails, reportez-vous à "Création d'ensembles de données de décision avec l'éditeur de requête de décision" à la page 16-6) ou *TQuery ;*
   - un cube de décision, *TDecisionCube*, lié à l'ensemble de données en définissant la propriété *DataSet* du cube de décision par le nom de l'ensemble de données ;
   - une source de décision, *TDecisionSource*, liée au cube de décision en définissant la propriété *DecisionCube* de la source de décision par le nom du cube de décision.
3. Ajoutez un pivot de décision, *TDecisionPivot*, et liez-le à la source de décision dans l'inspecteur d'objets en définissant la propriété *DecisionSource* du pivot par le nom de la source de décision. Le pivot de décision est facultatif mais utile ; il permet au développeur de fiches et à l'utilisateur final de changer les dimensions affichées dans les grilles ou les graphes de décision en appuyant sur des boutons.

   Dans son orientation par défaut (horizontale), les boutons de gauche du pivot de décision correspondent aux champs de gauche de la grille de décision (les lignes) ; les boutons de droite correspondent aux champs du haut de la grille de décision (les colonnes).

   Vous pouvez déterminer l'endroit où apparaissent les boutons du pivot de décision en définissant sa propriété *GroupLayout* par *xtVertical*, *xtLeftTop* ou *xtHorizontal* (la valeur par défaut). Pour plus d'informations sur les propriétés du pivot de décision, reportez-vous à "Utilisation de pivots de décision" à la page 16-10.
4. Ajoutez un ou plusieurs graphes et/ou grilles de décision, liés à la source de décision. Pour plus de détails, reportez-vous à "Création et utilisation de grilles de décision" à la page 16-11 et "Création et utilisation de graphes de décision" à la page 16-14.
5. Utilisez l'éditeur de requête de décision ou la propriété *SQL* de *TDecisionQuery* (ou *TQuery*) pour spécifier les tables, les champs et les calculs récapitulatifs à afficher dans la grille ou dans le graphe. Le dernier champ de SQL SELECT peut être un champ récapitulatif. Les autres champs de SELECT doivent être des champs GROUP BY. Pour en savoir plus, reportez-vous à "Création d'ensembles de données de décision avec l'éditeur de requête de décision" à la page 16-6.
6. Définissez la propriété *Active* de la requête de décision (ou d'un autre composant ensemble de données) à *True*.

7 Utilisez la grille et le graphe de décision pour montrer et représenter graphiquement les différentes dimensions des données. Reportez-vous à "Utilisation de grilles de décision" à la page 16-12 et "Utilisation de graphes de décision" à la page 16-15, pour avoir des instructions et des suggestions.

Pour voir une illustration dans une fiche de tous les composants d'aide à la décision, reportez-vous à la figure 16.1 à la page 16-2.

Utilisation d'ensembles de données avec les composants d'aide à la décision

Le seul composant d'aide à la décision liant directement un ensemble de données à un cube de décision est *TDecisionCube.* Le composant *TDecisionCube* s'attend à recevoir des données dont les groupes et les calculs récapitulatifs ont été définis par une instruction SQL d'un format acceptable. La phrase GROUP BY doit contenir les mêmes champs non récapitulatifs (et dans le même ordre) que la phrase SELECT. Les champs récapitulatifs doivent être identifiés.

Le composant requête de décision, *TDecisionQuery,* est une forme spécialisée de *TQuery*. Vous pouvez utiliser TDecisionQuery pour définir de manière plus simple les dimensions (lignes et colonnes) et les valeurs récapitulatives utilisées pour fournir des données aux cubes de décision (*TDecisionCube)*. Vous pouvez aussi utiliser un *TQuery* ordinaire ou un autre ensemble de données BDE, comme ensemble de données pour *TDecisionCube*, mais la configuration correcte de l'ensemble de données et de *TDecisionCube* est dès lors à la charge du concepteur.

Pour fonctionner correctement avec un cube de décision, tous les champs de l'ensemble de données doivent être soit des dimensions, soit des champs récapitulatifs. Les récapitulations doivent être de type additif (comme la somme des valeurs ou le nombre de valeurs) et s'appliquent à chaque combinaison de valeurs des dimensions. Pour faciliter la configuration, les noms des sommes de l'ensemble de données peuvent commencer par "Sum..." tandis que ceux des dénombrements peuvent commencer par "Count...".

Le cube de décision ne peut pivoter, faire le sous-total ou forer que pour les récapitulatifs dont les cellules sont additives (SUM et COUNT sont additives alors que AVERAGE, MAX et MIN ne le sont pas). Ne concevez d'analyse croisée que pour les grilles qui contiennent uniquement des agrégats additifs. Si vous utilisez des agrégats non additifs, utilisez une grille de décision statique qui n'effectue pas de pivot, de sous-total ou de forage.

Comme la moyenne peut être calculée en divisant SUM par COUNT, une moyenne du pivot est ajoutée automatiquement quand les dimensions SUM et COUNT d'un champ sont placées dans l'ensemble de données. Utilisez ce type de moyenne de préférence à celle calculée par l'instruction AVERAGE.

Il est également possible de calculer des moyennes en utilisant COUNT(*). Pour utiliser COUNT(*) afin de calculer des moyennes, placez un sélecteur "COUNT(*) COUNTALL" dans la requête. Si vous utilisez COUNT(*) pour calculer des

moyennes, l'agrégat peut être utilisé pour tous les champs. N'utilisez COUNT(*) que dans les cas où aucun des champs ne peut contenir de valeurs vierges ou si l'opérateur COUNT n'est pas disponible pour tous les champs.

Création d'ensembles de données de décision avec TQuery ou TTable

Si vous utilisez un composant *TQuery* ordinaire comme ensemble de données de décision, vous devez configurer manuellement l'instruction SQL, en fournissant une phrase GROUP BY qui contienne les mêmes champs (et dans le même ordre) que la phrase SELECT.

L'instruction SQL doit ressembler à ce qui suit :

```
SELECT ORDERS."Terms", ORDERS."ShipVIA",
  ORDERS."PaymentMethod", SUM( ORDERS."AmountPaid" )
FROM "ORDERS.DB" ORDERS
GROUP BY ORDERS."Terms", ORDERS."ShipVIA", ORDERS."PaymentMethod"
```

L'ordre des champs dans l'instruction SELECT doit correspondre à l'ordre des champs de GROUP BY.

Avec *TTable*, vous devez spécifier au cube de décision les informations sur les champs de la requête qui servent de regroupement et ceux servant de récapitulatifs. Pour ce faire, remplissez la zone Type de dimension pour chaque champ du *DimensionMap* du cube de décision. Il faut spécifier pour chaque champ si c'est une dimension ou un récapitulatif et dans ce cas le type de récapitulatif. Comme le calcul de moyenne de pivot dépend du calcul SUM/ COUNT, il faut également spécifier le nom de champ de base afin de permettre au cube de décision d'associer les paires de récapitulatifs SUM et COUNT.

Création d'ensembles de données de décision avec l'éditeur de requête de décision

Toutes les données utilisées par les composants d'aide à la décision passent par le cube de décision, qui accepte un ensemble de données spécialement formaté, le plus souvent produit par une requête SQL. Pour plus d'informations, voir "Utilisation d'ensembles de données avec les composants d'aide à la décision" à la page 16-5.

Bien que *TTable* et *TQuery* puissent être utilisés comme ensembles de données de décision, il est plus facile d'utiliser *TDecisionQuery*; ; l'éditeur de requête de décision, fourni avec lui, peut être utilisé pour spécifier les tables, les champs et les calculs récapitulatifs qui apparaîtront dans le cube de décision, et vous aidera à configurer correctement les parties SELECT et GROUP BY de l'instruction SQL.

Pour utiliser l'éditeur de requête de décision :

1. Sélectionnez le composant requête de décision dans la fiche, puis cliquez avec le bouton droit de la souris et choisissez Editeur de requête de décision. La boîte de dialogue Editeur de requête de décision apparaît.

2. Choisissez la base de données à utiliser.
3. Pour des requêtes sur une seule table, cliquez Sélection des tables et champs.

   Pour les requêtes complexes mettant en œuvre des jointures multitables, cliquez sur le bouton Constructeur de requêtes pour afficher le constructeur SQL ou tapez l'instruction SQL dans la boîte de saisie de la page d'onglet SQL.
4. Revenez à la boîte de dialogue Editeur de requête de décision.
5. Dans la boîte de dialogue Editeur de requête de décision, sélectionnez les champs dans la boîte liste des champs disponibles et placez-les dans Dimensions ou Récapitulatifs en cliquant sur le bouton flèche droite approprié. A mesure que vous ajoutez des champs dans la liste Récapitulatifs, sélectionnez dans le menu affiché le type de récapitulatif à utiliser : somme, nombre ou moyenne.
6. Par défaut, tous les champs et calculs récapitulatifs définis dans la propriété *SQL* de la requête de décision apparaissent dans les boîtes liste Dimensions et Récapitulatifs. Pour supprimer une dimension ou un récapitulatif, sélectionnez l'élément dans la liste et cliquez sur la flèche gauche située à côté de la liste, ou double-cliquez sur l'élément à supprimer. Pour l'ajouter à nouveau, sélectionnez-le dans la boîte liste des champs disponibles et cliquez sur la flèche droite appropriée.

Lorsque le contenu de la requête de décision est défini, vous pouvez manipuler ensuite l'affichage des dimensions avec la propriété *DimensionMap* et les boutons de *TDecisionPivot*. Pour plus d'informations, voir la section suivante, "Utilisation des cubes de décision", "Utilisation de sources de décision" à la page 16-10, et "Utilisation de pivots de décision" à la page 16-10.

Remarque Quand vous utilisez l'éditeur de requête de décision, la requête est initialement gérée en utilisant la syntaxe SQL ANSI-92 puis, si c'est nécessaire, elle est ensuite traduite dans le dialecte utilisé par le serveur. L'éditeur ne lit et n'affiche que du SQL ANSI standard. La traduction dans le dialecte approprié est affectée automatiquement à la propriété *SQL* du *TDecisionQuery*. Pour modifier une requête, éditez la version ANSI-92 dans l'éditeur et pas celle contenue dans la propriété *SQL*.

Utilisation des cubes de décision

Le composant cube de décision, *TDecisionCube*, est un lieu de stockage de données multidimensionnelles qui extrait ses données d'un ensemble de données (généralement une instruction SQL spécialement structurée et entrée via *TDecisionQuery* ou *TQuery*). Les données sont stockées d'une façon qui permet de les réorganiser ou d'effectuer d'autres calculs récapitulatifs sans avoir besoin de lancer la requête une seconde fois.

Propriétés et événements des cubes de décision

Les propriétés *DimensionMap* de *TDecisionCube* ne contrôlent pas seulement les dimensions et les champs récapitulatifs qui apparaissent, mais permettent aussi de définir des plages de données ou de spécifier le nombre maximal de dimensions que le cube de décision pourra supporter. Vous pouvez aussi indiquer d'afficher ou non les données au cours de la conception. Vous pouvez afficher les noms, les valeurs (catégories), les sous-totaux ou les données. L'affichage des données pendant la conception peut prendre du temps, selon la source des données.

Lorsque vous cliquez sur les points de suspension en regard de *DimensionMap*, dans l'inspecteur d'objets, la boîte de dialogue Editeur de cube de décision apparaît. Vous pouvez utiliser ses pages et ses contrôles pour définir les propriétés *DimensionMap*.

L'événement *OnRefresh* est déclenché chaque fois qu'est reconstruit le cache du cube de décision. Les développeurs peuvent accéder aux nouvelles valeurs des propriétés *DimensionMap* et les changer à ce moment-là pour libérer la mémoire, changer le nombre maximal de dimensions ou de champs récapitulatifs, etc. *OnRefresh* sert également lorsque les utilisateurs accèdent à l'éditeur de cube de décision ; le code de l'application peut alors répondre aux modifications apportées par l'utilisateur.

Utilisation de l'éditeur de cube de décision

Vous pouvez utiliser l'éditeur de cube de décision pour définir les propriétés *DimensionMap* des cubes de décision. Vous pouvez afficher l'éditeur de cube de décision via l'inspecteur d'objets, comme indiqué dans la section précédente. Vous pouvez aussi cliquer avec le bouton droit sur un cube de décision dans une fiche au moment de la conception et choisir Editeur de cube de décision.

La boîte de dialogue Editeur de cube de décision possède deux onglets :

- Paramètres de dimensions, utilisé pour activer ou désactiver les dimensions disponibles, renommer et reformater des dimensions, placer des dimensions dans un état "perforé de manière permanente", et définir les plages de valeurs à afficher.
- Contrôle de la mémoire, utilisé pour définir le nombre maximal de dimensions et de champs récapitulatifs pouvant être actifs en même temps, pour afficher des informations sur l'utilisation de la mémoire et pour déterminer les noms et les données qui apparaissent à la conception.

Visualisation et modification des paramètres de dimensions

Pour visualiser les paramètres de dimensions, affichez l'éditeur de cube de décision et cliquez sur l'onglet Paramètres de dimensions. Ensuite, sélectionnez une dimension ou un champ récapitulatif dans la liste des champs disponibles.

Ces informations apparaissent dans les boîtes situées sur le côté droit de l'éditeur :

- Pour modifier le nom d'une dimension ou d'un champ récapitulatif apparaissant sur le pivot, la grille ou le graphe de décision, entrez un nouveau nom dans la boîte de saisie Nom affichée.
- Pour savoir si le champ sélectionné est une dimension ou un champ récapitulatif, lisez le texte se trouvant dans la boîte de saisie Type. Si l'ensemble de données est un composant *TTable*, vous pouvez utiliser Type pour spécifier si le champ sélectionné est une dimension ou un champ récapitulatif.
- Pour désactiver ou activer la dimension ou le champ récapitulatif sélectionné, modifiez le paramétrage de la boîte liste déroulante Type actif : Actif, Si besoin ou Inactif. Désactiver une dimension ou la définir par Si besoin économise de la mémoire.
- Pour modifier le format de cette dimension ou de ce champ récapitulatif, entrez une chaîne de formatage dans la boîte de saisie Format.
- Pour afficher cette dimension ou ce champ récapitulatif par Année, Trimestre ou Mois, changez le paramétrage de la boîte liste déroulante Groupage. Dans la boîte liste Groupage, vous pouvez placer la dimension ou le récapitulatif sélectionné dans un état "perforé" permanent. Cela peut être utile pour économiser de la mémoire lorsqu'une dimension a de nombreuses valeurs. Pour plus d'informations, voir "Considérations relatives au contrôle de la mémoire" à la page 16-21.
- Pour déterminer le point de départ des intervalles, commencez par choisir la valeur adaptée de regroupement dans la liste déroulante Groupage puis entrez la valeur de départ de l'intervalle dans la liste déroulante Valeur initiale.

Définition du maximum de dimensions et de récapitulations

Pour déterminer le nombre maximal de dimensions et de champs récapitulatifs disponibles pour les pivots, les grilles et les graphes liés au cube de décision sélectionné, affichez l'éditeur de cube de décision, cliquez sur l'onglet Contrôle de la mémoire et utilisez les contrôles de saisie pour ajuster le paramétrage en cours, si nécessaire. Ces paramètres permettent de contrôler la quantité de mémoire nécessaire au cube de décision. Pour plus d'informations, reportez-vous à "Considérations relatives au contrôle de la mémoire" à la page 16-21.

Visualisation et modification des options de conception

Pour déterminer combien d'informations apparaîtront lors de la conception, affichez l'éditeur de cube de décision, cliquez sur l'onglet Contrôle de la mémoire. Cochez ensuite les paramètres qui indiquent les noms et les données à afficher. L'affichage des données ou des noms des champs lors de la conception peut diminuer les performances dans certains cas à cause du temps nécessaire à l'extraction des données.

Utilisation de sources de décision

Le composant source de décision, *TDecisionSource,* définit l'état en cours des pivots des grilles ou des graphes de décision. Lorsque deux de ces objets utilisent la même source de décision, leurs pivots partagent le même état.

Propriétés et événements

Voici les propriétés et les événements spéciaux qui contrôlent l'aspect et le comportement des sources de décision :

- La propriété *ControlType* de *TDecisionSource* indique si les boutons du pivot de décision doivent agir comme des cases à cocher (sélections multiples) ou des boutons radio (sélections mutuellement exclusives).
- Les propriétés *SparseCols* et *SparseRows* de *TDecisionSource* indiquent si les colonnes ou les lignes sans valeur doivent être affichées ; si *True*, les colonnes ou les lignes vides sont affichées.
- *TDecisionSource* possède les événements suivants :
  - *OnLayoutChange* se produit lorsque l'utilisateur effectue des pivotements ou des perforations qui réorganisent les données.
  - *OnNewDimensions* se produit lorsque les données elles-mêmes sont modifiées, par exemple lorsque les champs récapitulatifs ou les dimensions sont modifiés.
  - *OnSummaryChange* se produit lorsque la valeur récapitulative en cours est modifiée.
  - *OnStateChange* se produit quand le cube de décision est activé ou désactivé.
  - *OnBeforePivot* se produit lorsque les modifications sont validées mais pas encore reflétées par l'interface utilisateur. Les développeurs ont la possibilité d'effectuer les changements, par exemple de capacité ou de l'état du pivot, avant que l'utilisateur de l'application ne puisse voir le résultat de son action.
  - *OnAfterPivot* est déclenché après une modification de l'état du pivot. Les développeurs peuvent intercepter des informations à ce moment.

Utilisation de pivots de décision

Le composant pivot de décision, *TDecisionPivot,* vous permet d'ouvrir ou de fermer les dimensions, ou champs, d'un cube de décision en appuyant sur des boutons. Lorsqu'une ligne ou une colonne est ouverte en appuyant sur un bouton *TDecisionPivot*, la dimension correspondante apparaît dans le composant *TDecisionGrid* ou *TDecisionGraph*. Lorsqu'une dimension est fermée, le détail de ses données n'apparaît pas ; elles s'intègrent aux totaux des autres dimensions. Une dimension peut aussi être en état "perforé", état dans lequel seules les

valeurs récapitulatives pour une catégorie particulière de la dimension apparaissent.

Vous pouvez utiliser le pivot de décision pour réorganiser les dimensions affichées par la grille et le graphe de décision. Faites simplement glisser un bouton vers la partie des lignes ou celle des colonnes, ou réorganisez les boutons dans la même partie.

Pour voir des illustrations de pivots de décision pendant la conception, reportez-vous aux figures 16.1, 16.2 et 16.3.

Propriétés des pivots de décision

Voici les propriétés spéciales qui contrôlent l'aspect et le comportement des pivots de décision :

- Les premières propriétés de *TDecisionPivot* définissent leur aspect et leur comportement généraux. Vous pouvez définir la propriété *ButtonAutoSize* de *TDecisionPivot* par *False* pour empêcher la réduction et le développement des boutons lorsque vous ajustez la taille du composant.
- La propriété *Groups* de *TDecisionPivot* définit quels boutons de dimensions apparaîtront. Vous pouvez grouper les boutons de sélection des lignes, des colonnes et des champs récapitulatifs à votre gré. Si vous voulez une disposition des groupes plus simple, vous pouvez placer quelque part sur votre fiche un *TDecisionPivot* contenant uniquement les lignes, et ailleurs un second contenant uniquement les colonnes.
- Généralement, *TDecisionPivot* est ajouté au-dessus de *TDecisionGrid*. Dans cette orientation par défaut (horizontale), les boutons du côté gauche de *TDecisionPivot* s'appliquent aux champs du côté gauche de *TDecisionGrid* (lignes) ; les boutons du côté droit s'appliquent aux champs du haut de *TDecisionGrid* (colonnes).
- Vous pouvez déterminer où apparaîtront les boutons de *TDecisionPivot* en définissant sa propriété *GroupLayout* par *xtVertical*, *xtLeftTop* ou *xtHorizontal* (valeur par défaut décrite au paragraphe précédent).

Création et utilisation de grilles de décision

Les composants grille de décision, *TDecisionGrid,* présentent des références croisées sous forme de tableaux. Ces tableaux de références croisées, également appelés tableaux croisés, sont décrits à la page 16-2. 16.1 à la page 16-2 montre une grille de décision placée sur une fiche pendant la conception.

Création de grilles de décision

Pour créer une fiche contenant un ou plusieurs tableaux de références croisées,

1 Suivez les étapes 1 à 3 de la section "Instructions relatives à l'utilisation de composants d'aide à la décision" à la page 16-3.

2 Ajoutez un ou plusieurs composants grille de décision (*TDecisionGrid*) et définissez dans l'inspecteur d'objets leur propriété *DecisionSource* par le nom du composant source de décision, *TDecisionSource*, auquel vous voulez relier les grilles.

3 Continuez par les étapes 5 à 7 de la section "Instructions relatives à l'utilisation de composants d'aide à la décision" à la page 16-3

Pour avoir la description de ce qui apparaît dans la grille de décision et savoir comment l'utiliser, lisez "Utilisation de grilles de décision" à la page 16-12.

Pour ajouter un graphe à la fiche, suivez les instructions de "Création de graphes de décision" à la page 16-14.

Utilisation de grilles de décision

Le composant grille de décision, *TDecisionGrid*, affiche les données du cube de décision (*TDecisionCube*) lié à la source de décision (*TDecisionSource*).

Par défaut, la grille apparaît avec les champs dimension à gauche ou en haut selon le groupage défini dans l'ensemble de données. Les catégories, une pour chaque valeur, apparaissent sous chacun des champs. Vous pouvez :

- Ouvrir et fermer les dimensions
- Réorganiser, ou faire pivoter, les lignes et les colonnes
- "Forer" pour obtenir les détails
- Limiter la sélection des dimensions à une seule dimension par axe

Pour plus d'informations sur les propriétés et les événements relatifs à la grille de décision, voir "Propriétés des grilles de décision" à la page 16-13.

Ouverture et fermeture des champs d'une grille de décision

Un signe plus (+) apparaît dans un champ dimension ou récapitulatif, quand un ou plusieurs champs sont fermés (cachés) à sa droite. Vous pouvez ouvrir d'autres champs et d'autres catégories en cliquant sur le signe plus. Un signe moins (-) indique un champ complètement ouvert (développé). Lorsque vous cliquez sur le signe moins, le champ se ferme. Cette possibilité de développement peut être désactivée ; pour plus de détails, voir "Propriétés des grilles de décision" à la page 16-13.

Réorganisation des lignes et des colonnes d'une grille de décision

Vous pouvez faire glisser des titres de lignes et de colonnes le long du même axe ou vers d'autres axes. Vous pouvez ainsi réorganiser la grille et examiner les données sous un angle nouveau, au fur et à mesure que vous changez le regroupement des données. La possibilité de pivoter peut être désactivée ; pour plus de détails, voir "Propriétés des grilles de décision" à la page 16-13.

Si vous incluez un pivot de décision, vous pouvez réorganiser l'affichage en appuyant sur ses boutons ou en les faisant glisser. Voir les instructions de "Utilisation de pivots de décision" à la page 16-10.

Perforation pour voir les détails dans les grilles de décision

Vous pouvez "forer" pour voir une dimension en détail.

Par exemple, si vous cliquez avec le bouton droit sur un libellé de catégorie (titre de ligne) pour une dimension qui en contient d'autres, vous pouvez choisir de "forer" et de voir uniquement les données de cette catégorie. Lorsqu'une dimension est "perforée", les libellés des catégories de cette dimension ne s'affichent pas sur la grille, car seuls les enregistrements correspondant à une seule catégorie sont affichés. Si vous avez un pivot de décision sur la fiche, il affiche les valeurs des autres catégories et vous permet d'en changer.

Pour "forer" dans une dimension,

- Cliquez avec le bouton droit de la souris sur le libellé d'une catégorie et choisissez Percer jusqu'à cette valeur, ou
- Cliquez avec le bouton droit de la souris sur un bouton du pivot et choisissez Perforé.

Pour que la dimension complète soit de nouveau active,

- Cliquez avec le bouton droit de la souris sur le bouton correspondant du pivot ou bien, cliquez avec le bouton droit de la souris dans le coin supérieur gauche de la grille de décision et sélectionnez la dimension.

Limite des dimensions à sélectionner dans les grilles de décision

Vous pouvez changer la propriété *ControlType* de la source de décision pour déterminer si plusieurs dimensions peuvent être sélectionnées pour chaque axe de la grille. Pour plus d'informations, voir "Utilisation de sources de décision" à la page 16-10.

Propriétés des grilles de décision

Le composant grille de décision, *TDecisionGrid*, affiche les données du composant *TDecisionCube* lié à *TDecisionSource.* Par défaut, les données apparaissent dans une grille avec les champs de catégorie à gauche et en haut de la grille.

Voici quelques propriétés spéciales qui contrôlent l'aspect et le comportement des grilles de décision :

- *TDecisionGrid* a des propriétés uniques pour chaque dimension. Pour les définir, choisissez *Dimensions* dans l'inspecteur d'objets, puis sélectionnez une dimension. Ses propriétés apparaissent alors dans l'inspecteur d'objets : *Alignment* définit l'alignement des libellés des catégories de cette dimension, *Caption* peut remplacer le nom par défaut de la dimension, *Color* définit la couleur des libellés des catégories, *FieldName* affiche le nom de la dimension active, *Format* peut contenir tout format standard pour ce type de données et *Subtotals* indique s'il faut afficher les sous-totaux pour cette dimension. Ces mêmes propriétés sont utilisées avec les champs récapitulatifs pour changer l'aspect des données récapitulatives de la grille. Pour définir les propriétés des

dimensions, cliquez sur un composant dans la fiche ou choisissez le composant dans la liste déroulante située en haut de l'inspecteur d'objets.

- La propriété *Options* de *TDecisionGrid* vous permet de contrôler l'affichage des lignes de la grille (*cgGridLines = True*), d'activer la fonction de réduction et de développement des dimensions avec les indicateurs + et - (*cgOutliner = True*) et d'activer la possibilité de pivoter par glisser-déplacer (*cgPivotable = True*).
- L'événement *OnDecisionDrawCell* de *TDecisionGrid* vous permet de changer l'aspect de chaque cellule au moment où elle est dessinée. L'événement passe en tant que paramètres par référence les valeurs de *String*, *Font* et *Color* de la cellule en cours. Vous êtes libre de modifier ces paramètres pour réaliser des effets, par exemple choisir une couleur particulière pour les valeurs négatives. En plus de la propriété *DrawState* qui est passée par *TCustomGrid*, l'événement transmet la valeur de *TDecisionDrawState*, qui peut être utilisée pour déterminer le type de cellule à dessiner. D'autres informations concernant la cellule peuvent être extraites via les fonctions *Cells*, *CellValueArray* ou *CellDrawState*.
- L'événement *OnDecisionExamineCell* de *TDecisionGrid* vous permet de connecter l'événement clic-droit aux cellules de données, afin de pouvoir afficher des informations (par exemple, des enregistrements détail) sur une cellule particulière. Lorsque l'utilisateur clique avec le bouton droit de la souris sur une cellule, l'événement est fourni avec toutes les informations qui entrent en jeu, c'est-à-dire la valeur récapitulative en cours et un tableau *ValueArray* contenant toutes les valeurs de la dimension utilisées pour calculer la valeur récapitulative.

Création et utilisation de graphes de décision

Les composants graphe de décision, *TDecisionGraph*, présentent des références croisées sous forme de graphes. Chaque graphe de décision montre la valeur d'un seul calcul récapitulatif, la somme, le nombre ou la moyenne, pour une ou plusieurs dimensions. Pour plus d'informations sur les références croisées, voir page 16-3. Pour voir des illustrations sur les graphes de décision pendant la conception, voir 16.1 à la page 16-2 et 16.4 à la page 16-16.

Création de graphes de décision

Pour créer une fiche ayant un ou plusieurs graphes de décision,

1. Suivez les étapes 1 à 3 de la section "Instructions relatives à l'utilisation de composants d'aide à la décision" à la page 16-3.
2. Ajoutez un ou plusieurs composants graphe de décision (*TDecisionGraph*) et définissez dans l'inspecteur d'objets leur propriété *DecisionSource* par le nom du composant source de décision, *TDecisionSource*, auquel vous voulez relier les graphes.

3 Continuez avec les étapes 5 à 7 de la section "Instructions relatives à l'utilisation de composants d'aide à la décision" à la page 16-3.

4 Enfin, cliquez avec le bouton droit de la souris sur le graphe et choisissez Modifier le graphe pour changer l'aspect des séries du graphe. Vous pouvez définir des propriétés modèles pour chaque dimension du graphe, puis définir les propriétés de chaque série pour remplacer ces valeurs par défaut. Pour plus de détails, voir "Personnalisation du graphe de décision" à la page 16-17.

Pour avoir la description de ce qui apparaît dans le graphe de décision et savoir comment l'utiliser, lisez la section suivante, "Utilisation de graphes de décision".

Pour ajouter une grille de décision (ou tableau croisé) à la fiche, suivez les instructions de "Création et utilisation de grilles de décision" à la page 16-11.

Utilisation de graphes de décision

Le composant graphe de décision, *TDecisionGraph*, affiche les champs de la source de décision (*TDecisionSource*) sous forme d'un graphe dynamique qui change lorsque les dimensions de données sont ouvertes, fermées, déplacées ou réorganisées à l'aide du pivot de décision (*TDecisionPivot*).

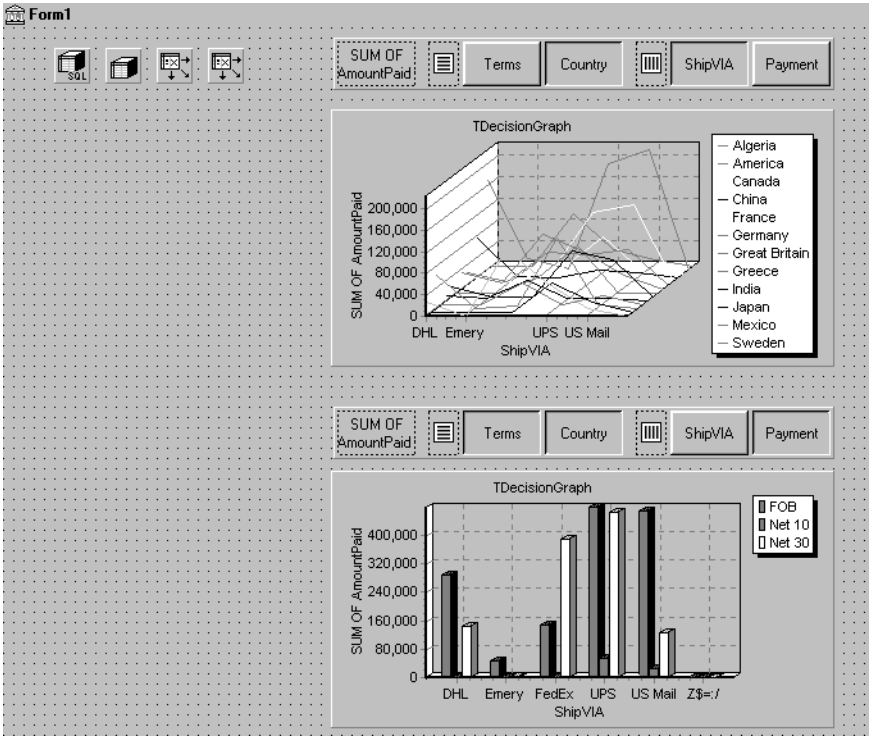
Les données représentées viennent d'un ensemble de données spécialement formaté, tel que *TDecisionQuery*. Pour avoir un aperçu de la façon dont les composants d'aide à la décision gèrent et disposent ces données, voir page 16-1.

Par défaut, la dimension de la première ligne est représentée par l'axe des x et la dimension de la première colonne par l'axe des y.

Vous pouvez utiliser les graphes de décision à la place, ou en plus, des grilles de décision (qui elles, présentent les références croisées sous forme tabulaire). Les grilles ou les graphes de décision qui sont liés à la même source de décision représentent les mêmes dimensions de données. Pour montrer différentes données récapitulatives pour les mêmes dimensions, vous pouvez lier plusieurs graphes de décision à la même source de décision. Pour montrer différentes dimensions, liez les graphes de décision à différentes sources de décision.

Par exemple, dans la figure suivante, le premier pivot de décision et le premier graphe sont liés à la première source de décision alors que le second pivot de décision et le second graphe sont liés à la seconde source de décision. Chaque graphe peut donc représenter des dimensions différentes.

Figure 16.4 Graphes de décision liés à différentes sources de décision



Pour plus d'informations sur ce qui apparaît dans un graphe de décision, voir la section suivante, "Affichage du graphe de décision".

Pour créer un graphe de décision, voir la section précédente, "Création de graphes de décision".

Pour connaître les propriétés des graphes de décision et savoir comment changer l'aspect et le comportement des graphes de décision, voir "Personnalisation du graphe de décision" à la page 16-17.

Affichage du graphe de décision

Par défaut, le graphe de décision représente les valeurs récapitulatives des catégories existant dans le premier champ de la ligne active (le long de l'axe des y) par rapport aux valeurs du premier champ de la colonne active (le long de l'axe des x). Chaque catégorie est représentée par une série.

Si une seule dimension est sélectionnée (par exemple, en cliquant sur un seul bouton de *TDecisionPivot)*, une seule série est représentée.

Si vous utilisez un pivot de décision, vous pouvez appuyer sur ses boutons pour déterminer les champs (dimensions) du cube de décision qui doivent être représentés. Pour échanger les axes du graphe, faites glisser les boutons de dimension du pivot de décision de part et d'autre de l'espace séparateur. Si le graphe est unidimensionnel, avec tous les boutons d'un côté de l'espace séparateur, vous pouvez utiliser les icônes de lignes ou de colonnes comme cible du déplacement pour ajouter des boutons de l'autre côté du séparateur et rendre le graphe multidimensionnel.

Si vous voulez qu'une seule ligne ou qu'une seule colonne soit active à la fois, vous pouvez donner la valeur *xtRadio* à la propriété *ControlType* de *TDecisionSource*. Un seul champ pourra alors être actif à la fois, et la fonctionnalité du pivot de décision correspondra au comportement du graphe. *xtRadioEx* fonctionne comme *xtRadio*, mais n'autorise pas l'état où les dimensions de toutes les lignes ou de toutes les colonnes sont fermées.

Si vous avez à la fois une grille et un graphe de décision connectés à la même *TDecisionSource*, il vaudra mieux définir *ControlType* par *xtCheck* pour revenir au comportement le plus souple de *TDecisionGrid*.

Personnalisation du graphe de décision

Le composant graphe de décision, *TDecisionGraph*, affiche les champs de la source de décision (*TDecisionSource*) sous forme d'un graphe dynamique qui change quand les dimensions sont ouvertes, fermées, déplacées ou réorganisées à l'aide du pivot de décision (*TDecisionPivot*). Vous pouvez modifier le type, les couleurs, les types de marqueurs des graphes linéaires et de nombreuses autres propriétés des graphes de décision.

Pour personnaliser un graphe,

1. Cliquez dessus avec le bouton droit de la souris et choisissez Modifier le graphe. La boîte de dialogue de modification de graphe apparaît.
2. Utilisez la page Graphe de la boîte de dialogue de modification de graphe pour voir la liste des séries visibles, sélectionner la définition de série à utiliser si deux ou plus sont disponibles pour la même série, changer le type de graphe d'un modèle ou d'une série et définir les propriétés globales du graphe.

   La liste Séries de la page Graphe montre toutes les dimensions du cube de décision (précédées de Modèle:) et les catégories actuellement visibles. Chaque catégorie, ou série, est un objet séparé. Vous pouvez :

   - ajouter ou supprimer des séries dérivées des séries existantes du graphe. Les séries dérivées peuvent fournir des annotations pour des séries existantes ou représenter des valeurs calculées à partir d'autres séries ;
   - changer le type de graphe par défaut et changer le titre des modèles et des séries.

   Vous trouverez dans l'aide en ligne la description des autres onglets de la page Graphe.

3 Utilisez la page Séries pour établir les modèles de dimensions, puis personnaliser les propriétés de chaque série du graphe.

Par défaut, les séries sont représentées par des barres d'histogramme qui peuvent avoir jusqu'à 16 couleurs. Vous pouvez modifier le type et les propriétés du modèle pour créer un nouveau modèle par défaut. Lorsque vous utilisez le pivot pour faire passer la source de décision par différents états, le modèle est utilisé pour créer de façon dynamique la série de chaque nouvel état. Pour avoir plus de détails sur les modèles, voir "Définition des modèles de graphe de décision par défaut" à la page 16-18.

Pour personnaliser une série individuelle, suivez les instructions de "Personnalisation des séries d'un graphe de décision" à la page 16-19.

Vous trouverez dans l'aide en ligne la description des autres onglets de la page Séries.

Définition des modèles de graphe de décision par défaut

Les graphes de décision affichent les valeurs provenant de deux dimensions du cube de décision : l'une est représentée par un axe et l'autre est utilisée pour créer l'ensemble des séries. Le modèle de cette dimension fournit les valeurs par défaut des propriétés des séries (si la série est représentée par une barre, une ligne, une aire, etc.) Au fur et à mesure que les utilisateurs pivotent d'un état vers l'autre, les séries exigées pour la dimension sont créées en utilisant le type de série et les autres valeurs par défaut spécifiées dans le modèle.

Un modèle distinct est fourni pour le cas où les utilisateurs pivotent vers un état dans lequel une seule dimension est active. Un état unidimensionnel est souvent représenté par un graphique sectoriel, et un modèle est fourni pour ce cas.

Vous pouvez :

- Changer le type du graphe par défaut.
- Changer les autres propriétés du modèle de graphe.
- Voir et définir les propriétés générales du graphe.

Changement du type de graphe de décision par défaut

Pour changer le type du graphe par défaut,

1 Sélectionnez un modèle dans la liste Séries de la page Graphe de la boîte de dialogue de modification de graphe.

2 Cliquez sur le bouton Modifier.

3 Sélectionnez un nouveau type et fermez la boîte de dialogue Galerie.

Changement des autres propriétés d'un modèle de graphe de décision

Pour changer la couleur ou les autres propriétés d'un modèle,

1. Sélectionnez la page Séries, en haut de la boîte de dialogue de modification de graphe.
2. Choisissez un modèle dans la liste déroulante en haut de la page.
3. Choisissez l'onglet correspondant à la propriété à modifier et faites vos choix.

Visualisation des propriétés globales d'un graphe de décision

Pour voir et définir les propriétés d'un graphe de décision autres que le type ou les séries,

1. Sélectionnez la page Graphe en haut de la boîte de dialogue de modification de graphe.
2. Choisissez l'onglet correspondant à la propriété à modifier et faites vos choix.

Personnalisation des séries d'un graphe de décision

Les modèles fournissent de nombreux paramètres par défaut pour chaque dimension du cube de décision, tels le type de graphe et la façon d'afficher les séries. D'autres paramètres par défaut, tels les couleurs des séries, sont définis par *TDecisionGraph*. Vous pouvez remplacer les paramètres par défaut de chaque série.

Les modèles doivent être utilisés pour que le programme crée les séries correspondant aux catégories et doivent être abandonnés quand ce n'est plus nécessaire. Si vous voulez, vous pouvez définir des séries personnalisées pour des valeurs particulières de catégories. Utilisez le pivot afin que le graphe affiche une série pour la catégorie que vous voulez personnaliser. Quand la série est affichée sur le graphe, vous pouvez utiliser l'éditeur de graphe pour :

- Changer le type de graphe.
- Changer d'autres propriétés concernant les séries.
- Enregistrer les séries spécifiques au graphe que vous venez de personnaliser.

Pour définir des modèles de séries et définir des options par défaut globales, voir "Définition des modèles de graphe de décision par défaut" à la page 16-18.

Changement du type de graphe des séries

Par défaut, les séries ont toutes le même type de graphe, défini par le modèle de sa dimension. Pour changer toutes les séries d'un même graphe, il suffit de changer le type du modèle. Pour ce faire, reportez-vous à "Changement du type de graphe de décision par défaut" à la page 16-18.

Pour changer le type de graphe d'une seule série,

1. Sélectionnez une série dans la liste Séries de la page Graphe de l'éditeur de graphe.
2. Cliquez sur le bouton Modifier.

3 Sélectionnez un nouveau type et fermez la boîte de dialogue Galerie.

4 Activez la case à cocher d'enregistrement des séries.

Changement des autres propriétés des séries d'un graphe de décision

Pour changer la couleur ou d'autres propriétés des séries d'un graphe de décision,

1 Sélectionnez la page Séries, en haut de la boîte de dialogue de modification de graphe.

2 Choisissez une série dans la liste déroulante, en haut de la page.

3 Choisissez l'onglet correspondant à la propriété à modifier et faites vos choix.

4 Activez la case à cocher d'enregistrement des séries.

Enregistrement des paramètres des séries d'un graphe de décision

Par défaut, seuls les paramètres des modèles sont enregistrés en mode conception. Les modifications faites à des séries particulières ne sont enregistrées que si la case d'enregistrement de ces séries est cochée dans la boîte de dialogue de modification de graphe.

L'enregistrement des séries occupant beaucoup de mémoire, pensez à désactiver cette case quand vous n'avez pas besoin de les enregistrer.

Utilisation des composants d'aide à la décision à l'exécution

A l'exécution, les utilisateurs peuvent effectuer de nombreuses opérations en cliquant avec le bouton gauche, en cliquant avec le bouton droit et en faisant glisser les composants d'aide à la décision visibles. Ces opérations, décrites plus haut, sont résumées ici.

Pivots de décision à l'exécution

Les utilisateurs peuvent :

- cliquer avec le bouton gauche sur le bouton récapitulatif, à l'extrémité gauche du pivot de décision, pour afficher la liste des récapitulations disponibles. Ils peuvent utiliser cette liste pour changer les données récapitulatives affichées dans les grilles et les graphes de décision.
- Cliquer avec le bouton droit de la souris sur un bouton de dimension et choisir de :
  - le déplacer de la partie lignes vers la partie colonnes ou l'inverse ;
  - "forer" pour afficher les données détail.

- Cliquer avec le bouton gauche sur un bouton de dimension, après avoir choisi la commande de forage et sélectionner :
  - ouvrir dimension, pour revenir au niveau supérieur de cette dimension ;
  - toutes les valeurs, pour basculer entre l'affichage dans les grilles de décision des récapitulations seulement ou des récapitulations plus les autres valeurs.
  - à partir de la liste des catégories disponibles, une catégorie à "forer" pour connaître les valeurs détail.
- Cliquer avec le bouton gauche sur un bouton de dimension pour ouvrir ou fermer cette dimension.
- Faire glisser les boutons de dimension depuis la partie lignes vers la partie colonnes et réciproquement ; ils peuvent ensuite les placer à côté des boutons existant dans cette partie ou sur l'icône de lignes ou de colonnes.

Grilles de décision à l'exécution

Les utilisateurs peuvent :

- Cliquer avec le bouton droit de la souris à l'intérieur de la grille de décision et choisir l'une des possibilités suivantes :
  - activer et désactiver alternativement les sous-totaux pour des groupes individuels de données, pour toutes les valeurs d'une dimension, ou pour toute la grille ;.
  - afficher l'éditeur de cube de décision, décrit à la page 16-8.
  - ouvrir et fermer alternativement les dimensions et les récapitulations.
- Cliquer sur + et sur – dans les titres de lignes ou de colonnes pour ouvrir et fermer les dimensions.
- Faire glisser les dimensions des lignes vers les colonnes et réciproquement.

Graphes de décision à l'exécution

Les utilisateurs peuvent faire glisser la souris d'un côté à l'autre ou de haut en bas du graphe pour faire défiler les catégories et les valeurs non visibles à l'écran.

Considérations relatives au contrôle de la mémoire

Un champ dimension ou récapitulatif chargé dans le cube de décision occupe de l'espace mémoire. L'ajout d'un nouveau champ récapitulatif augmente de façon linéaire l'occupation mémoire : par exemple, un cube de décision avec deux champs récapitulatifs occupe deux fois plus de mémoire qu'avec un seul, avec trois champs récapitulatifs il occupe trois fois plus de mémoire, etc. La

consommation de mémoire pour les dimensions augmente plus vite. L'ajout d'une dimension de 10 valeurs multiplie par 10 la consommation de mémoire (par rapport à l'ajout d'une dimension qui aurait une seule valeur) et l'ajout d'une dimension de 100 valeurs la multiplie par 100. L'ajout de dimensions à un cube de décision peut avoir un effet dramatique sur l'utilisation de la mémoire et entraîner très vite une baisse des performances. Cet effet est particulièrement prononcé quand les dimensions ajoutées comportent de nombreuses valeurs.

Les composants d'aide à la décision contiennent un certain nombre d'options qui vous aident à contrôler comment et quand la mémoire est utilisée. Pour plus d'informations sur les propriétés et les techniques indiquées ici, recherchez *TDecisionCube* dans l'aide en ligne.

Définition du maximum de dimensions, de champs récapitulatifs, et de cellules

Les propriétés *MaxDimensions* et *MaxSummaries* des cubes de décision sont utilisées avec la propriété *CubeDim.ActiveFlag* pour contrôler le nombre de dimensions et de champs récapitulatifs pouvant être chargés en même temps. Dans l'éditeur de cube de décision (groupe Capacité du cube, page Contrôle de la mémoire), vous pouvez définir le nombre maximal de valeurs afin de contrôler le nombre de dimensions et de champs récapitulatifs pouvant être présents en mémoire.

La limitation du nombre de dimensions et de champs récapitulatifs permet de réduire grossièrement la quantité de mémoire utilisée par le cube de décision. Mais, elle ne permet pas de distinguer les dimensions ayant peu de valeurs de celles en ayant beaucoup. Pour avoir un meilleur contrôle des besoins en mémoire du cube de décision, vous devez aussi limiter le nombre de cellules. Définissez le nombre maximal de cellules dans l'éditeur de cube de décision (groupe Capacité du cube, page Contrôle de la mémoire).

Définition de l'état des dimensions

La propriété *ActiveFlag* contrôle les dimensions à charger. Vous pouvez définir cette propriété dans la page Paramètres de dimensions de l'éditeur de cube de décision, en utilisant Type actif. Quand ce contrôle est mis à *Actif*, la dimension sera chargée inconditionnellement et occupera toujours l'espace mémoire. Notez que le nombre de dimensions dans cet état doit toujours être inférieur à *MaxDimensions*, et que le nombre des champs récapitulatifs mis à *Actif* doit être inférieur à *MaxSummaries*. Ne mettez à *Actif* une dimension ou un champ récapitulatif que s'il faut absolument qu'il soit disponible à tout moment. Le choix de *Actif* diminue la capacité de mémoire disponible que peut gérer le cube.

Lorsque *ActiveFlag* est définie à *AsNeeded*, une dimension ou un champ récapitulatif n'est chargé que s'il peut l'être sans dépasser les limites de *MaxDimensions*, *MaxSummaries* ou *MaxCells*. Le cube de décision permutera en et hors mémoire les dimensions et les champs récapitulatifs marqués *AsNeeded* pour respecter les limites imposées par *MaxCells*, *MaxDimensions* et *MaxSummaries*.

C'est-à-dire qu'une dimension ou un champ récapitulatif ne sera pas en mémoire quand il n'est pas utilisé. Définir par *AsNeeded* les dimensions qui ne sont pas fréquemment utilisées entraîne de meilleures performances pour le chargement et le pivotement, malgré le temps d'accès aux dimensions non chargées.

Utilisation de dimensions paginées

Quand Binning a la valeur Set dans la page Paramètres de dimensions de l'éditeur de cube de décision et si Start Value n'est pas NULL, la dimension est dite "paginée" ou "perforée de manière permanente". Il n'est possible d'accéder aux données que pour une seule valeur à la fois de cette dimension même s'il est possible par code d'accéder séquentiellement à plusieurs valeurs. Il n'est pas possible d'ouvrir ou de pivoter une telle dimension.

Les données de dimensions comportant un grand nombre de valeurs différentes consomment beaucoup de mémoire. En paginant de telles dimensions, il est possible d'afficher le récapitulatif pour une seule valeur à la fois. Les informations affichées ainsi sont plus lisibles et la gestion mémoire est simplifiée.

Chapitre

17

Connexion aux bases de données

La plupart des composants ensemble de données peuvent se connecter directement à un serveur de bases de données. Une fois connecté, l'ensemble de données communique avec le serveur automatiquement. Lorsque vous ouvrez l'ensemble de données, il se remplit avec les données du serveur et, lorsque vous validez des enregistrements, ceux-ci sont envoyés au serveur et appliqués. Un composant connexion peut être partagé par plusieurs ensembles de données, ou chaque ensemble de données peut utiliser sa propre connexion.

Chaque type d'ensemble de données se connecte au serveur de bases de données à l'aide de son propre type de composant connexion, qui utilise un mécanisme unique d'accès aux données. Le tableau suivant présente ces mécanismes et les composants connexion associés :

Tableau 17.1 Composants connexion de base de données

Mécanisme d'accès aux données	Composant connexion
BDE (moteur de bases de données Borland).	TDatabase
Objets de données ActiveX (ActiveX Data Objects, ADO).	TADOConnection
dbExpress.	TSQLConnection
InterBase Express.	TIBDatabase

Remarque Pour une présentation de certains avantages et inconvénients de chacun de ces mécanismes, voir "Utilisation des bases de données" à la page 14-1.

Le composant connexion fournit toutes les informations nécessaires à l'établissement d'une connexion de base de données. Ces informations diffèrent d'un type de composant connexion à l'autre :

- Pour la description d'une connexion BDE, voir "Identification de la base de données" à la page 20-15.
- Pour la description d'une connexion ADO, voir "Connexion à un stockage de données avec TADOConnection" à la page 21-3.

- Pour la description d'une connexion dbExpress, voir "Configuration de TSQLConnection" à la page 22-3.
- Pour la description d'une connexion InterBase Express, voir l'aide en ligne de *TIBDatabase*.

Bien que chaque type d'ensemble de données utilise un composant connexion différent, ils dérivent tous de *TCustomConnection*. Ils effectuent tous la plupart des mêmes tâches et mettent à disposition la plupart des mêmes propriétés, méthodes et événements. Ce chapitre présente de nombreuses tâches communes.

Utilisation de connexions implicites

Quel que soit le mécanisme d'accès aux données utilisé, vous pouvez toujours créer le composant connexion explicitement puis l'utiliser pour gérer la connexion au serveur de bases de données et la communication avec celui-ci. Pour les ensembles de données BDE et ADO, vous pouvez également décrire la connexion de base de données par le biais des propriétés de l'ensemble de données puis laisser celui-ci générer une connexion implicite. Pour les ensembles de données BDE, vous spécifiez une connexion implicite avec la propriété *DatabaseName*. Pour les ensembles de données ADO, vous utilisez la propriété *ConnectionString*.

Lorsque vous utilisez une connexion implicite, vous n'avez pas besoin de créer explicitement un composant connexion. Cela peut simplifier le développement de votre application et la connexion par défaut que vous spécifiez peut couvrir de nombreuses situations. Toutefois, pour les applications client/serveur stratégiques complexes impliquant de nombreux utilisateurs et différentes contraintes en matière de connexions de bases de données, vous devez créer vos propres composants connexion afin d'adapter chaque connexion de base de données aux besoins de votre application. Les composants connexion explicites offrent un contrôle accru. Par exemple, vous devez accéder au composant connexion pour effectuer les tâches suivantes :

- Personnaliser la prise en charge de la connexion au serveur de bases de données. (Les connexions implicites affichent une boîte de dialogue d'ouverture de session par défaut invitant l'utilisateur à indiquer un nom d'utilisateur et un mot de passe.)
- Contrôler les transactions et spécifier leurs niveaux d'isolement.
- Exécuter des commandes SQL sur le serveur sans utiliser un ensemble de données.
- Réaliser des actions sur tous les ensembles de données ouverts connectés à la même base de données.

En outre, si un serveur est utilisé par plusieurs ensembles de données, le recours à un composant connexion permet de ne spécifier le serveur qu'en un seul endroit. Ainsi, si vous changez le serveur, vous n'avez pas besoin de mettre à jour plusieurs composants ensemble de données mais uniquement le composant connexion.

Contrôles des connexions

Pour qu'une connexion au serveur de base de données puisse être établie, votre application doit fournir certaines informations essentielles qui décrivent le serveur désiré. Chaque type de composant connexion met à disposition un ensemble différent de propriétés permettant d'identifier le serveur. En règle générale, toutefois, ils permettent tous de nommer le serveur souhaité et fournissent un ensemble de paramètres de connexion qui contrôlent l'établissement de la connexion. Les paramètres de connexion varient d'un serveur à l'autre. Ils peuvent inclure des informations telles que les nom et mot de passe utilisateur, la taille maximale des champs BLOB, les rôles SQL, etc.

Une fois que vous avez identifié le serveur désiré et les paramètres de connexion, vous pouvez utiliser le composant connexion pour ouvrir ou fermer une connexion explicitement. Le composant connexion génère des événements lorsqu'il ouvre ou ferme une connexion, qui vous permettent de personnaliser la façon dont votre application réagit aux modifications apportées à la connexion de base de données.

Connexion à un serveur de bases de données

Un composant connexion permet d'établir une connexion à un serveur de bases de données de deux façons :

- Appeler la méthode *Open*.
- Attribuer à la propriété *Connected* la valeur *True*.

Le fait d'appeler la méthode *Open* attribue à *Connected* la valeur *True*.

Remarque Lorsqu'un composant connexion n'est pas connecté à un serveur et qu'une application essaie d'ouvrir l'un de ses ensembles de données associés, l'ensemble de données appelle automatiquement la méthode *Open* du composant connexion.

Lorsque vous attribuez à *Connected* la valeur *True*, le composant connexion génère d'abord un événement *BeforeConnect*, dans lequel vous pouvez effectuer toute initialisation. Par exemple, vous pouvez utiliser cet événement pour modifier les paramètres de connexion.

Après l'événement *BeforeConnect*, il est possible que le composant connexion affiche une boîte de dialogue d'ouverture de session par défaut, suivant votre choix en matière de contrôle de connexion au serveur. Il transmet ensuite le nom d'utilisateur et le mot de passe au pilote et ouvre une connexion.

Une fois la connexion ouverte, le composant connexion génère un événement *AfterConnect*, dans lequel vous pouvez effectuer toute tâche nécessitant une connexion ouverte.

Remarque Certains composants connexion génèrent également des événements supplémentaires lors de l'établissement d'une connexion.

Une fois établie, la connexion est conservée tant qu'au moins un ensemble de données actif l'utilise. Si aucun ensemble de données n'est plus actif, le

composant connexion abandonne la connexion. Certains composants connexion mettent à disposition une propriété *KeepConnection* qui permet à la connexion de demeurer ouverte même si tous les ensembles de données qui l'utilisent sont fermés. Si *KeepConnection* a pour valeur *True*, la connexion est maintenue. Pour les connexions vers des serveurs de bases de données distants, ou pour les applications qui ouvrent et ferment fréquemment des ensembles de données, il est préférable que *KeepConnection* ait pour valeur *True* afin de réduire le trafic sur le réseau et d'accélérer l'application. Si *KeepConnection* a pour valeur *False,* la connexion est fermée dès qu'il n'y a plus d'ensemble de données actif utilisant la base de données. Si un ensemble de données (qui utilise la base de données) est ultérieurement ouvert, la connexion doit être à nouveau établie et initialisée.

Déconnexion d'un serveur de base de données

Un composant connexion permet de se déconnecter d'un serveur de deux façons :

- Attribuer à la propriété *Connected* la valeur *False*.
- Appeler la méthode *Close*.

Le fait d'appeler *Close* attribue à *Connected* la valeur *False*.

Lorsque *Connected* prend pour valeur *False*, le composant connexion génère un événement *BeforeDisconnect*, dans lequel vous pouvez effectuer tout nettoyage avant la fermeture de la connexion. Par exemple, vous pouvez utiliser cet événement pour placer en mémoire cache les informations sur tous les ensembles de données ouverts avant qu'ils ne soient fermés.

Après l'événement *BeforeConnect*, le composant connexion ferme tous les ensembles de données ouverts et se déconnecte du serveur.

Enfin, le composant connexion génère un événement *AfterDisconnect*, dans lequel vous pouvez répondre à la modification de l'état de la connexion, par exemple en activant un bouton Connexion dans l'interface utilisateur.

Remarque Le fait d'appeler *Close* ou d'attribuer à *Connected* la valeur *False* provoque la déconnexion d'un serveur de base de données même si le composant connexion dispose d'une propriété *KeepConnection* ayant pour valeur *True*.

Contrôle de la connexion au serveur

La plupart des serveurs de bases de données distants incluent une gestion de la sécurité pour empêcher les accès non autorisés. Généralement, le serveur demande un nom d'utilisateur et un mot de passe lors de la procédure de connexion avant d'autoriser l'accès à une base de données.

Lors de la conception, si un serveur requiert une procédure de connexion, une boîte de dialogue de connexion standard demande de saisir un nom d'utilisateur et un mot de passe au moment de la première tentative de connexion à la base de données.

A l'exécution, trois moyens permettent de gérer la procédure de connexion d'un serveur :

- Laisser la boîte de dialogue de connexion par défaut demander et traiter la connexion. C'est l'approche par défaut. Initialisez la propriété *LoginPrompt* du composant connexion à *True* (valeur par défaut) et ajoutez DBLogDlg à la clause uses de l'unité qui déclare le composant connexion. Votre application affiche la boîte de dialogue standard de connexion lorsque le serveur attend un nom d'utilisateur et un mot de passe.
- Fournir les informations de connexion avant la tentative de connexion. Chaque type de composant connexion utilise un mécanisme différent pour spécifier le nom d'utilisateur et le mot de passe :
  - Pour les ensembles de données BDE, dbExpress et InterBase Express, les paramètres de connexion du nom d'utilisateur et du mot de passe sont accessibles par le biais de la propriété *Params*. (Pour les ensembles de données BDE, les valeurs de paramètre peuvent également être associées à un alias BDE, tandis que pour les ensembles de données dbExpress, elles peuvent également être associées à un nom de connexion).
  - Pour les ensembles de données ADO, le nom d'utilisateur et le mot de passe peuvent être inclus dans la propriété *ConnectionString* (ou fournis comme paramètres de la méthode *Open*).

  Si vous spécifiez le nom d'utilisateur et le mot de passe avant que le serveur ne les demande, veillez à affecter au paramètre *LoginPrompt* la valeur *False* afin que la boîte de dialogue de connexion par défaut ne s'affiche pas. Par exemple, le code suivant définit le nom d'utilisateur et le mot de passe sur un composant connexion SQL dans le gestionnaire d'événement *BeforeConnect*, en décodant un mot de passe crypté associé au nom de connexion en cours :

```
procedure TForm1.SQLConnectionBeforeConnect(Sender: TObject);
begin
  with Sender as TSQLConnection do
  begin
    if LoginPrompt = False then
    begin
      Params.Values['User_Name'] := 'SYSDBA';
      Params.Values['Password'] := Decrypt(Params.Values['Password']);
    end;
  end;
end;
```

  L'initialisation du nom d'utilisateur et du mot de passe à la conception ou l'utilisation de chaînes codées en dur dans le code génèrent l'intégration des valeurs dans le fichier exécutable de l'application. Cette approche compromet la sécurité du serveur car elle permet de les trouver facilement.
- Fournir votre propre traitement de l'événement de connexion. Le composant connexion génère un événement lorsqu'il requiert le nom d'utilisateur et le mot de passe.
  - Pour *TDatabase*, *TSQLConnection* et *TIBDatabase*, il s'agit d'un événement *OnLogin*. Le gestionnaire d'événement possède deux paramètres, le

composant connexion et une copie locale des paramètres de nom d'utilisateur et de mot de passe dans une liste de chaînes. (*TSQLConnection* comprend également le paramètre de base de données). Vous devez affecter à la propriété *LoginPrompt* la valeur *True* pour que cet événement puisse se produire. Lorsque la propriété *LoginPrompt* a pour valeur *False* et qu'un gestionnaire est affecté pour l'événement *OnLogin*, il est impossible d'établir une connexion à la base de données, car la boîte de dialogue par défaut ne s'affiche pas, et le gestionnaire d'événement *OnLogin* ne s'exécute jamais.

- Pour *TADOConnection*, l'événement est un événement *OnWillConnect*. Le gestionnaire d'événement possède cinq paramètres, le composant connexion et quatre paramètres qui renvoient les valeurs déterminant la connexion (notamment deux paramètres pour le nom d'utilisateur et le mot de passe). Cet événement se produit toujours, quelle que soit la valeur de *LoginPrompt*.

Ecrivez un gestionnaire pour l'événement dans lequel vous définissez les paramètres de connexion. Dans l'exemple suivant, les valeurs des paramètres USER NAME et PASSWORD sont fournies à l'aide d'une variable globale (*UserName*) et d'une méthode qui renvoie un mot de passe en fonction d'un nom d'utilisateur (*PasswordSearch*) :

```
procedure TForm1.Database1Login(Database: TDatabase; LoginParams: TStrings);
begin
  LoginParams.Values['USER NAME'] := UserName;
  LoginParams.Values['PASSWORD'] := PasswordSearch(UserName);
end;
```

Comme dans les autres méthodes de transmission de paramètres de connexion, lorsque vous écrivez un gestionnaire d'événement *OnLogin* ou *OnWillConnect*, évitez de coder en dur le mot de passe dans le code de votre application. Il doit apparaître uniquement sous la forme d'une valeur cryptée, d'une entrée de base de données sécurisée utilisée par votre application pour rechercher la valeur, ou être dynamiquement obtenu auprès de l'utilisateur.

Gestion des transactions

Une *transaction* est un groupe d'actions devant être menées avec succès sur une ou plusieurs tables d'une base de données avant d'être *validées* (rendues définitives). Si l'une des actions du groupe échoue, toutes les actions sont *annulées* (abandonnées). Les transactions préservent l'homogénéité de la base de données en cas d'occurrence d'un problème suite à l'exécution d'une des actions qui les composent.

Par exemple, dans une application bancaire, le transfert de fonds d'un compte vers un autre est une opération qui mérite d'être protégée avec une transaction. Si, après diminution du solde d'un compte, une erreur se produit dans l'augmentation du solde de l'autre compte, il est souhaitable d'annuler la transaction afin que la base de données continue de refléter le solde total correct.

Il est toujours possible de gérer les transactions par émission directe de commandes SQL à la base de données. A l'exception de certaines bases de

données qui n'offrent aucune prise en charge des transactions, la plupart des bases de données fournissent leur propre modèle de gestion des transactions. Si votre serveur de base de données le permet, vous pouvez coder directement votre propre gestion des transactions afin de tirer parti des fonctionnalités avancées de gestion des transactions, telles que la mise en mémoire cache des schémas.

Si vous n'avez pas besoin d'utiliser de fonctionnalités avancées, les composants connexion fournissent un ensemble de méthodes et de propriétés permettant de gérer les transactions sans explicitement émettre de commande SQL. Grâce à ces propriétés et méthodes, vous n'avez pas besoin de personnaliser votre application en fonction de chaque type de serveur de base de données utilisé, sous réserve que le serveur prenne en charge les transactions. (Le moteur BDE offre également une prise en charge limitée des transactions pour les tables locales sans prise en charge des transactions du serveur. Lorsque le moteur BDE n'est pas utilisé, toute tentative de démarrage de transactions sur une base de données qui ne les prend pas en charge amène les composants connexion à déclencher une exception.)

Attention lorsqu'un composant fournisseur d'ensemble de donnéesapplique des mises à jour, il génère implicitement les transactions pour toutes les mises à jour. Veillez à ce que toutes les transactions explicitement démarrées n'entrent pas en conflit avec celles générées par le fournisseur.

Démarrage d'une transaction

Lorsque vous démarrez une transaction, toutes les instructions suivantes qui réalisent des opérations de lecture ou d'écriture sur la base de données interviennent dans le contexte de cette transaction jusqu'à ce que la transaction soit achevée de façon explicite ou, dans le cas de transactions se chevauchant, jusqu'à ce qu'une autre transaction soit démarrée. Chaque instruction est considérée comme faisant partie d'un groupe. Si les modifications ne sont pas validées avec succès dans la base de données, chaque modification apportée dans le groupe doit être annulée.

Lorsque la transaction est en cours de traitement, la vue des données des tables de la base de données est déterminée par le niveau d'isolement des transactions. Pour plus d'informations sur les niveaux d'isolement des transactions, voir "Spécification du niveau d'isolement des transactions" à la page 17-10.

Pour *TADOConnection*, démarrez une transaction en appelant la méthode *BeginTrans* :

```
Level := ADOConnection1.BeginTrans;
```

BeginTrans renvoie le niveau d'imbrication de la transaction démarrée. Une transaction imbriquée est une transaction qui fait partie d'une autre transaction parent. Une fois que le serveur a démarré la transaction, la connexion ADO reçoit un événement *OnBeginTransComplete*.

Pour *TDatabase,* utilisez la méthode *StartTransaction TDataBase* ne prend pas en charge les transactions imbriquées ou se chevauchant. Si vous appelez la

méthode *StartTransaction* du composant *TDatabase* alors qu'une autre transaction est en cours, une exception est déclenchée. Pour éviter d'appeler *StartTransaction*, vous pouvez vérifier la propriété *InTransaction* :

```
if not Database1.InTransaction then
  Database1.StartTransaction;
```

TSQLConnection utilise également la méthode *StartTransaction* dans une version qui offre davantage de contrôle. Notamment, *StartTransaction* accepte un descripteur de transaction qui permet de gérer plusieurs transactions simultanées et de spécifier le niveau d'isolement des transactions pour chaque transaction. (Pour plus d'informations sur les niveaux de transaction, voir "Spécification du niveau d'isolement des transactions" à la page 17-10.) Pour gérer plusieurs transactions simultanées, attribuez au champ *TransactionID* du descripteur de transaction une valeur unique. *TransactionID* peut prendre n'importe quelle valeur, sous réserve qu'elle soit unique (c'est-à-dire non conflictuelle avec une transaction en cours). Suivant le serveur, les transactions démarrées par *TSQLConnection* peuvent être imbriquées (comme dans le cas d'une connexion ADO) ou se chevaucher.

```
var
  TD: TTransactionDesc;
begin
  TD.TransactionID := 1;
  TD.IsolationLevel := xilREADCOMMITTED;
  SQLConnection1.StartTransaction(TD);
```

Par défaut, dans le cas des transactions se chevauchant, la première transaction devient inactive lorsque la seconde démarre, bien que vous puissiez différer la validation ou l'annulation de la première. Si vous utilisez *TSQLConnection* avec une base de données InterBase, vous pouvez identifier chaque ensemble de données dans votre application avec une transaction active particulière, en définissant sa propriété *TransactionLevel*. En d'autres termes, après le démarrage d'une seconde transaction, vous pouvez continuer à utiliser les deux transactions simultanément, en associant simplement un ensemble de données à la transaction qui vous intéresse.

Remarque Contrairement à *TADOConnection*, *TSQLConnection* et *TDatabase* ne reçoivent pas d'événements au démarrage des transactions.

InterBase Express offre davantage de contrôle que *TSQLConnection* en utilisant un composant transaction distinct au lieu de démarrer les transactions au moyen du composant connexion. Vous pouvez, toutefois, utiliser *TIBDatabase* pour démarrer une transaction par défaut :

```
if not IBDatabase1.DefaultTransaction.InTransaction then
  IBDatabase1.DefaultTransaction.StartTransaction;
```

Vous pouvez obtenir des transactions se chevauchant à l'aide de deux composants transaction séparés. Chaque composant transaction possède un ensemble de paramètres qui permettent de configurer les propriétés de la transaction. Ces paramètres vous permettent de spécifier le niveau d'isolement, ainsi que d'autres propriétés de la transaction.

Achèvement d'une transaction

Au mieux, une transaction ne doit durer que le temps nécessaire. Plus une transaction demeure active, plus d'utilisateurs peuvent simultanément accéder à la base de données. De même, plus nombreuses sont les transactions qui démarrent et s'achèvent simultanément pendant la durée de vie de votre transaction, plus grande est la probabilité pour que votre transaction entre en conflit avec une autre transaction lorsque vous essayez de valider vos modifications.

Achèvement d'une transaction réussie

Lorsque les actions qui composent la transaction ont toutes réussi, vous pouvez rendre définitives les modifications de la base de données en validant la transaction. Dans le cas de *TDatabase*, vous validez une transaction à l'aide de la méthode *Commit* :

```
MyOracleConnection.Commit;
```

Dans le cas de *TSQLConnection*, vous utilisez également la méthode *Commit*, mais vous devez spécifier la transaction à valider en fournissant le descripteur de transaction communiqué à la méthode *StartTransaction* :

```
MyOracleConnection.Commit(TD);
```

Dans le cas de *TIBDatabase*, vous validez un objet transaction à l'aide de sa méthode *Commit* :

```
IBDatabase1.DefaultTransaction.Commit;
```

Dans le cas de *TADOConnection*, vous validez une transaction à l'aide de la méthode *CommitTrans* :

```
ADOConnection1.CommitTrans;
```

Remarque Il est possible de valider une transaction imbriquée de sorte que seules les modifications ultérieures soient annulées si la transaction parent est annulée.

Une fois la transaction correctement validée, un composant connexion ADO reçoit un événement *OnCommitTransComplete*. Les autres composants connexion ne reçoivent pas d'événements similaires.

Un appel pour valider la transaction en cours est généralement tenté dans une instruction **try...except**. Ainsi, si une transaction ne peut pas être correctement validée, vous pouvez utiliser le bloc **except** pour traiter l'erreur et renouveler l'opération ou pour annuler la transaction.

Achèvement d'une transaction non réussie

Si une erreur se produit pendant l'application des modifications faisant partie de la transaction ou pendant la tentative de validation de la transaction, vous pouvez supprimer toutes les modifications qui composent la transaction. La suppression de ces modifications correspond à l'annulation de la transaction.

Dans le cas de *TDatabase*, vous annulez une transaction en appelant la méthode *Rollback* :

```
MyOracleConnection.Rollback;
```

Dans le cas de *TSQLConnection*, vous utilisez également la méthode *Rollback*, mais vous devez spécifier la transaction à annuler en fournissant le descripteur de transaction communiqué à la méthode *StartTransaction* :

```
MyOracleConnection.Rollback(TD);
```

Dans le cas de *TIBDatabase*, vous annulez un objet transaction en appelant sa méthode *Rollback* :

```
IBDatabase1.DefaultTransaction.Rollback;
```

Dans le cas de *TADOConnection*, vous annulez une transaction en appelant la méthode *RollbackTrans* :

```
ADOConnection1.RollbackTrans;
```

Une fois la transaction correctement annulée, un composant connexion ADO reçoit un événement *OnRollbackTransComplete*. Les autres composants connexion ne reçoivent pas d'événements similaires.

Un appel pour annuler la transaction en cours se produit généralement

- Dans un code de gestion des exceptions, lorsque vous ne pouvez pas rétablir la situation après une erreur de base de données.
- Dans un code d'événement de bouton ou de menu, comme lorsqu'un utilisateur clique sur un bouton Annuler.

Spécification du niveau d'isolement des transactions

Le niveau d'isolement des transactions détermine comment une transaction interagit avec d'autres transactions simultanées quand elles portent sur les mêmes tables. Il affecte, en particulier, ce qu'une transaction "voit" des changements apportés à une table par les autres transactions.

Chaque type de serveur prend en charge un ensemble différent de niveaux possibles d'isolement des transactions. Ces derniers sont au nombre de trois :

- *DirtyRead* : lorsque le niveau d'isolement a pour valeur *DirtyRead*, votre transaction voit toutes les modifications apportées par les autres transactions, même si elles n'ont pas été validées. Les changements non validés ne sont pas permanents et peuvent être annulés à tout moment. Cette valeur représente le niveau d'isolement le plus faible et n'est pas disponible pour de nombreux serveurs de bases de données (tels qu'Oracle, Sybase, MS-SQL et InterBase).
- *ReadCommitted* : lorsque le niveau d'isolement a pour valeur *ReadCommitted*, seuls les changements validés apportés par les autres sont visibles. Bien que ce paramètre empêche votre transaction de voir les changements non validés susceptibles d'être annulés, vous pouvez obtenir une vue incohérente de l'état de la base de données si une autre transaction est validée pendant le

processus de lecture. Ce niveau est disponible pour toutes les transactions à l'exception des transactions locales gérées par le BDE.

- *RepeatableRead* : lorsque le niveau d'isolement a pour valeur *RepeatableRead*, votre transaction est en mesure de voir un état cohérent des données de la base de données. Votre transaction voit une seule capture instantanée des données. Elle ne peut pas voir les changements de données ultérieurs apportés par d'autres transactions simultanées, mêmes si elles sont validées. Ce niveau d'isolement garantit qu'après la lecture d'un enregistrement par la transaction, la vue de cet enregistrement ne changera pas. C'est le niveau où votre transaction est la mieux isolée des changements apportés par d'autres transaction. Ce niveau n'est pas disponible sur certains serveurs, tels que Sybase et MS-SQL, ni pour les transactions locales gérées par le BDE.

En outre, *TSQLConnection* vous permet d'adapter les niveaux d'isolement aux bases de données. Les niveaux d'isolement personnalisés sont définis par le pilote *dbExpress*. Reportez-vous à la documentation du pilote pour plus d'informations.

Remarque Pour une description détaillée de la mise en œuvre de chaque niveau d'isolement, reportez-vous à la documentation de votre serveur.

TDatabase et *TADOConnection* vous permettent de spécifier le niveau d'isolement des transactions par le biais de la propriété *TransIsolation*. Lorsque vous attribuez à *TransIsolation* une valeur non prise en charge par le serveur de bases de données, vous obtenez le niveau d'isolement immédiatement supérieur (éventuellement disponible). En l'absence de niveau supérieur, le composant connexion déclenche une exception en cas de tentative de démarrage d'une transaction.

Lorsque vous utilisez *TSQLConnection*, le niveau d'isolement des transactions est contrôlé par le champ *IsolationLevel* du descripteur de transaction.

Lorsque vous utilisez InterBase Express, le niveau d'isolement des transactions est contrôlé par un paramètre de transaction.

Envoi de commandes au serveur

Tous les composants connexion de base de données, à l'exception de *TIBDatabase*, permettent d'exécuter des instructions SQL sur le serveur associé en appelant la méthode *Execute*. Bien que la méthode *Execute* puisse renvoyer un curseur lorsque l'instruction est une instruction SELECT, cette utilisation n'est pas recommandée. La meilleure façon d'exécuter des instructions qui renvoient des données consiste à utiliser un ensemble de données.

La méthode *Execute* est très pratique pour exécuter des instructions SQL simples qui ne renvoient pas d'enregistrements. Parmi ces instructions figurent les instructions DDL (Data Definition Language, langage de définition de données), qui manipulent ou créent des métadonnées de bases de données, telles que CREATE INDEX, ALTER TABLE et DROP DOMAIN. Certaines instructions SQL DML (Data Manipulation Language, langage de manipulation de données) ne

renvoient pas non plus d'ensemble de résultat. Les instructions DML réalisant une action sur des données mais ne renvoyant pas d'ensemble de résultat sont les suivantes : INSERT, DELETE et UPDATE.

La syntaxe de la méthode *Execute* dépend du type de connexion :

- Dans le cas de *TDatabase*, *Execute* possède quatre paramètres : une chaîne qui spécifie une instruction SQL unique à exécuter, un objet *TParams* qui fournit toutes les valeurs de paramètre de l'instruction, une valeur booléenne qui indique si l'instruction doit être placée en mémoire cache en vue d'un appel ultérieur et un pointeur vers un curseur BDE pouvant être renvoyé (il est recommandé d'indiquer nil).
- Dans le cas de *TADOConnection*, il existe deux versions de la méthode *Execute.* La première possède une valeur WideString qui spécifie l'instruction SQL et un second paramètre qui désigne un ensemble d'options qui déterminent si l'instruction est exécutée de façon asynchrone et si elle renvoie des enregistrements. Cette première syntaxe renvoie une interface pour les enregistrements renvoyés. La seconde syntaxe possède une valeur WideString qui spécifie l'instruction SQL, un deuxième paramètre qui renvoie le nombre d'enregistrements affectés par l'exécution de l'instruction et un troisième qui désigne des options, telles que l'exécution ou non de l'instruction de façon asynchrone. Aucune des syntaxes ne permet la transmission de paramètres.
- Dans le cas de *TSQLConnection*, *Execute* possède trois paramètres : une chaîne qui spécifie une instruction SQL unique à exécuter, un objet *TParams* qui fournit les valeurs de paramètre de cette instruction et un pointeur qui peut recevoir un objet *TCustomSQLDataSet* créé pour renvoyer les enregistrements.

Remarque *Execute* ne peut exécuter qu'une instruction SQL en même temps. A la différence des utilitaires de script SQL, un seul appel d'*Execute* ne permet pas d'exécuter plusieurs instructions SQL. Pour ce faire, appelez *Execute* autant de fois que nécessaire.

Il est relativement facile d'exécuter une instruction ne comprenant aucun paramètre. Par exemple, le code suivant exécute une instruction CREATE TABLE (DDL) sans aucun paramètre sur un composant *TSQLConnection* :

```
procedure TForm1.CreateTableButtonClick(Sender: TObject);
var
  SQLstmt: String;
begin
  SQLConnection1.Connected := True;
  SQLstmt := 'CREATE TABLE NewCusts ' +
    '( ' +
    '  CustNo INTEGER, ' +
    '  Company CHAR(40), ' +
    '  State CHAR(2), ' +
    '  PRIMARY KEY (CustNo) ' +
    ')';
  SQLConnection1.Execute(SQLstmt, nil, nil);
end;
```

Pour utiliser des paramètres, vous devez créer un objet *TParams*. Pour chaque valeur de paramètre, utilisez la méthode *TParams.CreateParam* afin d'ajouter un

objet *TParam*. Ensuite, utilisez les propriétés de *TParam* pour décrire le paramètre et définir sa valeur.

Ce processus est illustré dans l'exemple suivant, qui utilise *TDatabase* pour exécuter une instruction INSERT. L'instruction INSERT possède un paramètre unique nommé :*StateParam*. Un objet *TParams* (appelé *stmtParams*) est créé afin de fournir la valeur "CA" pour ce paramètre.

```
procedure TForm1.INSERT_WithParamsButtonClick(Sender: TObject);
var
  SQLstmt: String;
  stmtParams: TParams;
begin
  stmtParams := TParams.Create;
  try
    Database1.Connected := True;
    stmtParams.CreateParam(ftString, 'StateParam', ptInput);
    stmtParams.ParamByName('StateParam').AsString := 'CA';
    SQLstmt := 'INSERT INTO "Custom.db" '+
      '(CustNo, Company, State) ' +
      'VALUES (7777, "Robin Dabank Consulting", :StateParam)';
    Database1.Execute(SQLstmt, stmtParams, False, nil);
  finally
    stmtParams.Free;
  end;
end;
```

Si l'instruction SQL comprend un paramètre mais que vous ne fournissez pas d'objet *TParam* pour indiquer sa valeur, l'instruction SQL peut générer une erreur lors de son exécution (cela dépend du type de la base de données utilisée). Si un objet *TParam* est fourni alors qu'aucun paramètre ne lui correspond dans l'instruction SQL, une exception est déclenchée lorsque l'application tente d'utiliser le *TParam*.

Utilisation d'ensembles de données associés

Tous les composants connexion de base de données gèrent une liste de tous les ensembles de données qui les utilisent pour se connecter à une base de données. Par exemple, un composant connexion utilise cette liste pour fermer tous les ensembles de données lorsqu'il ferme la connexion à la base de données.

Vous pouvez également utiliser cette liste pour effectuer des opérations sur tous les ensembles de données utilisant un composant connexion spécifique pour se connecter à une base de données particulière.

Fermeture d'ensembles de données sans déconnexion du serveur

Le composant connexion ferme automatiquement tous les ensembles de données lorsque vous fermez sa connexion. Toutefois, les ensembles de données doivent parfois être fermés sans pour autant mettre fin à la connexion au serveur de base de données.

Pour fermer tous les ensembles de données ouverts sans provoquer la déconnexion du serveur, vous pouvez utiliser la méthode *CloseDataSets*.

Dans le cas de *TADOConnection* et *TIBDatabase*, l'appel de *CloseDataSets* laisse toujours la connexion ouverte. Dans le cas de *TDatabase* et *TSQLConnection*, vous devez également attribuer à la propriété *KeepConnection* la valeur *True*.

Déplacement parmi les ensembles de données associés

Pour effectuer des actions (autres que leur fermeture) sur tous les ensembles de données qui utilisent un composant connexion, utilisez les propriétés *DataSets* et *DataSetCount*. *DataSets* est un tableau indicé de tous les ensembles de données liés au composant connexion. Pour tous les composants connexion, à l'except de *TADOConnection*, cette liste ne comprend que les ensembles de données actifs. *TADOConnection* répertorie également les ensembles de données inactifs. *DataSetCount* représente le nombre d'ensembles de données dans le tableau.

Remarque Lorsque vous utilisez un ensemble de données client spécialisé pour placer les mises à jour en mémoire cache (par opposition à l'ensemble de données client générique *TClientDataSet*), la propriété *DataSets* comprend l'ensemble de données interne détenu par l'ensemble de données client, non celui-ci.

Vous pouvez utiliser *DataSets* avec *DataSetCount* pour effectuer un cycle sur tous les ensembles de données actuellement actifs depuis votre code. Par exemple, le code suivant parcourt tous les ensembles de données actifs et désactive tous les contrôles qui utilisent les données qu'ils fournissent :

```
var
  I: Integer;
begin
  with MyDBConnection do
  begin
    for I := 0 to DataSetCount - 1 do
      DataSets[I].DisableControls;
  end;
end;
```

Remarque *TADOConnection* gère les objets commande ainsi que les ensembles de données. A l'image des ensembles de données, vous pouvez parcourir les objets commande, à l'aide des propriétés *Commands* et *CommandCount*.

Obtention de métadonnées

Tous les composants connexion de base de données peuvent extraire du serveur de bases de données des listes de métadonnées, bien que le type des métadonnées extraites est variable. Les méthodes qui extraient les métadonnées remplissent une liste de chaînes avec les noms de différentes entités disponibles sur le serveur. Vous pouvez alors utiliser ces informations pour, par exemple, permettre aux utilisateurs de sélectionner une table dynamiquement à l'exécution.

Vous pouvez utiliser un composant *TADOConnection* pour extraire les métadonnées des tables et procédures stockées disponibles sur le stockage de données ADO. Vous pouvez alors utiliser ces informations pour, par exemple, permettre aux utilisateurs de sélectionner une table ou procédure stockée dynamiquement à l'exécution.

Enumération des tables disponibles

La méthode *GetTableNames*copie une liste des noms de tables dans un objet liste de chaînes existant. Cela permet, par exemple, de remplir une boîte liste avec des noms de table, grâce à laquelle l'utilisateur pourra choisir la table qu'il souhaite ouvrir. La ligne suivante remplit une boîte liste avec le nom de toutes les tables de la base de données :

```
MyDBConnection.GetTableNames(ListBox1.Items, False);
```

GetTableNames possède deux paramètres : la liste de chaînes à remplir avec les noms de table et une valeur booléenne qui indique si la liste doit comprendre les tables système ou les tables ordinaires. Notez que, tous les serveurs n'utilisant pas des tables système pour stocker les métadonnées, la demande de tables système peut aboutir à la génération d'une liste vide.

Remarque Pour la plupart des composants connexion de base de données, *GetTableNames* renvoie la liste de toutes les tables non-système disponibles lorsque le second paramètre a pour valeur *False*. Dans le cas de *TSQLConnection*, toutefois, vous contrôlez plus facilement les types ajoutés à la liste lorsque vous ne récupérez pas uniquement les noms de tables système. Lorsque vous utilisez *TSQLConnection*, les types de noms ajoutés à la liste sont contrôlés par la propriété *TableScope*. *TableScope* indique si la liste doit contenir tout ou partie des éléments suivants : tables ordinaires, tables système, synonymes et vues.

Enumération des champs d'une table

La méthode *GetFieldNames*remplit une liste de chaînes existante avec les noms de tous les champs (colonnes) d'une table spécifiée. *GetFieldNames* possède deux paramètres, le nom de la table dont vous souhaitez énumérer les champs et une liste de chaînes existante à remplir avec les noms de champ :

```
MyDBConnection.GetFieldNames('Employee', ListBox1.Items);
```

Enumération des procédures stockées disponibles

Pour obtenir une liste de toutes les procédures stockées de la base de données, utilisez la méthode *GetProcedureNames* Cette méthode accepte un seul paramètre, une liste de chaînes existante à remplir :

```
MyDBConnection.GetProcedureNames(ListBox1.Items);
```

Remarque *GetProcedureNames* est uniquement disponible pour *TADOConnection* et *TSQLConnection*.

Enumération des index disponibles

Pour obtenir la liste de tous les index d'une table spécifique, utilisez la méthode *GetIndexNames*. Cette méthode prend en compte deux paramètres, la table dont vous souhaitez connaître les index et une liste de chaînes existante à remplir :

```
SQLConnection1.GetIndexNames('Employee', ListBox1.Items);
```

Remarque *GetIndexNames* est uniquement disponible pour *TSQLConnection*, bien que la plupart des ensembles de type table possèdent une méthode équivalente.

Enumération des paramètres de procédure stockée

Pour obtenir la liste de tous les paramètres d'une procédure stockée spécifique, utilisez la méthode *GetProcedureParams*. *GetProcedureParams* remplit un objet *TList* avec les pointeurs des enregistrements des descriptions de paramètres, dans lequel chaque enregistrement décrit un paramètre d'une procédure stockée spécifiée, notamment ses nom, index, type de paramètre, type de champ, etc.

GetProcedureParams accepte deux paramètres, le nom de la procédure stockée et un objet *TList* existant :

```
SQLConnection1.GetProcedureParams('GetInterestRate', List1);
```

Vous pouvez convertir les descriptions de paramètre ajoutées à la liste en l'objet plus courant *TParams* en appelant la procédure globale *LoadParamListItems* *GetProcedureParams* allouant dynamiquement les différents enregistrements, votre application doit les libérer lorsqu'elle n'a plus besoin des informations. La routine globale *FreeProcParams* peut effectuer cette opération.

Remarque *GetProcedureParams* est uniquement disponible pour *TSQLConnection*.

Chapitre

18

Présentation des ensembles de données

L'unité fondamentale pour accéder aux données est la famille d'objets ensemble de données. Les applications utilisent des ensembles de données pour tous les accès aux bases de données. Un objet ensemble de données représente un ensemble d'enregistrements d'une base de données organisé en une table logique. Ces enregistrements peuvent être issus d'une seule table de base de données ou peuvent représenter les résultats de l'exécution d'une requête ou d'une procédure stockée.

Tous les objets ensemble de données utilisés dans des applications de bases de données descendent de *TDataSet* et héritent des champs de données, propriétés, méthodes et événements de cette classe. Ce chapitre décrit les fonctionnalités de *TDataSet* héritées par les objets ensemble de données utilisés dans les applications de bases de données. Avant d'utiliser tout objet de base de données, vous devez avoir assimilé ces fonctionnalités partagées.

TDataSet est un ensemble de données virtuel, ce qui signifie que la plupart de ses propriétés et méthodes sont déclarées comme **virtual** ou **abstract**. Une *méthode virtuelle* est une déclaration de fonction ou de procédure dont l'implémentation est redéfinie par les objets descendants. Une *méthode abstract* est une fonction ou une procédure sans véritable implémentation. La déclaration est un prototype qui décrit une méthode (ainsi que ses paramètres et le type renvoyé, le cas échéant) devant être implémentée dans tous les objets ensemble de données descendants (l'implémentation peut être différente pour chaque ensemble de données).

Etant donné que *TDataSet* contient des méthodes **abstract**, vous ne pouvez pas l'utiliser directement dans une application sans générer d'erreur d'exécution. A la place, soit vous créez des instances des descendants de *TDataSet* intégrés et les utilisez dans votre application, soit vous dérivez votre propre objet ensemble de données depuis *TDataSet* ou depuis ses descendants et écrivez des implémentations pour toutes ses méthodes **abstract**.

TDataSet définit la plupart des fonctionnalités communes à tous les objets ensemble de données. Ainsi, il définit la structure de base de tous les ensembles de données : un tableau de composants *TField* qui correspond aux colonnes d'une ou de plusieurs tables de bases de données, aux champs de référence ou aux champs calculés fournis par votre application. Pour plus d'informations sur les composants *TField*, voir chapitre 19, "Manipulation des composants champ".

Ce chapitre décrit l'utilisation des fonctionnalités de bases de données courantes associées à *TDataSet*. Gardez cependant à l'esprit que, bien que *TDataSet* introduise les méthodes liées à ces fonctionnalités, tous les objets dépendants de *TDataSet* ne les implémentent pas. En particulier, les ensembles de données unidirectionnels n'en implémentent qu'un sous-ensemble limité.

Utilisation des descendants de TDataSet

TDataSet a plusieurs descendants immédiats, chacun correspondant à un différent mécanisme d'accès aux données. Vous ne travaillez pas directement avec ces descendants. Au lieu de cela, chaque descendant introduit les propriétés et les méthodes permettant d'utiliser un mécanisme d'accès aux données particulier. Ces propriétés et méthodes sont alors exposées par les classes descendantes adaptées aux différents types de données serveur. Les descendants immédiats de *TDataSet* sont :

- *TBDEDataSet*, qui utilise le moteur BDE (Borland Database Engine) pour communiquer avec le serveur de base de données. Les descendants de *TBDEDataSet* que vous utilisez sont *TTable*, *TQuery*, *TStoredProc* et *TNestedTable*. Les fonctionnalités uniques des ensembles de données basés sur le BDE sont décrites dans le chapitre 20, "Utilisation du moteur de bases de données Borland".
- *TCustomADODataSet*, qui utilise les ADO (objets de données ActiveX) pour communiquer avec un datastore OLEDB. Les descendants de *TCustomADODataSet* que vous utilisez sont *TADODataSet*, *TADOTable*, *TADOQuery* et *TADOStoredProc*. Les fonctionnalités uniques des ensembles de données basés sur les ADO sont décrites dans le chapitre 21, "Utilisation des composants ADO".
- *TCustomSQLDataSet*, qui utilise dbExpress pour communiquer avec un serveur de base de données. Les descendants de *TCustomSQLDataSet* que vous utilisez sont *TSQLDataSet*, *TSQLTable*, *TSQLQuery* et *TSQLStoredProc*. Les fonctionnalités uniques des ensembles de données dbExpress sont décrites dans le chapitre 22, "Utilisation d'ensembles de données unidirectionnels".
- *TIBCustomDataSet*, qui communique directement avec un serveur de base de données InterBase. Les descendants de *TIBCustomDataSet* que vous utilisez sont *TIBDataSet*, *TIBTable*, *TIBQuery* et *TIBStoredProc*.
- *TCustomClientDataSet*, qui représente les données d'un autre composant ensemble de données ou les données d'un fichier dédié sur disque. Les descendants de *TCustomClientDataSet* que vous utilisez sont *TClientDataSet*, qui peut se connecter à un ensemble de données externe (source), et les ensembles

de données client spécifiques à un mécanisme d'accès aux données (*TBDEClientDataSet, TSQLClientDataSet* et *TIBClientDataSet*), qui utilisent un ensemble de données source interne. Les fonctionnalités uniques des ensembles de données client sont décrites dans le chapitre 23, "Utilisation d'ensembles de données client".

Les avantages et inconvénients des divers mécanismes d'accès aux données employés par ces descendants de *TDataSet* sont décrits dans "Utilisation des bases de données" à la page 14-1.

En plus des ensembles de données intégrés, vous pouvez créer vos propres descendants *TDataSet* personnalisés, par exemple pour fournir des données à partir d'un processus autre qu'un serveur de base de données, comme une feuille de calcul. L'écriture d'ensembles de données personnalisés vous offre la souplesse de gérer les données avec la méthode de votre choix tout en continuant d'utiliser les contrôles données de la VCL pour construire votre interface utilisateur. Pour plus d'informations sur la création de composants personnalisés, voir chapitre 40, "Présentation générale de la création d'un composant".

Bien que chaque descendant de *TDataSet* ait ses propres propriétés et méthodes uniques, certaines des propriétés et méthodes introduites par les classes descendantes sont les mêmes que celles introduites par d'autres classes descendantes utilisant un autre mécanisme d'accès aux données. Par exemple, il y a des similitudes entre les divers composants "table" (*TTable*, *TADOTable*, *TSQLTable* et *TIBTable*). Pour plus d'informations sur les similitudes introduites par les descendants de *TDataSet*, voir "Types d'ensembles de données" à la page 18-27.

Détermination des états d'un ensemble de données

L'*état*, ou le *mode*, d'un ensemble de données détermine les opérations possibles sur ses données. Par exemple, quand un ensemble de données est fermé, son état devient *dsInactive*, ce qui signifie que rien ne peut affecter ses données. Au moment de l'exécution, vous pouvez examiner la propriété en lecture seule *State* de l'ensemble de données, pour déterminer son état en cours. Le tableau suivant récapitule les valeurs possibles de la propriété *State* et leur signification :

Tableau 18.1 Valeurs possibles pour la propriété State des ensembles de données

Valeur	Etat	Signification
dsInactive	Inactif	L'ensemble de données est fermé. Ses données sont indisponibles.
dsBrowse	Visualisation	L'ensemble de données est ouvert. Ses données peuvent être vues mais pas modifiées. C'est l'état par défaut d'un ensemble de données ouvert.
dsEdit	Edition	L'ensemble de données est ouvert. La ligne en cours peut être modifiée. (non supporté par les ensembles de données unidirectionnels)

Tableau 18.1 Valeurs possibles pour la propriété State des ensembles de données (suite)

Valeur	Etat	Signification
dsInsert	Insertion	L'ensemble de données est ouvert. Une nouvelle ligne peut être insérée. (non supporté par les ensembles de données unidirectionnels)
dsSetKey	Indexation (*SetKey*)	L'ensemble de données est ouvert. Active la définition de portées et de valeurs clé pour les opérations portant sur des portées et les opérations *GotoKey*. (non supporté par tous les ensembles de données)
dsCalcFields	Champs calculés (*CalcFields*)	L'ensemble de données est ouvert. Indique qu'un événement *OnCalcFields* est en cours. Interdit toute modification de champ non calculé.
dsCurValue	CurValue	L'ensemble de données est ouvert. Indique que la propriété CurValue des champs est lue par un gestionnaire d'événement qui répond aux erreurs en appliquant des mises à jour en mémoire cache.
dsNewValue	NewValue	L'ensemble de données est ouvert. Indique que la propriété NewValue des champs est lue par un gestionnaire d'événement qui répond aux erreurs en appliquant des mises à jour en mémoire cache.
dsOldValue	OldValue	L'ensemble de données est ouvert. Indique que la propriété OldValue des champs est lue par un gestionnaire d'événement qui répond aux erreurs en appliquant des mises à jour en mémoire cache.
dsFilter	Filtrage	L'ensemble de données est ouvert. Indique qu'une opération de filtrage est en cours. Un ensemble de données restreint peut être visualisé, sans qu'aucune donnée ne puisse être changée. (non supporté par les ensembles de données unidirectionnels)
dsBlockRead	Lecture de bloc	L'ensemble de données est ouvert. Les contrôles orientés données ne sont pas mis à jour et les événements ne sont pas déclenchés lorsque l'enregistrement en cours est modifié.
dsInternalCalc	Calcul interne	L'ensemble de données est ouvert. Un événement *OnCalcFields* est en cours pour des valeurs calculées stockées avec l'enregistrement. (uniquement pour les ensembles de données client)
dsOpening	Ouverture	L'ensemble de données est dans le processus d'ouverture, mais n'a pas fini. Cet état arrive quand l'ensemble de données est ouvert pour une lecture asynchrone.

En général, une application vérifie l'état de l'ensemble de données pour déterminer le moment d'effectuer certaines opérations. Par exemple, vous pouvez chercher l'état *dsEdit* ou *dsInsert* pour savoir s'il faut valider les mises à jour.

Remarque A chaque fois que l'état d'un ensemble de données change, l'événement *OnStateChange* est appelé pour tous les composants source de données associés. Pour plus d'informations concernant les composants source de données et sur *OnStateChange*, voir "Réponse aux modifications effectuées par le biais de la source de données" à la page 15-5.

Ouverture et fermeture des ensembles de données

Pour lire ou écrire des données dans un ensemble de données, une application doit d'abord l'ouvrir. Il y a deux moyens de procéder,

- Définir la propriété *Active* de l'ensemble de données par *True*, soit dans l'inspecteur d'objets à la conception, soit dans le code à l'exécution :

```
CustTable.Active := True;
```

- Appeler la méthode *Open* de l'ensemble de données au moment de l'exécution,

```
CustQuery.Open;
```

Quand vous ouvrez l'ensemble de données, celui-ci reçoit d'abord un événement *BeforeOpen*, puis il ouvre un curseur, se remplit lui-même de données et, enfin, reçoit un événement *AfterOpen*.

L'ensemble de données qui vient d'être ouvert est en mode visualisation, ce qui veut dire que votre application peut lire des données et naviguer dans cet ensemble de données.

Il y a deux façons de fermer un ensemble de données,

- Définir la propriété *Active* de l'ensemble de données par *False*, soit dans l'inspecteur d'objets à la conception, soit dans le code à l'exécution :

```
CustQuery.Active := False;
```

- Appeler la méthode *Close* de l'ensemble de données à l'exécution,

```
CustTable.Close;
```

De même que l'ensemble de données reçoit les événements *BeforeOpen* et *AfterOpen* quand vous l'ouvrez, il reçoit les événements *BeforeClose* et *AfterClose* quand vous le fermez. Ces gestionnaires répondent à la méthode *Close* d'un ensemble de données. Vous pouvez utiliser ces événements, par exemple, pour inviter l'utilisateur à valider les modifications en attente ou à les annuler avant de fermer l'ensemble de données. Le code suivant illustre un tel gestionnaire :

```
procedure TForm1.CustTableVerifyBeforeClose(DataSet: TDataSet);
begin
  if (CustTable.State in [dsEdit, dsInsert]) then begin
    case MessageDlg('Emettre les modifications avant de fermer?',
mtConfirmation, mbYesNoCancel, 0) of
      mrYes:    CustTable.Post;   { enregistrer les modifications }
      mrNo:     CustTable.Cancel; { abandonner les modifications }
      mrCancel: Abort;            { annuler la fermeture de l'ensemble de données }
    end;
  end;
end;
```

Remarque Il est parfois nécessaire de fermer au préalable l'ensemble de données pour changer certaines de ses propriétés, comme la propriété *TableName* sur un composant *TTable*. Lorsque vous rouvrez l'ensemble de données, la nouvelle valeur de propriété prend effet.

Navigation dans les ensembles de données

Chaque ensemble de données actif dispose d'un *curseur* qui pointe sur la ligne en cours dans l'ensemble de données. Ce sont les valeurs de champ de cette *ligne en cours* qui apparaissent dans les contrôles à champ unique, orientés données d'une fiche, comme *TDBEdit*, *TDBLabel* et *TDBMemo*. Si l'ensemble de données prend en charge l'édition, l'enregistrement en cours contient les valeurs qui sont manipulables par des méthodes d'édition, d'insertion et de suppression.

Vous pouvez changer de ligne en cours en déplaçant le curseur pour le faire pointer sur une autre ligne. Le tableau suivant dresse la liste des méthodes pouvant être utilisées dans le code d'une application, pour se déplacer vers d'autres enregistrements.

Tableau 18.2 Méthodes de navigation relatives aux ensembles de données

Méthodes	Déplace le curseur vers
First	La première ligne d'un ensemble de données.
Last	La dernière ligne d'un ensemble de données. (non disponible pour les ensembles de données unidirectionnels)
Next	La ligne suivante d'un ensemble de données.
Prior	La ligne précédente d'un ensemble de données. (non disponible pour les ensembles de données unidirectionnels)
MoveBy	Le nombre spécifié de lignes en avant ou en arrière d'un ensemble de données.

Le composant visuel, orienté données, *TDBNavigator*, encapsule ces méthodes sous la forme de boutons sur lesquels l'utilisateur peut cliquer pour se déplacer parmi les enregistrements lors de l'exécution. Pour plus d'informations sur le composant navigateur, voir "Navigation et manipulation d'enregistrements" à la page 15-33.

Chaque fois que vous changez l'enregistrement en cours à l'aide d'une de ces méthodes (ou d'une autre méthode de navigation basée sur un critère de recherche), l'ensemble de données reçoit deux événements : *BeforeScroll* (avant de quitter l'enregistrement en cours) et *AfterScroll* (après avoir atteint le nouvel enregistrement). Vous pouvez utiliser ces événements pour mettre à jour votre interface utilisateur (par exemple, pour mettre à jour une barre d'état qui donne des informations sur l'enregistrement en cours).

TDataSet définit deux propriétés booléennes qui donnent des indications utiles pour parcourir les enregistrements d'un ensemble de données.

Tableau 18.3 Propriétés de navigation des ensembles de données

Propriété	Description
Bof (Début de fichier)	*True* : le curseur se trouve sur la première ligne de l'ensemble de données.
	False : le curseur n'est pas répertorié comme étant sur la première ligne de l'ensemble de données
Eof (Fin de fichier)	*True* : le curseur se trouve sur la dernière ligne de l'ensemble de données.
	False : le curseur n'est pas répertorié comme étant sur la première ligne de l'ensemble de données

Utilisation des méthodes First et Last

La méthode *First* place le curseur sur la première ligne d'un ensemble de données et définit la propriété *Bof* par *True*. Si le curseur est déjà sur la première ligne, *First* n'a aucun effet.

Par exemple, le code suivant permet d'aller sur le premier enregistrement de *CustTable* :

```
CustTable.First;
```

La méthode *Last* place le curseur sur la dernière ligne d'un ensemble de données et définit la propriété *Eof* par *True*. Si le curseur est déjà sur la dernière ligne, *Last* n'a aucun effet.

Le code suivant permet d'aller sur le dernier enregistrement de *CustTable* :

```
CustTable.Last;
```

Remarque La méthode *Last* déclenche une exception dans les ensembles de données unidirectionnels.

Conseil Bien qu'il puisse exister de nombreuses raisons pour aller sur la première ou sur la dernière ligne d'un ensemble de données sans que l'utilisateur n'intervienne, vous pouvez offrir à ce dernier la possibilité de naviguer parmi les enregistrements en utilisant le composant *TDBNavigator*. Le composant navigateur contient des boutons qui, s'ils sont actifs et visibles, permettent à l'utilisateur d'aller sur la première ou la dernière ligne de l'ensemble de données actif. Les événements *OnClick* de ces boutons appellent les méthodes *First* et *Last* de l'ensemble de données. Pour plus d'informations concernant le fonctionnement du composant navigateur, voir "Navigation et manipulation d'enregistrements" à la page 15-33.

Utilisation des méthodes Next et Prior

La méthode *Next* déplace le curseur d'une ligne en avant dans l'ensemble de données et définit la propriété *Bof* par *False* si l'ensemble de données n'est pas vide. Si le curseur se trouve déjà sur la dernière ligne de l'ensemble de données, lorsque vous appelez *Next*, celle-ci n'a aucun effet.

Par exemple, le code suivant provoque le déplacement du curseur sur le prochain enregistrement de *CustTable* :

```
CustTable.Next;
```

La méthode *Prior* déplace le curseur d'une ligne en arrière dans l'ensemble de données et définit la propriété *Eof* par *False* si l'ensemble de données n'est pas vide. Si le curseur se trouve déjà sur la première ligne dans l'ensemble de données, lorsque vous appelez *Prior*, celle-ci n'a aucun effet.

Par exemple, le code suivant permet d'aller sur l'enregistrement précédent de *CustTable* :

```
CustTable.Prior;
```

Remarque La méthode *Prior* déclenche une exception dans les ensembles de données unidirectionnels.

Utilisation de la méthode MoveBy

MoveBy vous permet de spécifier le nombre de lignes du déplacement du curseur dans l'ensemble de données, vers l'avant ou vers l'arrière. Le déplacement se fait par rapport à la position de l'enregistrement en cours au moment où *MoveBy* est appelée. *MoveBy* définit également les propriétés *Bof* et *Eof* de l'ensemble de données.

Cette fonction accepte un paramètre entier qui indique le nombre d'enregistrements parcourus lors du déplacement. Un entier positif indique un déplacement vers l'avant et un entier négatif, un déplacement vers l'arrière.

Remarque La méthode *MoveBy* déclenche une exception dans les ensembles de données unidirectionnels si vous utilisez un argument négatif.

MoveBy renvoie le nombre de lignes effectivement parcourues. Si le déplacement voulu va au-delà du début ou de la fin de l'ensemble de données, le nombre de lignes renvoyées par *MoveBy* sera différent de celui voulu pour le déplacement. C'est parce que *MoveBy* s'arrête quand il atteint le premier ou le dernier enregistrement de l'ensemble de données.

Le code suivant provoque un déplacement de deux enregistrements vers l'arrière dans *CustTable* :

```
CustTable.MoveBy(-2);
```

Remarque Si votre application utilise *MoveBy* dans un environnement de base de données multi-utilisateur, ne perdez pas de vue que les ensembles de données sont fluides. Un enregistrement qui se trouvait cinq enregistrements en arrière il y a un instant, peut se retrouver maintenant quatre, six ou même un nombre

d'enregistrements inconnu en arrière si plusieurs utilisateurs accèdent simultanément à la base de données et font des modifications.

Utilisation des propriétés Eof et Bof

Les deux propriétés *Eof* (fin de fichier) et *Bof* (début de fichier), accessibles en lecture et à l'exécution uniquement, sont utiles pour parcourir tous les enregistrements d'un ensemble de données.

Eof

Quand la valeur de *Eof* est *True*, cela indique que le curseur se trouve sans équivoque sur la dernière ligne de l'ensemble de données. *Eof* passe à *True* quand une application :

- Ouvre un ensemble de données vide.
- Réussit à appeler la méthode *Last* de l'ensemble de données.
- Appelle la méthode *Next* de l'ensemble de données et que son exécution échoue (car le curseur se trouve déjà sur la dernière ligne de l'ensemble de données).
- Appelle *SetRange* sur une portée ou un ensemble de données vide.

Eof vaut *False* dans tous les autres cas ; vous devez supposer que *Eof* vaut *False* sauf si l'une des conditions ci-dessus est vérifiée *et* si vous avez testé directement la valeur de la propriété.

Le test sur *Eof* se fait généralement dans une condition de boucle pour contrôler le processus itératif sur tous les enregistrements d'un ensemble de données. Si vous ouvrez un ensemble de données contenant plusieurs enregistrements (ou si vous appelez *First*) *Eof* vaut *False*. Pour parcourir un à un les enregistrements d'un ensemble de données, vous devez créer une boucle qui avance sur chaque enregistrement en appelant *Next*, et se termine quand *Eof* vaut *True*. *Eof* reste à *False* jusqu'à ce que vous appeliez *Next* alors que le curseur se trouve déjà sur le dernier enregistrement.

L'exemple suivant montre l'une des façons de programmer une boucle de traitement d'enregistrements pour un ensemble de données appelé *CustTable* :

```
CustTable.DisableControls;
try
  CustTable.First; { Se positionne sur le premier enregistrement ; Eof devient False }
  while not CustTable.Eof do { boucle jusqu'à ce que Eof devienne True }
  begin
    { Le traitement de l'enregistrement se fait ici }
    ⋮
    CustTable.Next; { Eof vaut false en cas de réussite; Eof vaut True si Next
                      échoue sur le dernier enregistrement}
  end;
finally
  CustTable.EnableControls;
end;
```

Conseil Cet exemple montre aussi comment désactiver puis réactiver des contrôles visuels, orientés données, rattachés à l'ensemble de données. Si vous désactivez les contrôles visuels pendant la durée de l'itération sur l'ensemble de données, le traitement sera accéléré car votre application n'a pas à mettre à jour le contenu des contrôles au fur et à mesure de l'évolution de l'enregistrement en cours. Une fois l'itération achevée, les contrôles doivent être réactivés pour être mis à jour en fonction de la nouvelle ligne en cours. Notez que l'activation de contrôles visuels a lieu dans la clause **finally** d'une instruction **try...finally**. Cela garantit que les contrôles ne resteront pas désactivés, même si une exception termine le traitement de la boucle de façon prématurée.

Bof

Quand la valeur de *Bof* est *True*, cela indique que le curseur se trouve sans équivoque sur la première ligne de l'ensemble de données. *Bof* est mis à *True* lorsqu'une application :

- Ouvre un ensemble de données.
- Appelle la méthode *First* de l'ensemble de données.
- Appelle la méthode *Prior* de l'ensemble de données et que son exécution échoue (car le curseur se trouve déjà sur la première ligne de l'ensemble de données).
- Appelle *SetRange* sur une portée ou un ensemble de données vide.

Bof prend la valeur *False* dans tous les autres cas ; vous devez supposer que *Bof* vaut *False* sauf si l'une des conditions ci-dessus est vérifiée *et* si vous avez testé directement la valeur de la propriété.

Comme *Eof*, *Bof* peut se trouver dans une condition de boucle pour contrôler un processus itératif sur des enregistrements d'un ensemble de données. L'exemple suivant montre l'une des façons de programmer une boucle de traitement d'enregistrements pour un ensemble de données appelé *CustTable*: :

```
CustTable.DisableControls; { accélère le traitement et empêche les
                             rafraîchissements écran }
try
  while not CustTable.Bof do { boucle jusqu'à ce que Bof devienne True }
  begin
    { Le traitement de l'enregistrement se fait ici }
    ⋮
    CustTable.Prior; { Bof vaut false en cas de réussite; Bof vaut True si Prior
                       échoue sur le premier enregistrement }
  end;
finally
  CustTable.EnableControls; { affiche la nouvelle ligne en cours dans les contrôles }
end;
```

Marquage d'enregistrements

Outre la possibilité de se déplacer d'un enregistrement à un autre dans un ensemble de données (ou de se déplacer selon un nombre déterminé d'enregistrements), il est possible de marquer un emplacement particulier dans un ensemble de données de façon à y revenir rapidement le moment voulu. *TDataSet* introduit une fonction de marquage constituée d'une propriété *Bookmark* et de cinq méthodes de définition de signets.

TDataSet implémente des méthodes virtuelles de gestion des signets. Bien que ces méthodes garantissent que chaque objet ensemble de données dérivé de *TDataSet* renvoie une valeur si une méthode de signet est appelée, les valeurs renvoyées sont simplement des valeurs par défaut qui n'indiquent pas la position en cours. Les descendants de *TDataSet* ont des niveaux de support de signets différents. Les ensembles de données dbExpress ne supportent pas les signets. Les ensembles de données ADO peuvent prendre en charge les signets, en fonction des tables de la base de données sous-jacente. Les ensembles de données BDE, InterBase express et client prennent toujours en charge les signets.

La propriété Bookmark

La propriété *Bookmark* indique quel est le signet en cours dans votre application. *Bookmark* est une chaîne qui identifie le signet en cours. Tout nouveau signet ajouté devient le signet en cours.

La méthode GetBookmark

Pour créer un signet, vous devez déclarer une variable de type *TBookmark* dans votre application, puis appeler *GetBookmark* pour allouer un espace de stockage à la variable et définir sa valeur par un emplacement particulier dans l'ensemble de données. Le type *TBookmark* est un pointeur.

Les méthodes GotoBookmark et BookmarkValid

Lorsqu'un signet lui est transmis, *GotoBookmark* déplace le curseur de l'ensemble de données à l'emplacement du signet. Avant d'appeler *GotoBookmark*, vous pouvez appeler *BookmarkValid* pour déterminer si le signet pointe sur un enregistrement. *BookmarkValid* renvoie *True* si le signet pointe sur un enregistrement.

La méthode CompareBookmarks

Vous pouvez aussi appeler *CompareBookmarks* pour voir si un signet sur lequel vous voulez vous déplacer est différent d'un autre signet ou du signet en cours. Si les deux signets font référence au même enregistrement (ou si tous les deux sont **nil**), *CompareBookmarks* renvoie 0.

La méthode FreeBookmark

FreeBookmark restitue la mémoire allouée à un signet lorsqu'il n'est plus nécessaire. Vous devez également appeler *FreeBookmark* avant de réutiliser un signet existant.

Un exemple d'utilisation de signets

Le code suivant illustre l'utilisation des signets :

```
procedure DoSomething (const Tbl: TTable)
var
  Bookmark: TBookmark;
begin
  Bookmark := Tbl.GetBookmark; { alloue la mémoire et affecte une valeur }
  Tbl.DisableControls; { désactive l'affichage des enregistrements
                         dans les contrôles orientés données }
  try
    Tbl.First; { se déplace sur le premier enregistrement de la table }
    while not Tbl.Eof do {parcourt tous les enregistrements de la table }
    begin
      { insérez ici votre traitement }
      ⋮
      Tbl.Next;
    end;
  finally
    Tbl.GotoBookmark(Bookmark);
    Tbl.EnableControls; { réactive l'affichage des enregistrements dans les
                          contrôles orientés données, si nécessaire }
    Tbl.FreeBookmark(Bookmark); {restitue la mémoire allouée au signet }
  end;
end;
```

Avant d'entrer dans le processus de parcours des enregistrements, les contrôles sont désactivés. Même si une erreur se produit durant le balayage des enregistrements, la clause **finally** permet d'être sûr que les contrôles seront toujours réactivés et que le signet sera toujours restitué, même si la boucle se termine prématurément.

Recherche dans les ensembles de données

Si un ensemble de données n'est pas unidirectionnel, vous pouvez effectuer la recherche en utilisant les méthodes *Locate* et *Lookup*. Ces méthodes autorisent des recherches dans tout type de colonne et dans tout ensemble de données.

Remarque Certains descendants de *TDataSet* offrent une famille supplémentaire de méthodes de recherche basées sur un index. Pour plus d'informations sur ces méthodes supplémentaires, voir "Utilisation d'index pour chercher des enregistrements" à la page 18-32.

Utilisation de la méthode Locate

Locate déplace le curseur sur la première ligne correspondant au critère de recherche spécifié. Dans sa forme la plus simple, vous transmettez à *Locate* le nom de la colonne de recherche, une valeur de champ pour établir la correspondance et un indicateur d'option qui spécifie si la recherche doit tenir compte des différences majuscules/minuscules et si elle utilise les

correspondances de clés partielles. (Correspondances dans lesquelles la chaîne de critère peut se limiter à un préfixe de la valeur de champ.) Par exemple, le code suivant déplace le curseur sur la première ligne de *CustTable* pour laquelle la valeur dans la colonne *Company* est "Professional Divers, Ltd." :

```
var
  LocateSuccess: Boolean;
  SearchOptions: TLocateOptions;
begin
  SearchOptions := [loPartialKey];
  LocateSuccess := CustTable.Locate('Company', 'Professional Divers, Ltd.', SearchOptions);
end;
```

Si *Locate* trouve une correspondance, le premier enregistrement contenant cette correspondance devient l'enregistrement en cours. *Locate* renvoie *True* si une correspondance est trouvée et *False* dans le cas contraire. Si la recherche échoue, l'enregistrement en cours reste le même.

Le vrai potentiel de *Locate* se manifeste quand vous effectuez une recherche sur plusieurs colonnes et que vous spécifiez plusieurs valeurs recherchées. Celles-ci sont des Variants, et vous pouvez donc spécifier des types de données différents pour vos critères de recherche. Pour spécifier plusieurs colonnes dans une chaîne de recherche, séparez les éléments de la chaîne par des points-virgules.

Les valeurs recherchées étant des Variants, vous devez soit transmettre un type tableau de Variants comme argument (par exemple, les valeurs renvoyées par la méthode *Lookup*), soit construire le tableau de Variants à la volée en utilisant la fonction *VarArrayOf*. Le code suivant illustre une recherche sur plusieurs colonnes faisant intervenir de multiples valeurs de recherche et utilisant les correspondances de clés partielles :

```
with CustTable do
  Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver','P']), loPartialKey);
```

Locate utilise la méthode de recherche la plus rapide pour trouver les correspondances d'enregistrements. Si les colonnes de la recherche sont indexées et que l'index est compatible avec les options de recherche spécifiées, *Locate* utilise cet index.

Utilisation de la méthode Lookup

Lookup recherche la première ligne qui correspond au critère de recherche. Si elle trouve une ligne correspondante, elle force le recalcul de tous les champs calculés et de tous les champs de référence associés à l'ensemble de données, puis elle renvoie un ou plusieurs champs de la ligne correspondante. *Lookup* ne déplace pas le curseur sur cette ligne ; elle ne fait que renvoyer certaines de ses valeurs.

Dans sa forme la plus simple, vous transmettez à *Lookup* le nom du champ de recherche, une valeur de champ pour établir la correspondance et le champ ou les champs à renvoyer. Par exemple, le code suivant recherche le premier enregistrement de *CustTable* pour laquelle la valeur dans le champ *Company* est

Professional Divers, Ltd., puis renvoie le nom de la société, le nom du contact commercial et son numéro de téléphone :

```
var
  LookupResults: Variant;
begin
  LookupResults := CustTable.Lookup('Company', 'Professional Divers, Ltd.',
     'Company;Contact; Phone');
end;
```

Lookup renvoie les valeurs des champs spécifiés du premier enregistrement qu'elle trouve. Les valeurs sont renvoyées en tant que Variants. Si plusieurs valeurs ont été demandées, *Lookup* renvoie un tableau de Variants. S'il n'existe aucun enregistrement correspondant, *Lookup* renvoie un Variant Null. Pour plus d'informations concernant les tableaux de Variants, voir l'aide en ligne.

Le vrai potentiel de *Lookup* se manifeste quand vous effectuez une recherche sur plusieurs colonnes et que vous spécifiez plusieurs valeurs à rechercher. Pour spécifier des chaînes contenant plusieurs colonnes ou des champs de résultats, vous devez séparer les éléments de la chaîne par des points-virgules.

Les valeurs recherchées étant des Variants, pour transmettre plusieurs valeurs, vous devez soit transmettre un type tableau de Variants comme argument (par exemple, les valeurs renvoyées par la méthode *Lookup*), soit construire le tableau de Variants à la volée en utilisant la fonction *VarArrayOf*. Le code suivant illustre une recherche sur plusieurs colonnes :

```
var
  LookupResults: Variant;
begin
with CustTable do
  LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver', 'Christiansted']),
    'Company; Addr1; Addr2; State; Zip');
end;
```

Comme *Locate*, *Lookup* utilise la méthode de recherche la plus rapide pour trouver les correspondances d'enregistrements. Si les colonnes de la recherche sont indexées, *Lookup* utilise l'index.

Affichage et édition d'ensembles de données en utilisant des filtres

Il arrive souvent qu'une application ne s'intéresse qu'à un sous-ensemble d'enregistrements d'un ensemble de données. Par exemple, votre application peut souhaiter récupérer ou visualiser dans une base de données des clients les enregistrements des sociétés dont le siège se trouve en Californie, ou bien rechercher un enregistrement contenant un ensemble de valeurs de champ déterminé. Dans tous les cas, vous pouvez utiliser des filtres pour limiter l'accès d'une application à un sous-ensemble de tous les enregistrements de l'ensemble de données.

Avec des ensembles de données unidirectionnels, vous ne pouvez limiter les enregistrements de l'ensemble de données que par le biais d'une requête. Avec d'autres descendants de *TDataSet*, vous pouvez toutefois définir un sous-ensemble dans les données déjà extraites. Pour limiter l'accès d'une application à un sous-ensemble de tous les enregistrements de l'ensemble de données, vous pouvez utiliser des filtres.

Le filtre spécifie des conditions qu'un enregistrement doit satisfaire pour être affiché. Les conditions de filtre peuvent être stipulées dans la propriété *Filter* de l'ensemble de données ou codées dans son gestionnaire d'événement *OnFilterRecord*. Les conditions de filtre sont basées sur les valeurs contenues dans un nombre quelconque de champs d'un ensemble de données, que ces champs soient indexés ou non. Ainsi, pour ne visualiser que les enregistrements correspondant à des entreprises situées en Californie, un filtre simple consistera à rechercher les enregistrements contenant la valeur "CA" dans le champ Etat.

Remarque Les filtres sont appliqués à chaque enregistrement récupéré dans l'ensemble de données. Pour extraire des volumes importants de données, il est plus efficace d'utiliser une requête pour restreindre la récupération des enregistrements ou de définir une portée sur un ensemble de données client indexée plutôt que d'utiliser des filtres.

Activation et désactivation des filtres

L'activation d'un filtre sur un ensemble de données est un processus qui se déroule en trois étapes :

1. Créer un filtre.
2. Définir des options de filtre pour les tests de filtres basés sur des chaînes, si nécessaire.
3. Définir la propriété *Filtered* par *True*.

Lorsque le filtrage est activé, seuls les enregistrements correspondant au critère de filtre sont disponibles pour une application. Le filtrage est toujours une condition temporaire. Vous pouvez désactiver le filtrage en définissant la propriété *Filtered* par *False*.

Création de filtres

Les deux méthodes suivantes permettent de créer un filtre pour un ensemble de données :

- Spécifiez des conditions de filtre dans la propriété *Filter*. *Filter* est particulièrement utile pour créer et appliquer des filtres à l'exécution.
- Ecrivez un gestionnaire d'événement *OnFilterRecord* pour les conditions de filtre simples ou complexes. Avec *OnFilterRecord*, les conditions de filtre sont spécifiées en mode conception. A la différence de la propriété *Filter*, qui se limite à une seule chaîne contenant une logique de filtre, l'événement

OnFilterRecord peut utiliser la logique des branchements et des boucles pour créer des conditions de filtre multiniveaux et complexes.

Lorsque vous créez des filtres en utilisant la propriété *Filter*, votre application peut créer, modifier et appliquer des filtres dynamiquement (en réponse à des saisies utilisateur, par exemple). L'inconvénient est que les conditions de filtre doivent pouvoir s'exprimer dans une seule chaîne texte, ne peuvent pas utiliser des constructions à base de branchements ou de boucles, et que leurs valeurs ne peuvent être ni testées ni comparées à des valeurs ne figurant pas encore dans l'ensemble de données.

La puissance de l'événement *OnFilterRecord* réside dans la possibilité pour un filtre d'être complexe et variable, d'être basé sur plusieurs lignes de code utilisant des constructions de branchements et de boucles, et de pouvoir tester et comparer les valeurs de l'ensemble de données avec des valeurs figurant à l'extérieur de l'ensemble de données, comme le texte d'une zone de saisie. Son inconvénient est que le filtre doit être défini en mode conception et qu'il ne peut pas être modifié en réponse à des saisies utilisateur. Vous pouvez toutefois créer plusieurs gestionnaires de filtres et permuter de l'un à l'autre en réponse à certaines conditions d'application.

Les sections suivantes décrivent comment créer des filtres en utilisant la propriété *Filter* et le gestionnaire d'événement *OnFilterRecord*.

Définition de la propriété Filter

Pour créer un filtre en utilisant la propriété *Filter*, vous devez spécifier une chaîne contenant la condition de filtre comme valeur de cette propriété. Par exemple, l'instruction suivante crée un filtre qui a pour effet de tester le champ *State* d'un ensemble de données pour voir s'il contient une valeur correspondant à l'état de Californie :

```
Dataset1.Filter := 'State = ' + QuotedStr('CA');
```

La valeur de la propriété *Filter* peut aussi être définie à partir du texte saisi dans un contrôle. Par exemple, l'instruction suivante affecte le texte d'une zone de saisie à la propriété *Filter* :

```
Dataset1.Filter := Edit1.Text;
```

Vous pouvez aussi créer une chaîne à partir de texte codé en dur et de données saisies par l'utilisateur dans un contrôle :

```
Dataset1.Filter := 'State = ' + QuotedStr(Edit1.Text);
```

Les enregistrements vierges ne s'affichent que s'ils sont explicitement inclus dans le filtre :

```
Dataset1.Filter := 'State <> ''CA'' or State = BLANK';
```

Remarque Après avoir spécifié une valeur pour la propriété *Filter*, pour appliquer le filtre à l'ensemble de données, définissez la propriété *Filtered* par *True*.

Les opérateurs logiques et de comparaison suivants permettent de comparer les valeurs des champs à des littéraux et à des constantes :

Tableau 18.4 Opérateurs logiques et de comparaison pouvant apparaître dans un filtre

Opérateur	Signification
<	Inférieur
>	Supérieur
>=	Supérieur ou égal
<=	Inférieur ou égal
=	Egal
<>	Différent
AND	Teste si deux instructions valent toutes deux *True*
NOT	Vérifie que l'instruction suivante ne vaut pas *True*
OR	Vérifie qu'au moins une des deux instructions vaut *True*
+	Ajoute des nombres, concatène des chaînes, ajoute des nombres à des valeurs date/heure (disponible seulement pour certains pilotes)
-	Soustrait un nombre d'un autre, une date d'une autre ou un nombre d'une date (disponible seulement pour certains pilotes)
*	Multiplie deux nombres (disponible seulement pour certains pilotes)
/	Divise deux nombres (disponible seulement pour certains pilotes)
*	caractère générique pour des comparaisons partielles (*FilterOptions* doit inclure *foPartialCompare*)

En utilisant des combinaisons de ces opérateurs, il est possible de créer des filtres sophistiqués. Par exemple, l'instruction suivante vérifie que deux conditions de test sont remplies avant d'afficher un enregistrement :

```
(Custno > 1400) AND (Custno < 1500);
```

Remarque Lorsque le filtrage est activé, les modifications apportées par l'utilisateur peuvent entraîner que l'enregistrement ne réponde plus aux conditions du filtre. La prochaine fois que l'enregistrement sera extrait de l'ensemble de données, il n'apparaîtra plus. Dans ce cas, le prochain enregistrement répondant à la condition de filtre devient l'enregistrement en cours.

Ecriture d'un gestionnaire d'événement OnFilterRecord

Vous pouvez écrire du code pour filtrer les enregistrements en utilisant les événements *OnFilterRecord* générés par l'ensemble à chaque récupération d'enregistrement. Ce gestionnaire d'événement implémente un test qui détermine si l'enregistrement est inclus dans ceux qui sont visibles dans l'application.

Pour indiquer qu'un enregistrement répond à une condition de filtre, votre gestionnaire *OnFilterRecord* définit son paramètre *Accept* par *True*, pour inclure un enregistrement, ou par *False*, pour l'exclure.

Par exemple, le filtre suivant affiche uniquement les enregistrements pour lesquels le champ State vaut CA :

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
  Accept := DataSet['State'].AsString = 'CA';
end;
```

Quand le filtrage est activé, l'ensemble de données génère un événement *OnFilterRecord* pour chaque enregistrement récupéré. Le gestionnaire d'événement teste chaque enregistrement et seuls ceux répondant aux conditions de filtre apparaissent dans l'application. Les performances étant directement dérivées du nombre de fois que l'événement se déclenche et de la durée du traitement de chaque événement, il est conseillé de réduire autant que possible le code du gestionnaire d'événement *OnFilterRecord*.

Permutation entre les gestionnaires d'événements filtre à l'exécution

Vous pouvez coder un nombre quelconque de gestionnaires d'événements *OnFilterRecord* et passer de l'un à l'autre à l'exécution. Par exemple, les instructions suivantes permettent de passer à un gestionnaire d'événement *OnFilterRecord* appelé *NewYorkFilter* :

```
DataSet1.OnFilterRecord := NewYorkFilter;
Refresh;
```

Définition d'options de filtre

La propriété *FilterOptions* permet de spécifier si un filtre comparant des champs basés sur des chaînes accepte des enregistrements à partir de comparaisons partielles et si les comparaisons chaîne tiennent compte de la distinction majuscules/minuscules. *FilterOptions* est une propriété ensemble qui peut être un ensemble vide (la valeur par défaut) ou contenir l'une des valeurs suivantes (ou les deux) :

Tableau 18.5 Valeurs de FilterOptions

Valeur	Signification
foCaseInsensitive	Ignore les différences majuscules/minuscules lors de la comparaison des chaînes.
foNoPartialCompare	Désactive les correspondances de chaînes partielles, il s'en suit qu'aucune correspondance ne peut être établie avec les chaînes se terminant par un astérisque (*).

Par exemple, les instructions suivantes définissent un filtre qui ignore les différences majuscules/minuscules lors de la comparaison des valeurs d'un champ *State* :

```
FilterOptions := [foCaseInsensitive];
Filter := 'State = ' + QuotedStr('CA');
```

Navigation parmi les enregistrements d'un ensemble de données filtré

Quatre méthodes permettent de naviguer parmi les enregistrements d'un ensemble de données filtré. Le tableau suivant dresse la liste de ces méthodes et décrit leur utilisation :

Tableau 18.6 Méthodes navigationnelles relatives aux ensembles de données filtrés

Méthodes	Utilisation
FindFirst	Se positionne sur le premier enregistrement correspondant au critère de filtre en cours. Cette recherche commence toujours par le premier enregistrement de l'ensemble de données non filtré.
FindLast	Se positionne sur le dernier enregistrement correspondant au critère de filtre en cours.
FindNext	Se déplace de l'enregistrement en cours dans l'ensemble de données filtré sur le prochain enregistrement.
FindPrior	Se déplace de l'enregistrement en cours dans l'ensemble de données filtré sur l'enregistrement précédent.

Par exemple, l'instruction suivante trouve le premier enregistrement filtré dans un ensemble de données :

```
DataSet1.FindFirst;
```

Dans la mesure où la propriété *Filter* a été définie ou que vous avez créé un gestionnaire d'événement *OnFilterRecord* pour votre application, ces méthodes positionnent le curseur sur l'enregistrement spécifié, que le filtre soit ou non activé. Si ces méthodes sont appelées lorsque le filtrage n'est pas activé, elles provoquent les effets suivants :

- Le filtrage est temporairement activé.
- En cas de correspondance, le curseur est positionné sur un enregistrement.
- Le filtrage est désactivé.

Remarque Si le filtrage est désactivé, si la propriété *Filter* n'a pas été définie et si aucun gestionnaire d'événement *OnFilterRecord* n'a été créé, ces méthodes ont le même effet que *First*, *Last*, *Next* et *Prior*.

Toutes les méthodes navigationnelles relatives aux filtres positionnent le curseur sur un enregistrement (si un enregistrement correspond au filtre spécifié), le traitent comme l'enregistrement en cours et *True*. Si aucun enregistrement ne correspond au filtre, la position du curseur reste inchangée, et les méthodes renvoient *False*. Vous pouvez vérifier l'état de la propriété *Found* pour envelopper ces appels et n'agir que lorsque *Found* vaut *True*. Par exemple, si le curseur est déjà positionné sur le dernier enregistrement correspondant de l'ensemble de données lorsque vous appelez *FindNext*, la méthode renvoie *False*, et l'enregistrement en cours reste le même.

Modification des données

Vous pouvez utiliser les méthodes d'ensembles de données suivantes pour insérer, mettre à jour et supprimer des données si la propriété en lecture seule *CanModify* vaut *True*. *CanModify* vaut *True* sauf si l'ensemble de données est unidirectionnel, la base de données sous-jacente à l'ensemble de données ne reconnaît pas les droits d'accès en lecture et en écriture ou un autre facteur intervient. (Les facteurs pouvant intervenir sont la propriété*ReadOnly* de certains ensembles de données ou la propriété *RequestLive* des composants *TQuery*.)

Tableau 18.7 Méthodes des ensembles de données pour éditer des données

Méthode	Description
Edit	Met l'ensemble de données à l'état *dsEdit* s'il n'est pas déjà à l'état *dsEdit* ou *dsInsert*.
Append	Emet les données en suspens, déplace le curseur à la fin de l'ensemble de données, puis met ce dernier à l'état *dsInsert*.
Insert	Emet les données en suspens, puis met l'ensemble de données à l'état *dsInsert*.
Post	Tente d'émettre l'enregistrement nouveau ou modifié vers la base de données. Si l'opération réussit, l'ensemble de données est mis à l'état *dsBrowse* ; dans le cas contraire, son état en cours reste inchangé.
Cancel	Annule l'opération en cours et met l'ensemble de données à l'état *dsBrowse*.
Delete	Supprime l'enregistrement en cours et met l'ensemble de données à l'état *dsBrowse*.

Modification d'enregistrements

Un ensemble de données doit être en mode *dsEdit* pour que l'application puisse modifier des enregistrements. Dans votre code, vous pouvez utiliser la méthode *Edit* pour basculer un ensemble de données en mode *dsEdit* si la propriété en lecture seule *CanModify* de l'ensemble de données vaut *True*.

Quand un ensemble de données passe en mode *dsEdit*, il reçoit d'abord un événement *BeforeEdit*. Une fois le passage en mode édition réussi, l'ensemble de données reçoit un événement *AfterEdit*. En général, ces événements sont utilisés pour mettre à jour l'interface utilisateur afin qu'elle indique l'état en cours de l'ensemble de données. Si l'ensemble de données ne peut passer en mode édition pour une raison ou pour une autre, un événement *OnEditError* survient, grâce auquel vous pouvez informer l'utilisateur du problème ou essayer de corriger la situation qui a empêché l'ensemble de données de passer en mode édition.

Dans les fiches de votre application, certains contrôles orientés données pourront mettre automatiquement votre ensemble de données à l'état *dsEdit* si

- La propriété *ReadOnly* du contrôle vaut *False* (la valeur par défaut).
- La propriété *AutoEdit* de la source de données du contrôle vaut *True* et
- La propriété *CanModify* de l'ensemble de données vaut *True*.

Remarque Même si un ensemble de données est à l'état *dsEdit*, la modification d'enregistrements peut échouer pour les bases de données SQL si l'utilisateur de votre application ne dispose pas des droits d'accès SQL appropriés.

Quand un ensemble de données se trouve en mode *dsEdit*, l'utilisateur peut modifier toutes les valeurs de champ de l'enregistrement qui apparaît dans les contrôles orientés données d'une fiche. Les contrôles orientés données, pour lesquels la modification est activée automatiquement, appellent *Post* quand l'utilisateur accomplit une action qui change la position du curseur (comme le déplacement vers un autre enregistrement dans une grille).

Si vous avez mis un composant navigateur dans votre fiche, l'utilisateur peut annuler les modifications en cliquant sur le bouton Annuler du navigateur. L'annulation des modifications renvoie l'ensemble de données à l'état *dsBrowse*.

Dans votre code, vous devez valider ou annuler les modifications en appelant les méthodes appropriées. Les modifications sont validées par l'appel de *Post*. Vous les annulez en appelant *Cancel*. Les méthodes *Edit* et *Post* sont souvent utilisées conjointement. Par exemple,

```
with CustTable do
begin
  Edit;
  FieldValues['CustNo'] := 1234;
  Post;
end;
```

Dans l'exemple précédent, la première ligne du fragment de code place l'ensemble de données en mode *dsEdit*. La ligne suivante affecte la chaîne 1234 au champ *CustNo* de l'enregistrement en cours. Pour finir, la dernière ligne écrit (émet) l'enregistrement modifié. Si les mises à jour ne sont pas en mémoire cache, les modifications sont écrites en retour dans la base de données. Si les mises à jour sont en mémoire cache, les modifications sont écrites dans un tampon temporaire, où elles restent jusqu'à ce que la méthode *ApplyUpdates* de l'ensemble de données soit appelée.

Ajout de nouveaux enregistrements

Un ensemble de données doit être en mode *dsInsert* pour que l'application puisse ajouter de nouveaux enregistrements. Dans votre code, vous pouvez utiliser les méthodes *Insert* ou *Append* pour mettre un ensemble de données en mode *dsInsert* si la propriété *CanModify* de l'ensemble de données, accessible seulement en lecture, vaut *True*.

Quand un ensemble de données passe en mode *dsInsert*, il reçoit d'abord un événement *BeforeInsert*. Une fois le passage en mode insertion réussi, l'ensemble de données reçoit d'abord un événement *OnNewRecord*, puis un événement *AfterInsert*. Vous pouvez utiliser ces événements, par exemple, pour fournir les valeurs initiales des enregistrements que vous venez d'insérer.

```
procedure TForm1.OrdersTableNewRecord(DataSet: TDataSet);
begin
  DataSet.FieldByName('OrderDate').AsDateTime := Date;
end;
```

Dans les fiches de votre application, les contrôles orientés données grille et navigateur pourront mettre automatiquement votre ensemble de données à l'état *dsInsert* si les conditions suivantes sont réunies :

- La propriété *ReadOnly* du contrôle vaut *False* (la valeur par défaut).
- La propriété *CanModify* de l'ensemble de données vaut *True*.

Remarque Même si un ensemble de données est à l'état *dsInsert*, l'ajout d'enregistrements peut échouer pour les bases de données SQL si l'utilisateur de votre application ne dispose pas des droits d'accès SQL appropriés.

Quand un ensemble de données se trouve en mode *dsInsert*, l'utilisateur ou bien l'application peut entrer des valeurs dans les champs associés au nouvel enregistrement. Sauf pour les contrôles grille et navigateur, il n'y a aucune différence apparente entre *Insert* et *Append*. Lors d'un appel à *Insert*, une ligne vide apparaît dans la grille au-dessus de l'enregistrement en cours. Lors d'un appel à *Append*, la grille défile jusqu'au dernier enregistrement de l'ensemble de données, une ligne vide apparaît à la fin de la grille, et les boutons *Suivant* et *Précédent* sont grisés sur tout composant navigateur associé à l'ensemble de données.

Les contrôles orientés données, pour lesquels l'insertion est activée automatiquement, appellent *Post* quand l'utilisateur accomplit une action qui change la position du curseur (comme le déplacement vers un autre enregistrement dans une grille). Vous devez sinon appeler *Post* dans votre code.

Post écrit le nouvel enregistrement dans la base de données ou, si vous utilisez les mises à jour en mémoire cache, *Post* écrit l'enregistrement dans un cache en mémoire. Pour écrire les insertions mises en mémoire cache et les ajouter à la base de données, appelez la méthode *ApplyUpdates* de l'ensemble de données.

Insertion d'enregistrements

Insert ouvre un nouvel enregistrement vide avant l'enregistrement en cours, puis positionne le curseur sur ce nouvel enregistrement de façon à que les valeurs de champ puissent être entrées par l'utilisateur ou par le code de votre application.

Quand une application *Post* (ou *ApplyUpdates* si vous utilisez les mises à jour en mémoire cache), le nouvel enregistrement inséré peut être écrit comme suit dans le journal des modifications :

- Pour les tables Paradox et dBASE indexées, l'enregistrement est inséré dans l'ensemble de données à une position déterminée par l'index.
- Pour les tables Paradox et dBASE non indexées, l'enregistrement est inséré dans l'ensemble de données à sa position actuelle.
- Pour les bases de données SQL, l'emplacement physique de l'insertion est spécifique à l'implémentation. Si la table est indexée, l'index est mis à jour avec les données du nouvel enregistrement.

Ajout d'enregistrements à la fin

Append ouvre un nouvel enregistrement vide à la fin de l'ensemble de données, en faisant en sorte que l'enregistrement vide devienne l'enregistrement en cours afin que les valeurs de champ de l'enregistrement puissent être entrées par l'utilisateur ou par le code de votre application.

Quand une application *Post* (ou *ApplyUpdates* si vous utilisez les mises à jour en mémoire cache), le nouvel enregistrement ajouté peut être écrit comme suit dans le journal des modifications :

- Pour les tables Paradox et dBASE indexées, l'enregistrement est inséré dans l'ensemble de données à une position déterminée par l'index.
- Pour les tables Paradox et dBASE non indexées, l'enregistrement est ajouté à la fin de l'ensemble de données.
- Pour les bases de données SQL, l'emplacement physique de l'ajout est spécifique à l'implémentation. Si la table est indexée, l'index est mis à jour avec les données du nouvel enregistrement.

Suppression d'enregistrements

Utilisez la méthode *Delete* pour supprimer l'enregistrement en cours d'un ensemble de données actif. Quand la méthode *Delete* est appelée,

1. L'ensemble de données reçoit un événement *BeforeDelete*.
2. L'ensemble de données tente de supprimer l'enregistrement en cours.
3. L'ensemble de données revient à l'état *dsBrowse*.
4. L'ensemble de données reçoit un événement *AfterDelete*.

Si vous voulez empêcher la suppression dans le gestionnaire d'événement *BeforeDelete*, vous pouvez appeler la procédure globale *Abort* :

```
procedure TForm1.TableBeforeDelete (Dataset: TDataset)begin
  if MessageDlg('Delete This Record?', mtConfirmation, mbYesNoCancel, 0) <> mrYes then
    Abort;
end;
```

Si *Delete* échoue, elle génère un événement *OnDeleteError*. Si le gestionnaire d'événement *OnDeleteError* ne peut pas corriger le problème, l'ensemble de données reste dans l'état *dsEdit*. Si *Delete* réussit, l'ensemble de données revient à l'état *dsBrowse* et l'enregistrement suivant celui qui est supprimé devient l'enregistrement en cours.

Si les mises à jour sont en mémoire cache, l'enregistrement supprimé n'est enlevé de la table de la base de données sous-jacente que si vous appelez *ApplyUpdates*.

Si vous avez doté vos fiches d'un composant navigateur, les utilisateurs pourront supprimer l'enregistrement en cours en cliquant sur le bouton d'effacement. Dans votre code, vous devez appeler explicitement *Delete* pour supprimer l'enregistrement en cours.

Validation des données

Quand vous avez fini de modifier un enregistrement, vous devez appeler la méthode *Post* pour écrire effectivement les modifications. La méthode *Post* se comporte de façon différente selon l'état dans lequel se trouve l'ensemble de données et selon que vous mettiez ou non les mises à jour en mémoire cache.

- Si vous ne mettez pas les mises à jour en mémoire cache, et que l'ensemble de données est dans l'état *dsEdit* ou *dsInsert*, *Post* écrit l'enregistrement en cours dans la base de données et remet l'ensemble de données dans l'état *dsBrowse*.
- Si vous mettez les mises à jour en mémoire cache, et que l'ensemble de données est dans l'état *dsEdit* ou *dsInsert*, *Post* écrit l'enregistrement en cours dans un cache interne et remet l'ensemble de données dans l'état *dsBrowse*. Les modifications ne sont écrites dans la base de données que lorsque vous appelez *ApplyUpdates*.
- Si l'ensemble de données est dans l'état *dsSetKey*, *Post* le met dans l'état *dsBrowse*.

Quel que soit l'état initial de l'ensemble de données, *Post* génère des événements *BeforePost* et *AfterPost*, avant et après l'écriture des modifications en cours. Vous pouvez utiliser ces événements pour mettre à jour l'interface utilisateur, ou empêcher l'ensemble de données de valider les modifications en appelant la procédure *Abort*. Si l'appel à *Post* échoue, l'ensemble de données reçoit un événement *OnPostError*, dans lequel vous pouvez informer l'utilisateur du problème ou essayer de le corriger.

L'émission peut se faire, soit explicitement soit implicitement, comme partie intégrante d'une autre procédure. Quand une application quitte l'enregistrement en cours, *Post* est appelée implicitement. Les appels aux méthodes *First*, *Next*, *Prior* et *Last* effectuent une validation (*Post*) si la table est en mode *dsEdit* ou *dsInsert*. En outre, les méthodes *Append* et *Insert* valident implicitement toute donnée en suspens.

Attention La méthode*Close* n'appelle pas implicitement *Post*. Utilisez l'événement *BeforeClose* pour valider explicitement toute modification en suspens.

Annulation des modifications

Une application peut à tout moment annuler les changements apportés à l'enregistrement en cours, si elle n'a pas, directement ou indirectement, déjà appelé *Post*. Par exemple, si un ensemble de données est en mode *dsEdit* et qu'un utilisateur a changé les données d'un ou de plusieurs champs, l'application peut rétablir les valeurs initiales de l'enregistrement en appelant la méthode *Cancel* de l'ensemble de données. Un appel à *Cancel* remet toujours l'ensemble de données dans l'état *dsBrowse*.

Si l'ensemble de données était en mode *dsEdit* ou *dsInsert* lorsque votre application a appelé *Cancel*, il reçoit les événements *BeforeCancel* et *AfterCancel* avant et après la restauration des valeurs initiales de l'enregistrement en cours.

Pour une fiche, vous pouvez permettre à l'utilisateur d'annuler les opérations de modification, d'insertion, et d'ajout en incluant le bouton Annuler dans un composant navigateur associé à l'ensemble de données, ou bien le code de votre propre bouton d'annulation.

Modification d'enregistrements entiers

Dans les fiches, tous les contrôles orientés données, à l'exception des grilles et des navigateurs, donnent accès à un champ unique d'enregistrement.

Par contre, dans votre code, vous pouvez utiliser les méthodes suivantes qui fonctionnent sur des structures d'enregistrements entiers à condition toutefois que la structure des tables de la base sous-jacente à l'ensemble de données soit stable et ne subisse aucun changement. Le tableau suivant récapitule les méthodes disponibles pour manipuler des enregistrements entiers plutôt que des champs individuels de ces enregistrements :

Tableau 18.8 Méthodes qui opèrent sur des enregistrements entiers

Méthode	Description
AppendRecord([tableau de valeurs])	Ajoute un enregistrement avec les valeurs de colonne spécifiées à la fin de la table ; analogue à *Append*. Exécute implicitement *Post*.
InsertRecord([tableau de valeurs])	Insère un enregistrement avec les valeurs de colonne spécifiées avant la position en cours du curseur dans la table ; analogue à *Insert*. Exécute implicitement *Post*.
SetFields([tableau de valeurs])	Définit les valeurs des champs correspondants ; analogue à l'affectation de valeurs aux composants *TField*. L'application doit exécuter explicitement *Post*.

Chacune de ces méthodes accepte comme argument un tableau de valeurs, où chaque valeur correspond à une colonne de l'ensemble de données sous-jacent. Les valeurs peuvent être littérales, variables ou NULL. Si le nombre de valeurs de l'argument est inférieur au nombre de colonnes dans l'ensemble de données, les valeurs restantes sont supposées avoir la valeur NULL.

Pour les ensembles de données non indexés, *AppendRecord* ajoute un enregistrement à la fin de l'ensemble de données et *InsertRecord* insère un enregistrement après la position en cours du curseur. Pour les ensembles de données indexés, les deux méthodes placent l'enregistrement à la bonne position dans la table, déterminée par l'index. Dans les deux cas, les deux méthodes déplacent le curseur sur le nouvel enregistrement.

SetFields affecte les valeurs spécifiées dans le tableau de paramètres aux champs de l'ensemble de données. Pour utiliser *SetFields*, l'application doit d'abord appeler *Edit* pour mettre l'ensemble de données en mode *dsEdit*. Pour appliquer les modifications dans l'enregistrement en cours, elle doit exécuter *Post*.

Si vous utilisez *SetFields* pour modifier certains champs, et non tous les champs d'un enregistrement existant, vous pouvez transmettre des valeurs NULL pour les champs que vous ne voulez pas changer. Si vous ne fournissez pas un

nombre de valeurs correspondant au nombre de champs d'un enregistrement, *SetFields* leur affecte la valeur NULL. Les valeurs NULL écrasent les valeurs existantes de ces champs.

Par exemple, supposons qu'une base de données dispose d'une table COUNTRY avec les colonnes Name, Capital, Continent, Area et Population. Si un composant *TTable* appelé *CountryTable* a été lié à la table COUNTRY, l'instruction suivante insérera un enregistrement dans la table COUNTRY :

```
CountryTable.InsertRecord(['Japan', 'Tokyo', 'Asia']);
```

Cette instruction ne spécifie aucune valeur pour Area et Population, des valeurs NULL leur sont donc affectées. La table est indexée sur Name, l'enregistrement est donc inséré à la position alphabétique de "Japan".

Pour mettre à jour l'enregistrement, l'application peut utiliser le code suivant :

```
with CountryTable do
begin
  if Locate('Name', 'Japan', loCaseInsensitive) then;
  begin
    Edit;
    SetFields(nil, nil, nil, 344567, 164700000);
    Post;
  end;
end;
```

Ce code affecte des valeurs aux champs Area et Population, avant de les enregistrer dans la base de données. Les trois pointeurs NULL agissent comme marqueurs de remplissage des trois premières colonnes pour indiquer que leur contenu actuel doit être préservé.

Champs calculés

En utilisant l'éditeur de champs, vous pouvez définir des champs calculés pour vos ensembles de données. Quand un ensemble de données contient des champs calculés, vous fournissez le code calculant les valeurs de ces champs dans un gestionnaire d'événement *OnCalcFields*. Pour des détails sur la définition des champs calculés en utilisant l'éditeur de champs, voir "Définition d'un champ calculé" à la page 19-8.

La propriété *AutoCalcFields* détermine quand *OnCalcFields* est appelé. Si *AutoCalcFields* vaut *True*, *OnCalcFields* est appelé quand :

- Un ensemble de données est ouvert.
- L'ensemble de données passe en mode édition.
- Un enregistrement est récupéré dans la base de données.
- La focalisation se déplace d'un composant visuel à un autre, ou bien d'une colonne à une autre dans un contrôle grille orienté données.

Si *AutoCalcFields* vaut *False*, alors *OnCalcFields* n'est pas appelé quand on modifie individuellement des champs se trouvant à l'intérieur d'un enregistrement (la quatrième condition ci-dessus).

Attention Comme *OnCalcFields* est fréquemment appelé, l'exécution du code que vous écrivez pour ce gestionnaire doit être brève. En outre, si *AutoCalcFields* vaut *True*, *OnCalcFields* ne doit accomplir aucune action qui modifie l'ensemble de données (ou l'ensemble de données lié s'il fait partie d'une relation maître-détail), car cela induit une récursion. Par exemple, si *OnCalcFields* exécute *Post*, et que *AutoCalcFields True*, alors *OnCalcFields* est à nouveau appelé, provoquant un nouvel appel à *Post*, etc.

Quand *OnCalcFields* s'exécute, l'ensemble de données passe en mode *dsCalcFields*. Cet état empêche les modifications et les ajouts dans les enregistrements sauf s'ils s'appliquent aux champs calculés modifiés par le gestionnaire lui-même. La raison qui empêche d'autres modifications est que *OnCalcFields* utilise les valeurs des autres champs pour dériver celles des champs calculés. Sinon, les changements des autres champs pourraient invalider les valeurs affectées aux champs calculés. L'exécution de *OnCalcFields* achevée, l'ensemble de données revient à l'état *dsBrowse*.

Types d'ensembles de données

"Utilisation des descendants de TDataSet" à la page 18-2 classe les descendants de *TDataSet* selon la méthode qu'ils utilisent pour accéder à leurs données. Un autre moyen pratique de classer les descendants de *TDataSet* est de considérer le type de données du serveur qu'ils représentent. Selon ce moyen, il y a trois classes d'ensembles de données de base :

- **Ensembles de données de type table** : Les ensembles de données de type table représentent une seule table du serveur de base de données, avec toutes ses lignes et toutes ses colonnes. Les ensembles de données de type table sont *TTable*, *TADOTable*, *TSQLTable* et *TIBTable*.

  Les ensembles de données de type table vous permettent de bénéficier des index définis sur le serveur. Comme il y a une correspondance un-à-un entre la table de la base de données et l'ensemble de données, vous pouvez utiliser les index du serveur qui ont été définis pour cette table. Les index permettent à votre application de trier les enregistrement de la table, accélèrent les recherches et les références, et peuvent servir de base à une relation maître/détail. Certains ensembles de données de type table tirent également avantage de la relation un-à-un l'entre ensemble de données et la table de la base de données pour vous permettre d'effectuer des opérations comme la création et la suppression de tables.

- **Ensembles de données de type requête** : Les ensembles de données de type requête représentent une seule commande, ou requête, SQL. Les requêtes peuvent représenter l'ensemble de résultats d'une commande d'exécution (généralement une instruction SELECT) ou peuvent exécuter une commande qui ne renvoie aucun enregistrement (par exemple une instruction UPDATE).

Les ensembles de données de type requête sont *TQuery*, *TADOQuery*, *TSQLQuery* et *TIBQuery*.

Pour utiliser efficacement un ensemble de données de type requête, vous devez être familier de SQL et de l'implémentation SQL de votre serveur, y compris des limitations et des extensions du standard SQL-92. Si vous débutez dans SQL, vous pouvez vous procurer un manuel externe qui traite de SQL en détail. Un des meilleurs est *Understanding the New SQL: A Complete Guide*, de Jim Melton et Alan R. Simpson, Morgan Kaufmann Publishers.

- **Ensembles de données de type procédure** : Les ensembles de données de type procédure représentent une procédure stockée sur le serveur de base de données. Les ensembles de données de type procédure sont *TStoredProc*, *TADOStoredProc*, *TSQLStoredProc* et *TIBStoredProc*.

  Une procédure stockée est un programme autonome écrit dans le langage de procédure et de déclenchement propre au système de base de données utilisé. Elles gèrent, en général, les tâches très répétitives sur les bases de données et sont particulièrement utiles pour les opérations qui s'appliquent à un grand nombre d'enregistrements ou qui utilisent des fonctions statistiques et mathématiques. Les procédures stockées améliorent généralement les performances des applications de bases de données, car :

  - Elles tirent parti de la plus grande puissance et de la plus grande rapidité fournies par le serveur.
  - Elles réduisent le trafic sur le réseau en déplaçant le traitement sur le serveur.

  Les procédures stockées peuvent renvoyer des données ou non. Celles qui le font doivent renvoyer les données sous forme d'un curseur (comme le résultat d'une requête SELECT), sous forme de plusieurs curseurs (en renvoyant réellement plusieurs ensembles de données), ou dans des paramètres de sortie. Ces différences proviennent en partie du serveur : certains d'entre eux n'autorisent pas les procédures stockées à renvoyer des données ou n'autorisent que les paramètres de sortie. Certains autres ne supportent pas les procédures stockées. Reportez-vous à la documentation de votre serveur pour savoir ce qu'il propose.

Remarque Vous pouvez, en général, utiliser un ensemble de données de type requête pour exécuter des procédures stockées car la plupart des serveurs fournissent des extensions SQL permettant de travailler avec les procédures stockées. Mais, chaque serveur utilise pour cela sa propre syntaxe. Si vous choisissez d'utiliser un ensemble de données de type requête à la place d'un ensemble de données de type procédure stockée, cherchez la syntaxe nécessaire dans la documentation de votre serveur.

Outre les ensembles de données faisant clairement partie d'une de ces trois catégories, quelques descendants de *TDataSet* appartiennent à plusieurs catégories :

- *TADODataSet* et *TSQLDataSet*possèdent une propriété *CommandType* qui vous permet de spécifier s'ils représentent une table, une requête ou une procédure stockée. Les propriétés et les méthodes sont presque les mêmes que celles des

ensembles de données de type requête, bien que *TADODataSet* vous permette de spécifier un index comme dans les ensembles de données de type table.

- *TClientDataSet* représente les données d'un autre ensemble de données. De ce fait, il peut représenter une table, une requête et une procédure stockée. *TClientDataSet* se comporte pratiquement comme un ensemble de données de type table puisqu'il supporte les index. Mais, il possède aussi quelques fonctionnalités des requêtes et des procédures stockées : la gestion des paramètres et la possibilité de s'exécuter sans renvoyer d'ensemble de résultats.
- D'autres ensembles de données client (*TBDEClientDataSet* et *TSQLClientDataSet*) possèdent une propriété *CommandType* qui vous permet de spécifier s'ils représentent une table, une requête ou une procédure stockée. Les propriétés et méthodes sont comme *TClientDataSet*, y compris le support des paramètres, des index, et la possibilité de s'exécuter sans renvoyer d'ensemble de résultats.
- *TIBDataSet* peut représenter à la fois des requêtes et des procédures stockées. En fait, il peut représenter plusieurs requêtes et procédures stockées simultanément, sans qu'il y ait des propriétés différentes pour chacune.

Utilisation d'ensembles de données de type table

Pour utiliser un ensemble de données de type table,

1. Placez le composant approprié dans un module de données ou sur une fiche, et attribuez à sa propriété *Name* une valeur unique appropriée à votre application.
2. Identifiez le serveur de base de données qui contient les données à utiliser. Chaque ensemble de données de type table fait cela différemment, mais, en général, il faut spécifier un composant de base de données :
   - Pour *TTable*, spécifiez un composant *TDatabase* ou un alias BDE en utilisant la propriété *DatabaseName*.
   - Pour *TADOTable*, spécifiez un composant *TADOConnection* en utilisant la propriété *Connection*.
   - Pour *TSQLTable*, spécifiez un composant *TSQLConnection* en utilisant la propriété *SQLConnection*.
   - Pour *TIBTable*, spécifiez un composant *TIBConnection* en utilisant la propriété *Database*.

   Pour plus d'informations sur l'utilisation des composants connexion aux bases de données, voir chapitre 17, "Connexion aux bases de données".
3. Définissez la propriété *TableName* par le nom de la table de la base de données. Vous pouvez sélectionner des tables dans une liste déroulante si vous avez déjà identifié un composant connexion aux bases de données.

4 Placez un composant source de données dans le module de données ou sur la fiche, et définissez sa propriété *DataSet* par le nom de l'ensemble de données. Le composant source de données est utilisé pour transmettre un ensemble de résultats à afficher entre l'ensemble de données et les composants orientés données.

Avantages de l'utilisation des ensembles de données de type table

Le principale avantage de l'utilisation des ensembles de données de type table est la disponibilité des index. Les index permettent à votre application de

- Trier les enregistrements de l'ensemble de données.
- Localiser rapidement les enregistrements.
- Limiter les enregistrements visibles.
- Etablir des relations maître/détail.

De plus, la relation un-à-un entre les ensembles de données de type table et les tables de bases de données permet à nombre d'entre eux d'être utilisés pour les opérations suivantes :

- Contrôle des accès en lecture/écriture aux tables
- Création et suppression des tables
- Vidage des tables
- Synchronisation des tables

Tri des enregistrements avec des index

Un index détermine l'ordre d'affichage des enregistrements d'une table. En général, les enregistrements apparaissent dans l'ordre ascendant selon un index primaire, ou par défaut. Ce comportement par défaut ne requiert pas l'intervention de l'application. Si vous voulez un ordre de tri différent, vous devez spécifier au choix

- Un autre index.
- La liste des colonnes sur lesquelles trier (non disponible sur les serveurs non basés sur SQL).

Les index vous permettent de présenter les données d'une table selon des ordres différents. Pour les tables SQL, cet ordre de tri est implémenté en utilisant l'index pour générer une clause ORDER BY dans une requête lisant les enregistrements d'une table. Pour les autres tables (comme Paradox et dBASE), l'index est utilisé par le mécanisme d'accès aux données pour présenter les données dans l'ordre voulu.

Obtention d'informations sur les index

Votre application peut obtenir des informations sur les index définis par le serveur à partir de tout ensemble de données de type table. Pour obtenir la liste des index disponibles pour l'ensemble de données, appelez la méthode *GetIndexNames*. *GetIndexNames* remplit une liste de chaînes avec des noms

d'index valides. Par exemple, le code suivant remplit une boîte liste des noms de tous les index définis pour l'ensemble de données *CustomersTable* :

```
CustomersTable.GetIndexNames(ListBox1.Items);
```

Remarque Pour les tables Paradox, l'index primaire n'est pas nommé et n'est donc pas renvoyé par *GetIndexNames*. Vous pouvez quand même redéfinir en index primaire l'index d'une table Paradox, lorsque vous avez utilisé un autre index, en définissant la propriété *IndexName* par une chaîne vide.

Pour obtenir des informations sur les champs de l'index en cours, utilisez les propriétés

- *IndexFieldCount*, qui détermine le nombre de colonnes de l'index.
- *IndexFields*, qui examine la liste des composants champ des colonnes qui constituent l'index.

Le code suivant illustre l'utilisation des propriétés *IndexFieldCount* et *IndexFields* pour parcourir une liste de noms de colonnes dans une application :

```
var
  I: Integer;
  ListOfIndexFields: array[0 to 20} of string;
begin
with CustomersTable do
  begin
  for I := 0 to IndexFieldCount - 1 do
     ListOfIndexFields[I] := IndexFields[I].FieldName;
  end;
end;
```

Remarque *IndexFieldCount* n'est pas valide pour une table dBASE ouverte sur une expression d'index.

Spécification d'un index avec IndexName

Utilisez la propriété *IndexName* pour forcer un index à être actif. Une fois actif, un index détermine l'ordre des enregistrements d'un ensemble de données. (Il peut aussi être utilisé pour un lien maître-détail, une recherche indexée ou un filtrage indexé).

Pour activer un index, définissez la propriété *IndexName* par le nom de l'index. Dans certains systèmes de bases de données, les index primaires n'ont pas de nom. Pour activer un de ces index, définissez la propriété *IndexName* par une chaîne vide.

A la conception, vous pouvez sélectionner un index dans la liste des index disponibles, en cliquant sur le bouton points de suspension dans l'inspecteur d'objets. A l'exécution, définissez *IndexName* en utilisant un littéral ou une variable ***String***. Pour obtenir la liste des index disponibles, appelez la méthode *GetIndexNames*.

Le code suivant définit l'index de *CustomersTable* par *CustDescending* :

```
CustomersTable.IndexName := 'CustDescending';
```

Création d'un index avec IndexFieldNames

Si aucun index n'est défini pour implémenter l'ordre de tri que vous voulez, vous pouvez créer un pseudo-index en utilisant la propriété *IndexFieldNames*.

Remarque *IndexName* et *IndexFieldNames* s'excluent mutuellement. Définir la valeur de l'une efface la valeur de l'autre.

La valeur de *IndexFieldNames* est de type string. Pour spécifier un ordre de tri, indiquez chacun des noms de colonnes dans l'ordre où ils doivent être utilisés en les délimitant par des points-virgules. Le tri est uniquement croissant. La distinction majuscules/minuscules du tri dépend des capacités de votre serveur. Consultez la documentation de votre serveur pour davantage d'informations.

Le code suivant définit l'ordre de tri de *PhoneTable* selon *LastName*, puis *FirstName* :

```
PhoneTable.IndexFieldNames := 'LastName;FirstName';
```

Remarque Si vous utilisez *IndexFieldNames* sur des tables Paradox ou dBASE, l'ensemble de données tente de trouver un index utilisant les colonnes que vous avez spécifiées. S'il n'en trouve pas, il déclenche une exception.

Utilisation d'index pour chercher des enregistrements

Vous pouvez effectuer une recherche dans n'importe quel ensemble de données en utilisant les méthodes *Locate* et *Lookup* de *TDataSet*. Mais, l'utilisation explicite d'index, peut améliorer pour certains ensembles de données de type table les performances de recherche des méthodes *Locate* et *Lookup*.

Les ensembles de données ADO supportent tous la méthode *Seek*, qui permet d'aller sur un enregistrement grâce à un ensemble de valeurs des champs de l'index en cours. *Seek* vous permet de spécifier où placer le curseur par rapport au premier ou au dernier enregistrement correspondant.

TTable et tous les types d'ensembles de données client supportent des recherches indexées similaires, mais ils utilisent une combinaison de méthodes correspondantes. Le tableau suivant présente les six méthodes fournies par *TTable* et les ensembles de données client pour prendre en charge les recherches indexées :

Tableau 18.9 Méthodes de recherche indexée

Méthode	Utilisation
EditKey	Réserve le contenu actuel du tampon de clés de recherche et place l'ensemble de données en mode *dsSetKey* afin de permettre à votre application de modifier les critères de recherche existants avant l'exécution de la recherche.
FindKey	Combine les méthodes *SetKey* et *GotoKey* en une seule méthode.
FindNearest	Combine les méthodes *SetKey* et *GotoNearest* en une seule méthode.
GotoKey	Recherche le premier enregistrement d'un ensemble de données correspondant exactement au critère de recherche et place le curseur dessus s'il en trouve un.

Tableau 18.9 Méthodes de recherche indexée (suite)

Méthode	Utilisation
GotoNearest	Recherche dans des champs chaîne la correspondance la plus proche pour un enregistrement, en se basant sur des valeurs de clé partielles et place le curseur sur cet enregistrement.
Indexation (SetKey)	Efface le contenu du tampon de clés de recherche et active l'état *dsSetKey* pour la table afin de permettre à votre application de spécifier un nouveau critère avant d'exécuter une recherche.

*GotoKey*et *FindKey* sont des fonctions booléennes qui, en cas de succès, placent le curseur sur un enregistrement correspondant et renvoient *True*. Si la recherche n'aboutit pas, le curseur n'est pas déplacé et ces fonctions renvoient *False*.

GotoNearest et *FindNearest* provoquent toujours le repositionnement du curseur sur la première correspondance exacte trouvée ou, si aucune correspondance n'est trouvée, sur le premier enregistrement supérieur au critère de recherche spécifié.

Exécution d'une recherche avec les méthodes Goto

Pour exécuter une recherche en utilisant les méthodes *Goto*, procédez comme suit :

1. Spécifiez l'index à utiliser pour la recherche. C'est le même index qui trie les enregistrements dans l'ensemble de données (voir "Tri des enregistrements avec des index" à la page 18-30). Pour spécifier l'index, utilisez les propriétés *IndexName* ou *IndexFieldNames*.
2. Ouvrez l'ensemble de données.
3. Mettez l'ensemble de données client à l'état *dsSetKey* en appelant la méthode *SetKey*.
4. Spécifiez la ou les valeurs à rechercher dans la propriété *Fields*. *Fields* est un objet *TFields*, qui gère une liste indexée de composants champ auxquels vous pouvez accéder en spécifiant les numéros ordinaux correspondant à chaque colonne. Le premier numéro de colonne d'un ensemble de données est 0.
5. Recherchez et accédez au premier enregistrement trouvé avec *GotoKey* ou *GotoNearest*.

Par exemple, le code ci-dessous, quand il est rattaché à l'événement *OnClick* d'un bouton, utilise la méthode *GotoKey* pour passer au premier enregistrement dont la valeur du premier champ de l'index correspond exactement au texte de la zone de saisie :

```
procedure TSearchDemo.SearchExactClick(Sender: TObject);
begin
  ClientDataSet1.SetKey;
  ClientDataSet1.Fields[0].AsString := Edit1.Text;
  if not ClientDataSet1.GotoKey then
    ShowMessage('Enregistrement non trouvé');
end;
```

GotoNearest est similaire. Elle recherche la première occurrence correspondant à une valeur de champ partielle. Elle ne peut être utilisée que pour des champs chaîne. Par exemple,

```
Table1.SetKey;
Table1.Fields[0].AsString := 'Sm';
Table1.GotoNearest;
```

S'il existe un enregistrement dont la valeur du premier champ indexé commence par les lettres "Sm", le curseur se positionne dessus. Sinon, la position du curseur ne change pas et *GotoNearest* renvoie *False*.

Exécution d'une recherche avec les méthodes Find

Les méthodes *Find* font la même chose que les méthodes *Goto*, à la différence que vous n'avez pas besoin de mettre explicitement l'ensemble de données à l'état *dsSetKey* pour spécifier les valeurs de champ clé servant de base à la recherche. Pour exécuter une recherche en utilisant les méthodes *Find*, procédez comme suit :

1. Spécifiez l'index à utiliser pour la recherche. C'est le même index qui trie les enregistrement dans l'ensemble de données (voir "Tri des enregistrements avec des index" à la page 18-30). Pour spécifier l'index, utilisez les propriétés *IndexName* ou *IndexFieldNames*.
2. Ouvrez l'ensemble de données.
3. Recherchez et accédez au premier enregistrement correspondant trouvé ou au plus proche avec *FindKey* ou *FindNearest*. Les deux méthodes ne prennent qu'un seul argument : une liste de valeurs de champ délimitées par des virgules (chaque valeur correspond à une colonne d'index de la table sous-jacente).

Remarque *FindNearest* ne peut être utilisée que pour les champs chaîne.

Spécification de l'enregistrement en cours après une recherche réussie

Par défaut, une recherche réussie provoque le positionnement du curseur sur le premier enregistrement correspondant au critère de recherche. Si vous préférez, vous pouvez mettre la propriété *KeyExclusive* à *True* afin de positionner le curseur sur l'enregistrement suivant le premier enregistrement correspondant au critère de recherche.

Par défaut, la propriété *KeyExclusive* est à *False*, et positionne le curseur sur le premier enregistrement correspondant au critère de recherche si celle-ci aboutit.

Recherche sur des clés partielles

Si l'ensemble de données comporte plusieurs colonnes clé et si vous voulez rechercher des valeurs dans un sous-ensemble d'une clé, vous devez donner à *KeyFieldCount* une valeur correspondant au nombre de colonnes sur lesquelles la recherche est effectuée. Par exemple, si l'index en cours de l'ensemble de données comprend trois colonnes et si vous voulez effectuer une recherche sur la première colonne seulement, vous devez donner *KeyFieldCount* la valeur 1.

En ce qui concerne les ensembles de données de type table avec des clés multicolonnes, vous ne pouvez rechercher les valeurs que dans des colonnes contiguës, en commençant par la première. Par exemple, pour une clé portant sur trois colonnes, vous pouvez rechercher des valeurs dans la première colonne, puis dans la première et la seconde, ou bien dans la première, la seconde et la troisième, mais pas seulement dans la première et la troisième.

Réitération ou extension d'une recherche

Chaque fois que vous appelez *SetKey* ou *FindKey*, la méthode efface les valeurs précédentes de la propriété *Fields*. Si vous voulez réitérer une recherche à l'aide de champs préalablement définis, ou bien si vous voulez les ajouter aux champs utilisés, faites appel à *EditKey* au lieu de *SetKey* et *FindKey*.

Par exemple, supposons que vous ayez déjà effectué dans la table Employee une recherche basée sur le champ City de l'index "CityIndex". Supposons en outre que "CityIndex" comprenne à la fois les champs *City* et *Company*. Pour trouver un enregistrement avec un nom de société précis dans une ville donnée, utilisez le code suivant :

```
Employee.KeyFieldCount := 2;
Employee.EditKey;
Employee['Company'] := Edit2.Text;
Employee.GotoNearest;
```

Limitation des enregistrements avec des portées

Vous pouvez temporairement voir et modifier un sous-ensemble de n'importe quel ensemble de données en utilisant des filtres (voir "Affichage et édition d'ensembles de données en utilisant des filtres" à la page 18-14). Certains ensembles de données de type table supportent un autre moyen d'accéder à un sous-ensemble d'enregistrements disponibles, les portées.

Les portées ne s'appliquent qu'aux *TTable* et aux ensembles de données client. Malgré leurs similarités, les portées et les filtres ont des utilisations différentes. Les rubriques suivantes présentent les différences entre les portées et les filtres et expliquent comment utiliser les portées.

Présentation des différences entre les portées et les filtres

Les portées et les filtres ont pour effet de restreindre la quantité d'enregistrements visibles, mais leur mode de fonctionnement diffère. Une portée est un ensemble d'enregistrements indexés contigus qui correspondent tous aux valeurs des limites définies. Prenons l'exemple d'une base de données d'employés indexée sur le nom de famille, vous pouvez appliquer une portée pour afficher tous les employés dont le nom de famille est supérieur à "Jones" et inférieur à "Smith". Du fait que les portées dépendent des index, vous devez faire en sorte que l'index en cours puisse permettre de définir la portée. De même que pour spécifier un index pour trier des enregistrements, vous pouvez utiliser la propriété *IndexName* ou *IndexFieldNames* pour affecter l'index sur lequel vous souhaitez définir une portée.

Un filtre est composé d'un ensemble d'enregistrements qui partagent les valeurs spécifiées, indépendamment de l'indexation. Supposons que vous souhaitiez appliquer un filtre sur une base de données d'employés vivant en Californie et ayant travaillé depuis au moins cinq ans dans l'entreprise. Bien qu'ils puissent utiliser les index lors de leur application, les filtres ne dépendent pas d'eux. Les filtres sont appliqués enregistrement par enregistrement au fur et à mesure qu'une application parcourt un ensemble de données.

En principe, les filtres sont plus souples que les portées. Toutefois, les portées peuvent être plus efficaces lorsque les ensembles de données sont très grands et que les enregistrements susceptibles d'intéresser l'application se trouvent déjà dans des groupes d'index contigus. Pour les très grands ensembles de données, il est souvent plus efficace d'utiliser la clause WHERE d'une requête pour sélectionner les données. Pour plus de détails sur la spécification d'une requête, voir "Utilisation d'ensembles de données de type requête" à la page 18-49.

Spécification de portées

Deux moyens s'excluant mutuellement permettent de spécifier une portée :

- Spécifiez le début et la fin séparément en utilisant *SetRangeStart* et *SetRangeEnd*.
- Spécifier les valeurs des deux extrémités simultanément en utilisant *SetRange*.

Définition des valeurs de début de portée

Appelez la procédure *SetRangeStart* pour placer l'ensemble de données à l'état *dsSetKey* et commencez à créer une liste de valeurs de début pour la portée. Après l'appel à *SetRangeStart*, les affectations suivantes de la propriété *Fields* sont traitées comme des valeurs d'index à utiliser lorsque la portée est appliquée. Les champs spécifiés doivent s'appliquer à l'index en cours.

Supposons, par exemple, que votre application utilise un composant *TSQLClientDataSet* appelé *Customers*, lié à la table CUSTOMER, et que vous ayez créé des composants champ persistants pour chaque champ de l'ensemble de données *Customers*. La table CUSTOMER est indexée sur la première colonne (*CustNo*). Dans une fiche de l'application, deux composants de saisie appelés *StartVal* et *EndVal* permettent d'indiquer les valeurs de début et de fin d'une portée. Le code ci-dessous peut être utilisé pour créer une portée et l'appliquer :

```
with Customers do
begin
  SetRangeStart;
  FieldByName('CustNo').AsString := StartVal.Text;
  SetRangeEnd;
  if (Length(EndVal.Text) > 0) then
    FieldByName('CustNo').AsString := EndVal.Text;
  ApplyRange;
end;
```

Ce code vérifie que le texte saisi dans *EndVal* n'est pas NULL avant d'affecter des valeurs à *Fields*. Si le texte saisi dans *StartVal* vaut NULL, tous les enregistrements à partir du début de l'ensemble de données seront inclus,

puisque toutes les valeurs sont supérieures à une valeur NULL. Par contre, si le texte entré dans *EndVal* a une valeur NULL, aucun enregistrement ne sera inclus, puisqu'aucun ne peut être inférieur à cette valeur.

Pour un index à plusieurs colonnes, vous pouvez spécifier une valeur de départ pour tous les champs de l'index ou pour certains de ces champs. Si aucune valeur n'est fournie pour l'un des champs utilisé dans l'index, une valeur NULL est affectée au champ lors de l'application de la portée. Si vous essayez de définir une valeur pour un champ ne figurant pas dans l'index, l'ensemble de données déclenche une exception.

Conseil Pour commencer au début de l'ensemble de données, n'appelez pas *SetRangeStart*.

Pour mettre fin à la spécification du début de la portée, appelez *SetRangeEnd* ou appliquez ou annulez la portée. Pour plus d'informations sur l'application et l'annulation des portées, voir "Application ou annulation d'une portée" à la page 18-40.

Définition des valeurs de fin de portée

Appelez la procédure *SetRangeEnd* pour placer l'ensemble de données à l'état *dsSetKey* et commencez à créer une liste de valeurs de fin pour la portée. Après l'appel à *SetRangeEnd*, les affectations suivantes de la propriété *Fields* sont traitées comme des valeurs d'index à utiliser lorsque la portée est appliquée. Les champs spécifiés doivent s'appliquer à l'index en cours.

Attention Pour qu'une portée se termine sur le dernier enregistrement de l'ensemble de données, spécifiez des valeurs de fin. Si vous ne fournissez pas de valeur de fin, Delphi suppose sinon que la valeur de fin est une valeur NULL. Une portée contenant des valeurs de fin NULL est toujours vide.

La façon la plus simple d'affecter des valeurs de fin de portée est d'appeler la méthode *FieldByName*. Par exemple,

```
with Contacts do
begin
  SetRangeStart;
  FieldByName('LastName').AsString := Edit1.Text;
  SetRangeEnd;
  FieldByName('LastName').AsString := Edit2.Text;
  ApplyRange;
end;
```

Comme dans la spécification des valeurs de début de portée, si vous essayez de définir une valeur pour un champ ne figurant pas dans l'index, l'ensemble de données déclenche une exception.

Pour mettre fin à la spécification de fin de la portée, appliquez ou annulez la portée. Pour plus d'informations sur l'application et l'annulation des portées, "Application ou annulation d'une portée" à la page 18-40.

Définition des valeurs de début et de fin de portée

Plutôt que d'utiliser des appels séparés à *SetRangeStart* et *SetRangeEnd* pour spécifier les limites de la portée, vous pouvez appeler la procédure *SetRange* pour placer l'ensemble de données à l'état *dsSetKey* et définir des valeurs de début et de fin pour la portée par un simple appel.

SetRange prend deux paramètres tableau constants : un ensemble de valeurs de début et un ensemble de valeurs de fin. Par exemple, l'instruction suivante définit une portée basée sur un index de deux colonnes :

```
SetRange([Edit1.Text, Edit2.Text], [Edit3.Text, Edit4.Text]);
```

Pour un index à plusieurs colonnes, vous pouvez spécifier une valeur de départ et de fin pour tous les champs de l'index ou pour certains de ces champs. Si aucune valeur n'est fournie pour l'un des champs utilisé dans l'index, une valeur NULL est affectée au champ lors de l'application de la portée. Pour ne pas spécifier de valeur pour le premier champ de l'index et spécifier des valeurs pour les champs suivants, passez une valeur NULL ou une valeur vide au premier champ.

Spécifiez toujours des valeurs de fin pour une portée, même si vous voulez qu'elle se termine sur le dernier enregistrement de l'ensemble de données. L'ensemble de données suppose sinon que la valeur de fin est une valeur NULL. Une portée contenant des valeurs de fin NULL est toujours vide car la portée de départ est supérieure ou égale à la portée de fin.

Spécification d'une portée à partir de clés partielles

Si une clé est composée d'un ou de plusieurs champs chaîne, les méthodes *SetRange* supportent les clés partielles. Par exemple, si un index est basé sur les colonnes *LastName* et *FirstName*, les spécifications de portée suivantes sont valides :

```
Contacts.SetRangeStart;
Contacts['LastName'] := 'Smith';
Contacts.SetRangeEnd;
Contacts['LastName'] := 'Zzzzzz';
Contacts.ApplyRange;
```

Ce code inclut tous les enregistrements dans une portée où *LastName* est supérieur ou égal à "Smith". La spécification des valeurs peut également se présenter ainsi :

```
Contacts['LastName'] := 'Sm';
```

Cette instruction inclut les enregistrements où *LastName* est supérieur ou égal à "Sm".

Inclusion ou exclusion d'enregistrements correspondant aux valeurs d'une portée

Par défaut, une portée inclut tous les enregistrements supérieurs ou égaux à la portée de début spécifiée et inférieurs ou égaux à la portée de fin spécifiée. Ce comportement est contrôlé par la propriété *KeyExclusive*. Par défaut, *KeyExclusive* est à *False*.

Si vous préférez, vous pouvez donner à la propriété *KeyExclusive* d'un ensemble de données la valeur *True* pour qu'elle exclue les enregistrements égaux à la portée de fin. Par exemple,

```
Contacts.KeyExclusive := True;
Contacts.SetRangeStart;
Contacts['LastName'] := 'Smith';
Contacts.SetRangeEnd;
Contacts['LastName'] := 'Tyler';
Contacts.ApplyRange;
```

Ce code inclut tous les enregistrements d'une portée pour lesquels *LastName* est supérieur ou égal à "Smith" et inférieur à "Tyler".

Modification d'une portée

Deux fonctions permettent de modifier les conditions relatives aux limites d'une portée : *EditRangeStart*, pour modifier les valeurs de début d'une portée, et *EditRangeEnd*, pour modifier les valeurs de fin de la portée.

Le processus d'édition et d'application d'une portée se déroule comme suit :

1. Mettez l'ensemble de données à l'état *dsSetKey* et modifiez la valeur de début de l'index pour la portée.
2. Modifiez la valeur de fin de l'index pour la portée.
3. Appliquez la portée à l'ensemble de données.

Vous pouvez modifier les valeurs de début ou de fin d'une portée, ou bien modifier les deux à la fois. Si vous modifiez les conditions relatives aux limites d'une portée actuellement appliquée à l'ensemble de données, les modifications ne sont pas appliquées tant que vous n'avez pas rappelé *ApplyRange*.

Modification du début de la portée

Appelez la procédure *EditRangeStart* pour placer l'ensemble de données à l'état *dsSetKey* et commencez à modifier la liste en cours des valeurs de début de la portée. Après avoir appelé *EditRangeStart*, les affectations suivantes de la propriété *Fields* écrasent les valeurs d'index en cours lors de l'application de la portée.

Conseil Si vous avez initialement créé une portée de début basée sur une clé partielle, vous pouvez utiliser *EditRangeStart* pour étendre la valeur de début de la portée. Pour plus d'informations sur les portées basées sur des clés partielles, voir "Spécification d'une portée à partir de clés partielles" à la page 18-38.

Modification de la fin de la portée

Appelez la procédure *EditRangeEnd* pour placer l'ensemble de données à l'état *dsSetKey* et commencez à modifier la liste en cours de valeurs de fin de portée. Après avoir appelé *EditRangeEnd*, les affectations suivantes de la propriété *Fields* sont traitées comme des valeurs d'index de fin à utiliser lors de l'application d'une portée.

Application ou annulation d'une portée

Lorsque vous appelez *SetRangeStart* ou *EditRangeStart* pour spécifier le début d'une portée, ou bien *SetRangeEnd* ou *EditRangeEnd* pour spécifier la fin d'une portée, l'ensemble de données passe à l'état *dsSetKey*. Il y demeure jusqu'à ce que vous appliquiez ou annuliez la portée.

Application d'une portée

Lorsque vous spécifiez une portée, les conditions relatives aux limites que vous définissez n'entrent en vigueur que lorsque vous appliquez la portée. Pour qu'une portée prenne effet, appelez la méthode *ApplyRange*. *ApplyRange* limite immédiatement l'utilisateur qui ne peut plus visualiser et accéder qu'aux données contenues dans le sous-ensemble de l'ensemble de données.

Annulation d'une portée

La méthode *CancelRange* met fin à l'application d'une portée et restaure l'accès à la totalité de l'ensemble de données. Même si l'annulation d'une portée restaure l'accès à tous les enregistrements de l'ensemble de données, les conditions relatives aux limites de la portée sont toujours disponibles afin que vous puissiez réappliquer la portée ultérieurement. Les limites d'une portée sont préservées jusqu'à ce que vous fournissiez de nouvelles limites ou modifiiez les limites existantes. Par exemple, le code suivant est correct :

```
⋮
MyTable.CancelRange;
⋮
{permet d'utiliser cette portée ultérieurement. Pas besoin d'appeler SetRangeStart, etc.}
MyTable.ApplyRange;
⋮
```

Création de relations maître/détail

Les ensembles de données de type table peuvent être liés par le biais de relations maître/détail. Lorsque vous définissez une relation maître/détail, vous reliez deux ensembles de données de sorte que tous les enregistrements de l'un (ensemble de données détail) correspondent toujours à un enregistrement unique dans l'autre (ensemble de données maître).

Les ensembles de données de type table prennent en charge les relations maître/détail selon deux processus très différents :

- Tous les ensembles de données de type table peuvent agir sur la partie détail d'un autre ensemble de données en reliant les curseurs. Ce processus est décrit dans "Comment faire de la table la partie détail d'un autre ensemble de données" ci-après.
- *TTable*, *TSQLTable* et tous les ensembles de données client peuvent agir sur la partie maître d'une relation maître/détail qui utilise des tables imbriquées. Ce processus est décrit dans "Utilisation de tables détail imbriquées" à la page 18-43.

Chacune de ces approches présente ses propres avantages. La liaison des curseurs vous permet de créer des relations maître/détail dans lesquelles la table maître est un ensemble de données de n'importe quel type. Avec les détails imbriqués, le type d'ensemble de données qui peut agir en tant que table détail est limité, mais ils offrent davantage de façons d'afficher les données. Si le maître est un ensemble de données client, les détails imbriqués fournissent un mécanisme plus robuste pour l'application des mises à jour en mémoire cache.

Comment faire de la table la partie détail d'un autre ensemble de données

Les propriétés *MasterSource* et *MasterFields* d'un ensemble de données de type table peuvent être utilisées pour établir des relations un-à-plusieurs entre deux ensembles de données.

La propriété *MasterSource* permet de spécifier une source de données utilisée par la table pour extraire des données de la table maître. Cette source de données peut être liée à n'importe quel type d'ensemble de données. Par exemple, en spécifiant dans cette propriété la source de données d'une requête, vous pourrez lier un ensemble de données client en tant que détail de la requête, de sorte que cet ensemble de données client se charge du suivi des événements survenant dans la requête.

L'ensemble de données est lié à la table maître par son index en cours. Avant d'indiquer quels champs de l'ensemble de données maître sont suivis par l'ensemble de données détail, définissez l'index de l'ensemble de données détail commençant par les champs correspondants. Vous pouvez utiliser la propriété *IndexName* ou la propriété *IndexFieldNames*.

Lorsque vous avez spécifié l'index à utiliser, servez-vous de la propriété *MasterFields* pour indiquer la ou les colonnes de l'ensemble de données maître correspondant aux champs d'index de la table détail. Pour lier des ensembles de données à partir de plusieurs noms de colonnes, séparez les noms des champs par des points-virgules :

```
Parts.MasterFields := 'OrderNo;ItemNo';
```

Pour créer des liens fiables entre deux ensembles de données, vous pouvez utiliser le concepteur de liaisons de champs. Pour ce faire, après avoir spécifié un *MasterSource* et un index, double-cliquez sur la propriété *MasterFields* dans l'inspecteur d'objets.

En suivant les procédures décrites ci-dessous, vous pourrez créer une fiche dans laquelle un utilisateur pourra parcourir les enregistrements sur des clients et affichera toutes les commandes passées par le client en cours. La table maître est *CustomersTable*, la table détail *OrdersTable*. L'exemple utilise le composant BDE *TTable*, mais vous pouvez utiliser les mêmes méthodes pour lier n'importe quels ensembles de données de type table.

1. Placez deux composants *TTable* et deux composants *TDataSource* dans un module de données.
2. Définissez les propriétés du premier composant *TTable* comme suit :
   - *DatabaseName* : DBDEMOS

- *TableName* : CUSTOMER
- *Name* : CustomersTable

3 Définissez les propriétés du deuxième composant *TTable* comme suit :

- *DatabaseName* : DBDEMOS
- *TableName* : ORDERS
- *Name* : OrdersTable

4 Définissez les propriétés du premier composant *TDataSource* comme suit :

- *Name* : CustSource
- *DataSet* : CustomersTable

5 Définissez les propriétés du deuxième composant *TDataSource* comme suit :

- *Name* : OrdersSource
- *DataSet* : OrdersTable

6 Placez deux composants *TDBGrid* sur une fiche.

7 Choisissez Fichier | Utiliser l'unité pour indiquer que la fiche doit utiliser le module de données.

8 Donnez à la propriété *DataSource* de la première grille la valeur "CustSource", et donnez à la propriété *DataSource* de la deuxième grille la valeur "OrdersSource".

9 Définissez la propriété *MasterSource* de *OrdersTable* par "CustSource". Cela lie la table CUSTOMER (la table maître) à la table ORDERS (la table détail).

10 Double-cliquez dans la zone de la valeur de la propriété *MasterFields* dans l'inspecteur d'objets pour appeler le concepteur de liaisons de champs afin de définir les propriétés suivantes :

- Dans le champ Index disponibles, choisissez *CustNo* pour lier les deux tables sur le champ *CustNo*.
- Sélectionnez *CustNo* dans les deux listes de champs Champs détail et Champs maître.
- Cliquez sur Ajouter pour ajouter cette condition de jointure. Dans la liste Champs joints,
  "CustNo -> CustNo" apparaît.
- Cliquez sur OK pour valider vos sélections et quitter le concepteur de liaisons de champs.

11 Définissez les propriétés *Active* de *CustomersTable* et de *OrdersTable* par *True* pour afficher les données dans les grilles de la fiche.

12 Compilez l'application et exécutez-la.

Si vous lancez l'application maintenant, vous pouvez constater que les tables sont liées et que quand vous vous déplacez sur un nouvel enregistrement de la table CUSTOMER, seuls apparaissent les enregistrements de la table ORDERS appartenant au client en cours.

Utilisation de tables détail imbriquées

Une table imbriquée est un ensemble de données détail qui est la valeur d'un champ unique d'un autre ensemble de données (maître). Pour les ensembles de données qui représentent les données du serveur, un ensemble de données détail imbriqué peut être utilisé uniquement pour un champ d'un ensemble de données du serveur. Les composants *TClientDataSet* ne représentent pas les données d'un serveur, mais ils peuvent contenir des champs d'ensembles de données si vous créez pour eux un ensemble de données contenant des détails imbriqués, ou s'ils reçoivent des données d'un fournisseur lié à la table maître dans une relation maître/détail.

Remarque Pour *TClientDataSet*, l'utilisation des ensembles détail imbriqués s'impose si vous souhaitez appliquer les mises à jour à partir de tables maître et détail dans un serveur de base de données.

Pour utiliser des ensembles détail imbriqués, la propriété *ObjectView* de l'ensemble de données client doit être *True*. Lorsque votre ensemble de données de type table contient des ensembles de données détail imbriqués, *TDBGrid* permet d'afficher les détails imbriqués dans une fenêtre surgissante. Pour plus d'informations, voir "Affichage des champs ensemble de données" à la page 19-30.

Vous pouvez aussi afficher et éditer ces ensembles de données détail dans des contrôles orientés données en utilisant un composant ensemble de données séparé pour l'ensemble détail. A la conception, créez des champs persistants pour les champs de votre ensemble de données (maître), en utilisant l'éditeur de champs : cliquez avec le bouton droit sur l'ensemble de données maître et choisissez Editeur de champs. Ajoutez un nouveau champ persistant à votre ensemble de données client en cliquant avec le bouton droit et en choisissant Ajouter des champs. Définissez votre nouveau champ avec le type champ ensemble de données. Dans l'éditeur de champs, définissez la structure de la table détail. Vous devez aussi ajouter des champs persistants pour tous les autres champs utilisés dans votre ensemble de données détail.

Le composant ensemble de données pour la table détail est un ensemble de données descendant d'un type autorisé pour la table maître. Les composants *TTable* acceptent uniquement les composants*TNestedDataSet* comme ensembles de données imbriqués. Les composants *TSQLTable* acceptent d'autres composants *TSQLTable*. Les composants *TClientDataset* acceptent d'autres ensembles de données client. Choisissez un ensemble de données du type approprié dans la palette de composants et ajoutez-le à votre fiche ou à votre module de données. Affectez à la propriété *DataSetField* de cet ensemble de données détail le champ persistant DataSet de l'ensemble de données maître. Enfin, placez un composant source de données dans le module de données ou sur la fiche, et définissez sa propriété *DataSet* par l'ensemble de données détail. Les contrôles orientés données peuvent utiliser cette source de données pour accéder aux données de l'ensemble de données détail.

Contrôle des accès en lecture/écriture aux tables

Par défaut, quand un ensemble de données de type table est ouvert, il demande l'accès en lecture et en écriture à la table de la base de données sous-jacente. Selon les caractéristiques de la table de la base de données sous-jacente, le droit d'écriture demandé peut ne pas être accordé (par exemple, quand vous demandez l'accès en écriture à une table SQL d'un serveur distant et que le serveur a limité l'accès à la table en lecture seulement).

Remarque Ce n'est pas vrai pour *TClientDataSet*, qui détermine si les utilisateurs peuvent modifier des données à partir d'informations procurées par le fournisseur de l'ensemble de données avec les paquets de données. Ce n'est pas vrai non plus pour *TSQLTable*, qui est un ensemble de données unidirectionnel, et donc toujours en lecture seule.

Quand une table s'ouvre, vous pouvez vérifier la propriété *CanModify* pour savoir si la base de données sous-jacente (ou le fournisseur de l'ensemble de données) autorise les utilisateurs à modifier les données de la table. Si *CanModify* vaut *False*, l'application ne peut pas écrire dans la base de données. Si *CanModify* vaut *True*, votre application peut écrire dans la base de données si la propriété *ReadOnly* de la table vaut *False*.

ReadOnly détermine si un utilisateur peut à la fois voir et modifier les données. Quand *ReadOnly* vaut *False* (valeur par défaut), un utilisateur peut à la fois voir et modifier les données. Pour limiter un utilisateur à la visualisation des données, définissez *ReadOnly* par *True* avant d'ouvrir la table.

Remarque *ReadOnly* est implémentée sur tous les ensembles de type table, à l'exception de *TSQLTable*, qui est toujours en lecture seule.

Création et suppression des tables

Certains ensembles de données de type table vous permettent de créer et de supprimer les tables sous-jacentes, à la conception ou à l'exécution. En général, les tables de bases de données sont créées et supprimées par l'administrateur de la base. Mais, il peut être pratique au cours du développement et des tests de créer et de détruire des tables utilisées par votre application.

Création de tables

TTable et *TIBTable* vous permettent tous deux de créer la table de base de données sous-jacente sans utiliser SQL. De même, *TClientDataSet* vous permet de créer un ensemble de données quand vous ne travaillez pas avec un fournisseur d'ensemble de données. En utilisant *TTable* et *TClientDataSet*, vous pouvez créer la table pendant la conception ou pendant l'exécution. *TIBTable* vous permet de créer des tables uniquement à l'exécution.

Avant de créer la table, vous devez définir des propriétés pour spécifier la structure de la table que vous créez. En particulier, vous devez spécifier :

- La base de données qui contiendra la nouvelle table. Pour *TTable*, vous spécifiez la base de données en utilisant la propriété *DatabaseName*. Pour *TIBTable*, vous devez utiliser un composant *TIBDataBase*, qui est affecté à la propriété *Database*. (Les ensembles de données client n'utilisent pas de base de données.)
- Le type de base de données (*TTable* seulement). Définissez la propriété *TableType* par le type de table souhaité. Pour les tables Paradox, dBASE ou ASCII, définissez *TableType* par *ttParadox*, *ttDBase* ou *ttASCII*, respectivement. Pour tous les autres types de tables, définissez *TableType* par *ttDefault*.
- Le nom de la table que vous voulez créer. *TTable* et *TIBTable* ont tous deux une propriété *TableName* pour stocker le nom de la nouvelle table. Les ensembles de données client n'utilisent pas de nom de table, mais vous devez spécifier la propriété *FileName* avant d'enregistrer la nouvelle table. Si vous créez une table qui reprend le nom d'une table existante, la table existante et toutes ses données sont remplacées par la nouvelle table. L'ancienne table et ses données ne peuvent pas être récupérées. Pour éviter de remplacer une table existante, vous devez vérifier la propriété *Exists* à l'exécution. *Exists* est disponible uniquement sur *TTable* et *TIBTable*.
- Les champs de la nouvelle table. Pour ce faire, vous pouvez procéder de deux façons :
  - Vous pouvez ajouter des définitions de champs à la propriété *FieldDefs*. En conception double-cliquez sur la propriété *FieldDefs* dans l'inspecteur d'objets pour afficher l'éditeur de collection. Utilisez l'éditeur de collection pour ajouter, supprimer ou modifier les propriétés de définitions de champs. A l'exécution, effacez toutes les définitions de champs existantes et utilisez ensuite la méthode *AddFieldDef* pour ajouter chaque nouvelle définition de champ. Pour chaque nouvelle définition de champ, définissez les propriétés de l'objet *TFieldDef* pour spécifier les attributs du champ.
  - Vous pouvez utiliser à la place des composants champs persistants. En conception double-cliquez sur l'ensemble de données pour afficher l'éditeur de champs. Dans l'éditeur de champs, cliquez avec le bouton droit et choisissez la commande Nouveau champ. Décrivez les principales propriétés de votre champ. Une fois que le champ est créé, vous pouvez modifier ses propriétés dans l'inspecteur d'objets en le sélectionnant dans l'éditeur de champs.
- Les index de la nouvelle table (facultatif). En conception double-cliquez sur la propriété *IndexDefs* dans l'inspecteur d'objets pour afficher l'éditeur de collection. Utilisez l'éditeur de collection pour ajouter, supprimer ou modifier les propriétés de définitions d'index. A l'exécution, effacez toutes les définitions d'index existantes et utilisez ensuite la méthode *AddIndexDef* pour ajouter chaque nouvelle définition d'index. Pour chaque nouvelle définition d'index, définissez les propriétés de l'objet *TIndexDef* pour spécifier les attributs de l'index.

Remarque Vous ne pouvez pas définir d'index pour la nouvelle table si vous utilisez des composants champ persistant au lieu d'objets définition de champ.

Pour créer la table pendant la conception, cliquez avec le bouton droit sur l'ensemble de données et choisissez Créer table (*TTable*) ou Créer ensemble de données (*TClientDataSet*). Cette commande n'apparaît pas dans le menu contextuel tant que n'avez pas spécifié toutes les informations nécessaires.

Pour créer la table à l'exécution, appelez la méthode *CreateTable* (*TTable* et *TIBTable*) ou la méthode *CreateDataSet* (*TClientDataSet*).

Remarque Vous pouvez établir les définitions pendant la conception, puis appeler la méthode *CreateTable* (ou *CreateDataSet*) à l'exécution pour créer la table. Cependant, pour faire cela vous devrez indiquer que les définitions spécifiées à l'exécution seront enregistrées avec le composant ensemble de données. (par défaut, les définitions de champs et d'index sont générées de façon dynamique à l'exécution). Spécifiez que les définitions doivent être enregistrées avec l'ensemble de données en définissant sa propriété *StoreDefs* par *True*.

Conseil Si vous utilisez *TTable*, vous pouvez pré-charger les définitions de champs et d'index d'une table existante pendant la conception. Définissez les propriétés *DatabaseName* et *TableName* pour spécifier la table existante. Cliquez avec le bouton droit sur le composant table et choisissez Mettre à jour la définition de table. Cela définit automatiquement les valeurs des propriétés *FieldDefs* et *IndexDefs* pour décrire les champs et les index de la table existante. Ensuite, réinitialisez *DatabaseName* et *TableName* pour spécifier la table que vous voulez créer, en annulant toutes les demandes de renommer la table existante.

Remarque Quand vous créez des tables Oracle8, vous ne pouvez pas créer de champs objets (champs ADT, champs de tableaux et champs d'ensembles de données).

Le code suivant crée une nouvelle table à l'exécution et l'associe à l'alias DBDEMOS. Avant de créer la nouvelle table, il vérifie que le nom de la table fourni ne correspond pas au nom d'une table existante :

```
var
  TableFound: Boolean;
begin
  with TTable.Create(nil) do // crée un composant TTable temporaire
  begin
    try
      { définit les propriétés du composant TTable temporaire }
      Active := False;
      DatabaseName := 'DBDEMOS';
      TableName := Edit1.Text;
      TableType := ttDefault;
      { définit les champs de la nouvelle table }
      FieldDefs.Clear;
      with FieldDefs.AddFieldDef do begin
        Name := 'First';
        DataType := ftString;
        Size := 20;
        Required := False;
      end;
      with FieldDefs.AddFieldDef do begin
```

```
        Name := 'Second';
        DataType := ftString;
        Size := 30;
        Required := False;
      end;
      { définit les index de la nouvelle table }
      IndexDefs.Clear;
      with IndexDefs.AddIndexDef do begin
        Name := '';
        Fields := 'First';
        Options := [ixPrimary];
      end;
      TableFound := Exists; // vérifie si la table existe déjà
      if TableFound then
        if MessageDlg('Remplacer la table existante ' + Edit1.Text + ' ?',
            mtConfirmation, mbYesNoCancel, 0) = mrYes then
          TableFound := False;
      if not TableFound then
        CreateTable; // crée la table
    finally
      Free; // détruit le composant TTable temporaire à la fin
    end;
  end;
end;
```

Suppression de tables

TTable et *TIBTable* vous permettent de supprimer des tables de la base de données sous-jacente sans utiliser SQL. Pour supprimer une table à l'exécution, appelez la méthode *DeleteTable* de l'ensemble de données. Par exemple, l'instruction suivante supprime la table sous-jacente d'un ensemble de données :

```
CustomersTable.DeleteTable;
```

Attention Quand vous supprimez une table avec *DeleteTable*, la table et toutes ses données disparaissent.

Si vous utilisez *TTable*, vous pouvez aussi supprimer des tables pendant la conception : cliquez avec le bouton droit sur le composant table et sélectionnez Supprimer une table dans le menu contextuel. L'option de menu Supprimer une table n'est présente que si le composant table représente une table de base de données existante (les propriétés *DatabaseName* et *TableName* spécifient une table existante).

Vidage des tables

De nombreux ensembles de données de type table ont une seule méthode qui vous permet de supprimer toutes les lignes de données de la table.

- Pour *TTable* et *TIBTable*, vous pouvez supprimer tous les enregistrements en appelant la méthode *EmptyTable* à l'exécution :

```
PhoneTable.EmptyTable;
```

- Pour *TADOTable*, vous pouvez utiliser la méthode *DeleteRecords*.

```
PhoneTable.DeleteRecords;
```

- Pour *TSQLTable*, vous pouvez aussi utiliser la méthode *DeleteRecords*. Notez, cependant, que la version *TSQLTable* de *DeleteRecords* ne prend jamais de paramètre.

```
PhoneTable.DeleteRecords;
```

- Pour les ensembles de données client, vous pouvez utiliser la méthode *EmptyDataSet*.

```
PhoneTable.EmptyDataSet;
```

Remarque Pour les tables des serveurs SQL, ces méthodes réussissent uniquement si vous avez le privilège DELETE pour cette table.

Attention Quand vous videz un ensemble de données, les données que vous supprimez disparaissent définitivement.

Synchronisation des tables

Si vous avez deux ensembles de données ou plus qui représentent la même table de base de données mais ne partagent pas un composant source de données, alors, chaque ensemble de données a sa propre vue des données et son propre enregistrement en cours. Au fur et à mesure que les utilisateurs accèdent aux enregistrements par le biais de chaque ensemble de données, les enregistrements en cours des composants diffèrent.

Si les ensembles de données sont tous des instances de *TTable*, ou tous des instances de *TIBTable*, ou tous des ensembles de données client, vous pouvez forcer l'enregistrement en cours de chacun de ces ensembles de données à être le même, en appelant la méthode *GotoCurrent*. *GotoCurrent* définit son propre enregistrement en cours de l'ensemble de données par l'enregistrement en cours de l'ensemble de données correspondant. Par exemple, le code suivant définit l'enregistrement en cours de *CustomerTableOne* comme le même que l'enregistrement en cours de *CustomerTableTwo* :

```
CustomerTableOne.GotoCurrent(CustomerTableTwo);
```

Conseil Si votre application doit synchroniser des ensembles de données de cette manière, placez les ensembles de données dans un module de données et ajoutez l'unité du module de données à la clause uses de chaque unité qui accède aux tables.

Pour synchroniser des ensembles de données de fiches distinctes, vous devez ajouter l'unité d'une fiche à la clause uses de l'autre, et qualifier au moins un des ensembles de données par le nom de sa fiche. Par exemple :

```
CustomerTableOne.GotoCurrent(Form2.CustomerTableTwo);
```

Utilisation d'ensembles de données de type requête

Pour utiliser un ensemble de données de type requête,

1. Placez le composant approprié dans un module de données ou sur une fiche, et attribuez à sa propriété *Name* une valeur unique appropriée à votre application.
2. Identifiez le serveur de base de données à qui envoyer la requête. Chaque ensemble de données de type requête fait cela différemment, mais, en général, il faut spécifier un composant de base de données :
   - Pour *TQuery*, spécifiez un composant *TDatabase* ou un alias BDE en utilisant la propriété *DatabaseName*.
   - Pour *TADOQuery*, spécifiez un composant *TADOConnection* en utilisant la propriété *Connection*.
   - Pour *TSQLQuery*, spécifiez un composant *TSQLConnection* en utilisant la propriété *SQLConnection*.
   - Pour *TIBQuery*, spécifiez un composant *TIBConnection* en utilisant la propriété *Database*.

   Pour plus d'informations sur l'utilisation des composants connexion aux bases de données, voir chapitre 17, "Connexion aux bases de données".
3. Spécifiez une instruction SQL dans la propriété *SQL* de l'ensemble de données et, éventuellement, spécifiez les paramètres de l'instruction. Pour plus d'informations, voir "Spécification de la requête" à la page 18-50 et "Utilisation de paramètres dans les requêtes" à la page 18-52.
4. Si les données de la requête doivent être utilisées avec des contrôles de données visuels, ajoutez un composant source de données au module de données et définissez sa propriété *DataSet* par l'ensemble de données de type requête. Le composant source de données suit les résultats de la requête (que l'on appelle *ensemble de résultats*) que les composants orientés données affichent. Connectez les composants orientés données à la source de données à l'aide de leurs propriétés *DataSource* et *DataField*.
5. Activez le composant requête. Pour les requêtes qui renvoient un ensemble de résultats, utilisez la propriété *Active* ou la méthode *Open*. Pour exécuter des requêtes qui effectuent seulement une action sur une table et ne renvoient aucun ensemble de résultats, utilisez la méthode *ExecSQL* à l'exécution. Si vous prévoyez d'exécuter la requête plusieurs fois, vous pouvez appeler *Prepare* pour initialiser la couche d'accès aux données et les valeurs des paramètres de liaison dans la requête. Pour plus d'informations sur la préparation d'une requête, voir "Préparation des requêtes" à la page 18-56.

Spécification de la requête

Pour les véritables ensembles de données de type requête, vous utilisez la propriété *SQL* pour spécifier l'instruction SQL à exécuter par l'ensemble de données. Certains ensembles de données, comme *TADODataSet*, *TSQLDataSet*, et les ensembles de données client, utilisent une propriété *CommandText* pour faire la même chose.

La plupart des requêtes qui renvoient des enregistrements sont des commandes SELECT. Généralement, elles définissent les champs à inclure, les tables dans lesquelles les sélectionner, les conditions qui limitent les enregistrements à inclure, l'ordre de l'ensemble de données résultant. Par exemple :

```
SELECT CustNo, OrderNo, SaleDate
FROM Orders
WHERE CustNo = 1225
ORDER BY SaleDate
```

Les requêtes qui ne renvoient pas d'enregistrements contiennent des instructions qui utilisent des instructions DDL (langage de définition des données) ou DML (langage de manipulation des données) autres que les instructions SELECT (Par exemple, les commandes INSERT, DELETE, UPDATE, CREATE INDEX et ALTER TABLE ne renvoient aucun enregistrement). Le langage utilisé dans les commandes est spécifique au serveur mais généralement conforme au standard SQL-92 du langage SQL.

La commande SQL que vous exécutez doit être acceptable pour le serveur que vous utilisez. Les ensembles de données n'évaluent pas la commande SQL et ne l'exécutent pas. Ils transmettent simplement la commande au serveur pour son exécution. Dans la plupart des cas, la commande SQL doit être constituée d'une seule instruction SQL complète, même si cette instruction peut être aussi complexe que nécessaire (par exemple, une instruction SELECT avec une clause WHERE qui utilise plusieurs opérateurs logiques imbriqués comme AND et OR). Certains serveurs supportent également la syntaxe "batch" qui autorise plusieurs instructions ; si votre serveur supporte cette syntaxe, vous pouvez entrer plusieurs instructions pour spécifier la requête.

Les instructions SQL utilisées par les requêtes peuvent être textuelles ou peuvent contenir des paramètres à remplacer. Les requêtes qui utilisent des paramètres sont appelées *requêtes paramétrées*. Quand vous utilisez des requêtes paramétrées, les valeurs réelles affectées aux paramètres sont insérées dans la requête avant d'exécuter cette dernière. L'utilisation des requêtes paramétrées est très souple, car vous pouvez, à l'exécution, changer la vue d'un utilisateur et accéder aux données à la volée, sans avoir à modifier l'instruction SQL. Pour plus d'informations sur les requêtes paramétrées, voir "Utilisation de paramètres dans les requêtes" à la page 18-52.

Spécification d'une requête en utilisant la propriété SQL

Quand vous utilisez un véritable ensemble de données de type requête *(TQuery, TADOQuery, TSQLQuery*, ou *TIBQuery)*, affectez la requête à la propriété *SQL*. La propriété *SQL* est un objet *TStrings*. Chaque chaîne différente de cet objet

TStrings est une ligne distincte de la requête. L'utilisation de plusieurs lignes n'affecte pas la façon dont la requête s'exécute sur le serveur, mais peut faciliter la modification et le débogage de la requête si l'instruction est divisée en unités logiques :

```
MyQuery.Close;
MyQuery.SQL.Clear;
MyQuery.SQL.Add('SELECT CustNo, OrderNO, SaleDate');
MyQuery.SQL.Add(' FROM Orders');
MyQuery.SQL.Add('ORDER BY SaleDate');
MyQuery.Open;
```

Le code ci-après montre la modification d'une seule ligne dans une instruction SQL existante. Dans ce cas, la clause ORDER BY existe déjà dans la troisième ligne de l'instruction. Elle est référencée via la propriété *SQL* en utilisant un index de 2.

```
MyQuery.SQL[2] := 'ORDER BY OrderNo';
```

Remarque L'ensemble de données doit être fermé lorsque vous spécifiez ou modifiez la propriété SQL.

Pendant la conception, utilisez l'éditeur de liste de chaînes pour spécifier la requête. Cliquez sur le bouton points de suspension de la propriété *SQL*, dans l'inspecteur d'objets, pour afficher l'éditeur de liste de chaînes.

Remarque Avec certaines versions de Delphi, si vous utilisez *TQuery*, vous pouvez aussi utiliser le constructeur SQL pour construire une requête basée sur une représentation visible des tables et des champs d'une base de données. Pour utiliser le constructeur SQL, sélectionnez le composant requête, cliquez avec le bouton droit pour appeler le menu contextuel et choisissez l'éditeur de requêtes graphique. Pour apprendre à utiliser le constructeur de requêtes, ouvrez-le et faites appel à son aide en ligne.

Comme la propriété *SQL* est un objet *TStrings*, vous pouvez charger le texte de la requête à partir d'un fichier, en appelant la méthode *TStrings.LoadFromFile* :

```
MyQuery.SQL.LoadFromFile('custquery.sql');
```

Vous pouvez aussi utiliser la méthode *Assign* de la propriété *SQL* pour copier le contenu d'un objet liste de chaînes dans la propriété *SQL*. La méthode *Assign* efface automatiquement le contenu en cours de la propriété *SQL* avant de copier la nouvelle instruction :

```
MyQuery.SQL.Assign(Memo1.Lines);
```

Spécification d'une requête en utilisant la propriété CommandText

Quand vous utilisez *TADODataSet*, *TSQLDataSet* ou un ensemble de données client, affectez le texte d'une instruction de requête à la propriété *CommandText* :

```
MyQuery.CommandText := 'SELECT CustName, Address FROM Customer';
```

Pendant la conception, vous pouvez taper directement la requête dans l'inspecteur d'objets, ou, si l'ensemble de données SQL dispose déjà d'une connexion à la base de données active, vous pouvez cliquer sur le bouton points de suspension de la propriété *CommandText* pour afficher l'éditeur de

CommandText. L'éditeur de CommandText énumère les tables disponibles et les champs contenus dans ces tables pour faciliter la composition de vos requêtes.

Utilisation de paramètres dans les requêtes

Une instruction SQL paramétrée contient des paramètres, ou variables, dont les valeurs peuvent être modifiées pendant la conception ou pendant l'exécution. Les paramètres peuvent remplacer les valeurs des données, comme celles qui sont utilisées pour les comparaisons dans une clause WHERE, qui apparaissent dans une instruction SQL. En général, les paramètres représentent les valeurs des données qui sont passées à l'instruction. Par exemple, dans l'instruction INSERT suivante, les valeurs à insérer sont passées sous forme de paramètres :

```
INSERT INTO Country (Name, Capital, Population)
VALUES (:Name, :Capital, :Population)
```

Dans cette instruction SQL, *:Name*, *:Capital* et *:Population* sont des marques de réservation pour les valeurs réelles fournies à l'instruction par votre application au cours de l'exécution. Remarquez que les noms des paramètres commencent par un deux points. Le deux points est requis pour distinguer les noms de paramètres des valeurs littérales. Vous pouvez aussi inclure des paramètres non nommés en insérant un point d'interrogation (?) dans votre requête. Les paramètres non nommés sont identifiés par leur position, puisqu'ils n'ont pas de nom.

Avant que l'ensemble de données puisse exécuter la requête, vous devez fournir une valeur pour chaque paramètre figurant dans le texte de la requête. *TQuery*, *TIBQuery*, *TSQLQuery* et les ensembles de données client utilisent la propriété *Params* pour stocker ces valeurs. *TADOQuery* utilise à la place la propriété *Parameters*. *Params* (ou *Parameters*) est une collection d'objets paramètre (*TParam* ou *TParameter*), où chaque objet représente un seul paramètre. Quand vous spécifiez le texte de la requête, l'ensemble de données génère cet ensemble d'objets paramètre et (selon le type de l'ensemble de données) initialise les propriétés qu'il peut déduire de la requête.

Remarque Vous pouvez supprimer la génération automatique des objets paramètre en réponse au changement du texte de la requête, en définissant la propriété *ParamCheck* par *False*. C'est utile pour les instructions DDL (langage de définition des données) contenant des paramètres qui font partie de l'instruction DDL et ne sont pas des paramètres de la requête. Par exemple, l'instruction DDL pour créer une procédure stockée peut définir des paramètres faisant partie de cette procédure stockée. En définissant *ParamCheck* par *False*, vous empêchez ces paramètres d'être pris par erreur pour les paramètres de la requête.

Les valeurs de paramètres doivent être liées à l'instruction SQL avant sa première exécution. Les composants requête le font pour vous automatiquement si vous n'appelez pas explicitement la méthode *Prepare* avant l'exécution d'une requête.

Conseil C'est une bonne habitude de programmation de fournir aux paramètres des noms de variable correspondant aux noms des colonnes avec lesquelles ils sont associés. Par exemple, si un nom de colonne est "Number," alors le paramètre

qui lui correspond peut être ":Number". L'utilisation de noms correspondants est particulièrement importante si l'ensemble de données utilise une source de données pour obtenir les valeurs des paramètres d'un autre ensemble de données. Ce processus est décrit dans "Etablissement de relations maître/détail en utilisant des paramètres" à la page 18-55.

Fourniture des paramètres pendant la conception

Au cours de la conception, vous pouvez spécifier les valeurs des paramètres en utilisant l'éditeur de collection de paramètres. Pour afficher l'éditeur de collection de paramètres, cliquez sur le bouton points de suspension de la propriété *Params* ou *Parameters* dans l'inspecteur d'objets. Si l'instruction SQL ne contient aucun paramètre, aucun objet n'apparaît dans l'éditeur de collection.

Remarque L'éditeur de collection de paramètres est le même éditeur de collection que celui qui apparaît pour d'autres propriétés de collection. Comme cet éditeur est partagé par d'autre propriétés, son menu contextuel (clic droit) contient les commandes Ajouter et Supprimer. Cependant, elles ne sont jamais activées pour les paramètres des requêtes. Le seul endroit où l'on peut ajouter ou supprimer des paramètres est dans l'instruction SQL elle-même.

Sélectionnez chaque paramètre dans l'éditeur de collection de paramètres. Puis, utilisez l'inspecteur d'objets pour modifier ses propriétés.

Quand vous utilisez la propriété *Params* (objets *TParam*), vous pouvez inspecter ou modifier ce qui suit :

- La propriété *DataType* indique le type de données de la valeur du paramètre. Pour certains ensembles de données, cette valeur peut être correctement initialisée. Si l'ensemble de données ne peut pas déduire le type, *DataType* vaut *ftUnknown*, et vous devez la modifier et indiquer le type de la valeur du paramètre.

  La propriété *DataType* indique le type de données logique du paramètre. En général, ces types de données sont conformes aux types de données du serveur. Pour avoir la correspondance entre les types logiques et les types de données du serveur, consultez la documentation du mécanisme d'accès aux données ((BDE, dbExpress, InterBase).

- La propriété *ParamType* indique le type du paramètre sélectionné. Pour les requêtes, elle est toujours initialisée à *ptInput*, car les requêtes ne peuvent contenir que des paramètres d'entrée. Si la valeur de *ParamType* est *ptUnknown*, changez-la en *ptInput*.

- La propriété *Value* spécifie la valeur du paramètre sélectionné. Vous pouvez laisser vide cette *Value* si votre application fournit les valeurs des paramètres au cours de l'exécution.

Quand vous utilisez la propriété *Parameters* (objets *TParameter*), vous pouvez inspecter ou modifier ce qui suit :

- La propriété *DataType* indique le type de données de la valeur du paramètre. Pour certains types de données, vous pouvez ajouter d'autres informations :
  - La propriété *NumericScale* indique le nombre de décimales des paramètres numériques.
  - La propriété *Precision* indique le nombre total de chiffres des paramètres numériques.
  - La propriété *Size* indique le nombre de caractères des paramètres chaîne.
- La propriété *Direction* indique le type du paramètre sélectionné. Pour les requêtes, elle est toujours initialisée à *pdInput*, car les requêtes ne peuvent contenir que des paramètres d'entrée.
- La propriété *Attributes* indique le type des valeurs que le paramètre acceptera. *Attributes* peut être défini par une combinaison de *psSigned*, *psNullable* et *psLong*.
- La propriété *Value* spécifie la valeur du paramètre sélectionné. Vous pouvez laisser vide cette *Value* si votre application fournit les valeurs des paramètres au cours de l'exécution.

Fourniture des paramètres pendant l'exécution

Pour créer des paramètres à l'exécution, vous pouvez utiliser

- la méthode *ParamByName* pour affecter des valeurs à un paramètre en se basant sur son nom (non disponible pour *TADOQuery*)
- la propriété *Params* ou *Parameters* pour affecter des valeurs à un paramètre en se basant sur sa position dans l'instruction SQL.
- la propriété *Params.ParamValues ou Parameters.ParamValues* pour affecter des valeurs à un ou à plusieurs paramètres d'une seule ligne de commande, en se basant sur le nom de chaque ensemble de paramètres.

Le code suivant utilise *ParamByName* pour affecter le texte d'une boîte texte au paramètre :Capital :

```
SQLQuery1.ParamByName('Capital').AsString := Edit1.Text;
```

Le même code peut être réécrit en utilisant la propriété *Params* et l'indice 0 (en supposant que le paramètre :Capital est le premier de l'instruction SQL) :

```
SQLQuery1.Params[0].AsString := Edit1.Text;
```

La ligne de commande ci-dessous définit trois paramètres à la fois, à l'aide de la propriété *Params.ParamValues* :

```
Query1.Params.ParamValues['Name;Capital;Continent'] :=
  VarArrayOf([Edit1.Text, Edit2.Text, Edit3.Text]);
```

Remarquez que *ParamValues* utilise des variants, ce qui évite de transtyper des valeurs.

Etablissement de relations maître/détail en utilisant des paramètres

Pour définir une relation maître/détail où l'ensemble détail est un ensemble de données de type requête, vous devez spécifier une requête utilisant des paramètres. Ces paramètres font référence aux valeurs de champs en cours dans l'ensemble de données maître. Comme les valeurs de champs en cours dans l'ensemble de données maître changent dynamiquement lors de l'exécution, vous devez re-lier les paramètres de l'ensemble détail chaque fois que change l'enregistrement maître. Bien qu'il soit possible d'écrire pour cela du code utilisant un gestionnaire d'événement, tous les ensembles de données de type requête à l'exception de *TIBQuery* fournissent un mécanisme plus facile avec la propriété *DataSource*.

Si les valeurs des paramètres d'une requête paramétrée ne sont pas liées au moment de la conception ni spécifiées au moment de l'exécution, les ensembles de type requête tentent de fournir ces valeurs en utilisant la propriété *DataSource*. *DataSource* identifie un autre ensemble de données où sont recherchés des noms de champs correspondant aux noms des paramètres non liés. Cet ensemble de données de recherche peut être de n'importe quel type. L'ensemble de données où s'effectue la recherche doit être créé et rempli avant la création de l'ensemble de données détail qui l'utilise. S'il existe des correspondances dans l'ensemble de données où s'effectue la recherche, l'ensemble de données détail lie les valeurs des paramètres aux valeurs des champs trouvés dans l'enregistrement en cours pointé par la source de données.

Pour comprendre comment cela fonctionne, considérez deux tables : une table de clients (Customer) et une table de commandes (Orders). Pour chaque client, la table des commandes contient l'ensemble des commandes effectuées par le client. La table des clients comprend un champ ID qui spécifie un ID de client unique. La table des commandes comprend un champ CustID qui spécifie l'ID du client ayant passé une commande.

La première étape consiste à définir l'ensemble de données Customer :

1. Ajoutez un ensemble de données de type requête à votre application et liez-le à la table Customer.
2. Ajoutez un composant *TDataSource* nommé *CustomerSource*. Définissez sa propriété *DataSet* par l'ensemble de données ajouté à l'étape 1. Cette source de données représente désormais l'ensemble de données Customer.
3. Ajoutez un ensemble de données de type requête et définissez sa propriété *SQL* par

   ```
   SELECT CustID, OrderNo, SaleDate
   FROM Orders
   WHERE CustID = :ID
   ```

   Remarquez que le nom du paramètre est le même que le nom du champ dans la table maître (Customer).
4. Définissez la propriété *DataSource* de l'ensemble de données détail par *CustomerSource*. Définir cette propriété transforme l'ensemble détail en requête liée.

A l'exécution, le paramètre *:ID* de l'instruction SQL pour l'ensemble de données détail, n'a pas de valeur, l'ensemble de données essaie alors de trouver une correspondance entre le nom du paramètre et le nom d'une colonne de l'ensemble de données identifié par *CustomerSource*. *CustomerSource* obtient ses données de l'ensemble de données maître qui, à son tour, dérive ses données de la table Customer. Comme la table Customer contient un colonne appelée "ID", la valeur du champ *ID* dans l'enregistrement en cours de l'ensemble de données maître est assignée au paramètre *:ID* de l'instruction SQL de l'ensemble de données détail. Les ensembles de données sont liés dans une relation maître/détail. Chaque fois que l'enregistrement en cours change dans l'ensemble de données Customer, l'instruction SELECT de l'ensemble de données détail s'exécute pour retrouver toutes les commandes s'appliquant à l'identificateur de client en cours.

Préparation des requêtes

La préparation d'une requête est une étape facultative qui précède l'exécution de la requête. La préparation d'une requête consiste à soumettre l'instruction SQL, et ses éventuels paramètres, à la couche accès aux données et au serveur de base de données, à des fins d'analyse, d'allocation des ressources et d'optimisation. Certains ensembles de données effectuent des opérations supplémentaires lors de la préparation d'une requête. Ces opérations améliorent les performances de la requête, ce qui accélère votre application, spécialement lorsque vous travaillez avec des requêtes modifiables.

Une application peut préparer une requête en définissant la propriété *Prepared* par *True*. Si vous ne préparez pas une requête avant de l'exécuter, l'ensemble de données le fait pour vous automatiquement chaque fois que vous appelez *Open* ou *ExecSQL*. Bien que l'ensemble de données prépare la requête pour vous, vous pouvez améliorer les performances en préparant l'ensemble de données de manière explicite avant de l'ouvrir pour la première fois.

```
CustQuery.Prepared := True;
```

Lorsque vous préparez l'ensemble de données de manière explicite, les ressources allouées à l'exécution de l'instruction ne sont pas libérées tant que vous ne définissiez pas *Prepared* par *False*.

Définissez la propriété *Prepared* par *False* si vous voulez que l'ensemble de données soit re-préparé avant son exécution (par exemple, si vous ajoutez un paramètre).

Remarque Quand vous changez le texte de la propriété *SQL* d'une requête, l'ensemble de données ferme automatiquement la requête et annule sa préparation.

Exécution de requêtes qui ne renvoient pas d'ensemble de résultats

Quand une requête renvoie un ensemble d'enregistrements (comme une requête SELECT), vous exécutez la requête de la même façon que vous remplissez d'enregistrements un ensemble de données : en définissant *Active* par *True* ou en appelant la méthode *Open*.

Mais, de nombreuses commandes SQL ne renvoient pas d'enregistrement. De telles commandes comprennent les instructions DDL (Data Definition Language) ou DML (Data Manipulation Language) autres que les instructions SELECT (par exemple, les commandes INSERT, DELETE, UPDATE, CREATE INDEX et ALTER TABLE ne renvoient pas d'enregistrement).

Pour tous les ensembles de données de type requête, vous pouvez exécuter une requête qui ne renvoie pas d'ensemble de résultats en appelant *ExecSQL* :

```
CustomerQuery.ExecSQL;  { la requête ne renvoie pas d'ensemble de résultats }
```

Conseil Si vous exécutez la requête plusieurs fois, c'est une bonne idée de définir la propriété *Prepared*par *True*.

Même si la requête ne renvoie aucun enregistrement, vous pouvez vouloir connaître le nombre d'enregistrements qu'elle affecte (par exemple, le nombre d'enregistrements supprimés par une requête DELETE). La propriété *RowsAffected* donne le nombre d'enregistrements affectés après l'appel de la méthode *ExecSQL*.

Conseil Quand vous ne savez pas au moment de la conception si la requête renvoie un ensemble de résultats (par exemple, si l'utilisateur fournit la requête à l'exécution de façon dynamique), vous pouvez programmer les deux types d'instructions d'exécution de la requête dans un bloc **try...except**. Placez un appel à la méthode *Open* dans la clause **try**. Une requête d'action est exécutée quand la requête est activée avec la méthode *Open*, mais, en plus, une exception est déclenchée. Vérifiez l'exception et supprimez-la si elle indique simplement l'absence d'ensemble de résultats. (Par exemple, *TQuery* l'indique par une exception *ENoResultSet*.)

Utilisation d'ensembles de résultats unidirectionnels

Quand un ensemble de données de type requête renvoie un ensemble de résultats, il reçoit également un curseur, ou pointeur, sur le premier enregistrement de cet ensemble de résultats. L'enregistrement pointé par le curseur est l'enregistrement actif en cours. L'enregistrement en cours est celui dont les valeurs des champs sont affichées dans les composants orientés données associés à la source de données de l'ensemble de résultats. Sauf si vous utilisez dbExpress, ce curseur est par défaut bidirectionnel. Un curseur bidirectionnel peut naviguer dans les enregistrements à la fois en avant et en arrière. Le curseur bidirectionnel requiert une charge de traitement supplémentaire et peut ralentir certaines requêtes.

Si vous n'avez pas besoin de naviguer vers l'arrière dans un ensemble de résultats, *TQuery* et *TIBQuery* vous permettent d'améliorer les performances de la requête en demandant à la place un curseur unidirectionnel. Pour demander un curseur unidirectionnel, définissez la propriété *UniDirectional* par *True*.

Définissez *UniDirectional* avant la préparation et l'exécution de la requête. Le code suivant illustre la définition de la propriété *UniDirectional* avant la préparation et l'exécution d'une requête :

```
if not (CustomerQuery.Prepared) then
begin
  CustomerQuery.UniDirectional := True;
  CustomerQuery.Prepared := True;
end;
CustomerQuery.Open;  { renvoie un ensemble de résultats avec un curseur unidirectionnel }
```

Remarque Ne confondez pas la propriété *UniDirectional* et un ensemble de données unidirectionnel. Les ensembles de données unidirectionnels (*TSQLDataSet*, *TSQLTable*, *TSQLQuery* et *TSQLStoredProc*) utilisent dbExpress, qui ne renvoie que des curseurs unidirectionnels. En plus de supprimer la navigation arrière, les ensembles de données unidirectionnels ne mettent pas les enregistrements en tampon, et ont donc d'autres limites (comme l'incapacité à utiliser des filtres).

Utilisation d'ensembles de données de type procédure stockée

La façon dont votre application utilise une procédure stockée dépend de la façon dont a été programmée la procédure stockée, du fait qu'elle renvoie des données ou non, du serveur de base de données utilisé ou d'une combinaison de ces différents facteurs.

De façon générale, pour accéder à une procédure stockée sur un serveur, vous devez faire ceci :

1. Placez le composant approprié dans un module de données ou sur une fiche, et attribuez à sa propriété *Name* une valeur unique appropriée à votre application.
2. Identifiez le serveur de base de données qui définit la procédure stockée. Chaque ensemble de données de type procédure stockée fait cela différemment, mais, en général, il faut spécifier un composant de base de données :
   - Pour *TStoredProc*, spécifiez un composant *TDatabase* ou un alias BDE en utilisant la propriété *DatabaseName*.
   - Pour *TADOStoredProc*, spécifiez un composant *TADOConnection* en utilisant la propriété *Connection*.
   - Pour *TSQLStoredProc*, spécifiez un composant *TSQLConnection* en utilisant la propriété *SQLConnection*.
   - Pour *TIBStoredProc*, spécifiez un composant *TIBConnection* en utilisant la propriété *Database*.

Pour plus d'informations sur l'utilisation des composants connexion aux bases de données, voir chapitre 17, "Connexion aux bases de données".

3 Spécifiez la procédure stockée à exécuter. Pour la plupart des ensembles de données de type procédure stockée, vous le faites en définissant la propriété *StoredProcName*. La seule exception est *TADOStoredProc*, qui dispose à la place d'une propriété *ProcedureName*.

4 Si la procédure stockée renvoie un curseur à utiliser avec des contrôles de données visuels, ajoutez un composant source de données au module de données et définissez sa propriété *DataSet* par l'ensemble de données de type procédure stockée. Connectez les composants orientés données à la source de données à l'aide de leurs propriétés *DataSource* et *DataField*.

5 Fournissez les valeurs des éventuels paramètres de la procédure stockée. Si le serveur ne fournit pas d'information sur tous les paramètres des procédures stockées, vous devez fournir des informations supplémentaires sur les paramètres d'entrée, comme les noms et les types de données de ces paramètres. Pour plus d'informations sur l'utilisation des paramètres des procédures stockées, voir "Utilisation de paramètres avec les procédures stockées" à la page 18-59.

6 Exécutez la procédure stockée. Pour les procédures stockées qui renvoient un curseur, utilisez la propriété *Active* ou la méthode *Open*. Pour exécuter des procédures stockées qui ne renvoient pas de résultat ou renvoient uniquement des paramètres de sortie, utilisez la méthode *ExecProc* à l'exécution. Si vous prévoyez d'exécuter la procédure stockée plusieurs fois, vous pouvez appeler *Prepare* pour initialiser la couche d'accès aux données et les valeurs des paramètres de liaison dans la procédure stockée. Pour plus d'informations sur la préparation d'une procédure stockée, voir "Exécution de procédures stockées qui ne renvoient pas d'ensemble de résultats" à la page 18-63.

7 Traitez les éventuels résultats. Ces résultats peuvent être renvoyés sous forme de paramètres de résultat et de sortie, ou bien sous forme d'un ensemble de résultats qui remplit l'ensemble de données de type procédure stockée. Certaines procédures stockées renvoient plusieurs curseurs. Pour plus de détails sur la façon d'accéder aux curseurs supplémentaires, voir "Lecture de plusieurs ensembles de résultats" à la page 18-64.

Utilisation de paramètres avec les procédures stockées

Il existe quatre types de paramètres pouvant être associés aux procédures stockées :

- *Paramètres d'entrée*, utilisés pour transmettre des valeurs à une procédure stockée pour leur traitement.
- *Paramètres de sortie*, utilisés par une procédure stockée pour transmettre en retour des valeurs à une application.

- *Paramètres d'entrée/sortie*, utilisés pour transmettre des valeurs à une procédure stockée pour leur traitement, et utilisés par la procédure stockée pour transmettre en retour des valeurs à une application.
- Un *paramètre de résultat*, utilisé par certaines procédures stockées pour renvoyer à l'application une erreur ou une valeur d'état. Une procédure stockée ne peut renvoyer qu'un seul paramètre de résultat.

Le type de paramètres utilisé par une procédure stockée dépend de l'implémentation générale propre au langage des procédures stockées sur votre serveur de base de données et de l'instance spécifique de la procédure stockée. Quel que soit le serveur, certaines procédures stockées peuvent utiliser ou ne pas utiliser les paramètres d'entrée. Au contraire, certaines utilisations des paramètres sont spécifiques au serveur. Par exemple, sur MS-SQL Server et Sybase, les procédures stockées renvoient toujours un paramètre de résultat, mais l'implémentation InterBase d'une procédure stockée ne renvoie jamais de paramètre de résultat.

L'accès aux paramètres des procédures stockées est fourni par la propriété *Params* (dans *TStoredProc*, *TSQLStoredProc*, *TIBStoredProc*) ou par la propriété *Parameters* (dans *TADOStoredProc*). Quand vous affectez une valeur à la propriété *StoredProcName* (ou *ProcedureName*), l'ensemble de données génère automatiquement un objet pour chaque paramètre de la procédure stockée. Pour certains ensembles de données, si le nom de la procédure stockée n'est pas spécifié jusqu'au moment de l'exécution, les objets correspondants à chaque paramètre doivent être créés par programme à ce moment-là. Ne pas spécifier la procédure stockée et créer manuellement les objets *TParam* ou *TParameter* permet à un ensemble de données seul d'être utilisé avec un nombre quelconque de procédures stockées.

Remarque Certaines procédures stockées renvoient un ensemble de données en plus des paramètres de sortie et de résultat. Les applications peuvent afficher les enregistrements des ensembles de données dans des contrôles orientés données, mais doivent traiter séparément les paramètres de sortie et les paramètres de résultat.

Définition des paramètres pendant la conception

Vous pouvez attribuer des valeurs aux paramètres des procédures stockées au moment de la conception en utilisant l'éditeur de collection de paramètres. Pour afficher l'éditeur de collection de paramètres, cliquez sur le bouton points de suspension de la propriété *Params* ou *Parameters* dans l'inspecteur d'objets.

Important Vous pouvez donner des valeurs aux paramètres d'entrée en les sélectionnant dans l'éditeur de collection de paramètres et en utilisant l'inspecteur d'objets pour définir la propriété *Value*. Mais, vous ne devez modifier ni les noms ni les types de données des paramètres d'entrée indiqués par le serveur. Si vous le faites, une exception se produit lorsque vous exécutez la procédure stockée.

Certains serveurs n'indiquent pas les noms ni les types de données des paramètres. En ce cas, vous devez définir les paramètres manuellement en utilisant l'éditeur de collection de paramètres. Cliquez avec le bouton droit et choisissez Ajouter pour ajouter des paramètres. Vous devez décrire entièrement

chaque paramètre que vous ajoutez. Même si vous n'avez pas besoin d'ajouter des paramètres, vous devez vérifier que les objets paramètre individuels sont corrects.

Si l'ensemble de données a une propriété *Params* (objets *TParam*), les propriétés suivantes doivent être correctement spécifiées :

- La propriété *Name* indique le type du paramètre sélectionné.
- La propriété *DataType* indique le type de données de la valeur du paramètre. Lorsque vous utilisez *TSQLStoredProc*, certains types de données exigent des informations supplémentaires :
  - La propriété *NumericScale* indique le nombre de décimales des paramètres numériques.
  - La propriété *Precision* indique le nombre total de chiffres des paramètres numériques.
  - La propriété *Size* indique le nombre de caractères des paramètres chaîne.
- La propriété *ParamType* indique le type du paramètre sélectionné. Ce peut être *ptInput* (pour les paramètres d'entrée), *ptOutput* (pour les paramètres de sortie), *ptInputOutput* (pour les paramètres d'entrée/sortie) ou *ptResult* (pour les paramètres de résultat).
- La propriété *Value* spécifie la valeur du paramètre sélectionné. Vous ne pouvez pas définir la valeur des paramètres de sortie ou des paramètres de résultat. C'est l'exécution de la procédure stockée qui définit ces types de paramètres. Pour les paramètres d'entrée ou d'entrée/sortie, vous pouvez laisser *Value* vide si votre application fournit les valeurs des paramètres au cours de l'exécution.

Si l'ensemble de données utilise une propriété *Parameters* (objets *TParameter*), les propriétés suivantes doivent être correctement spécifiées :

- La propriété *Name* indique le nom du paramètre tel qu'il est défini par la procédure stockée.
- La propriété *DataType* indique le type de données de la valeur du paramètre. Pour certains types de données, vous pouvez ajouter d'autres informations :
  - La propriété *NumericScale* indique le nombre de décimales des paramètres numériques.
  - La propriété *Precision* indique le nombre total de chiffres des paramètres numériques.
  - La propriété *Size* indique le nombre de caractères des paramètres chaîne.
- La propriété *Direction* indique le type du paramètre sélectionné. Ce peut être *pdInput* (pour les paramètres d'entrée), *pdOutput* (pour les paramètres de sortie), *pdInputOutput* (pour les paramètres d'entrée/sortie) ou *pdReturnValue* (pour les paramètres de résultat).

- La propriété *Attributes* indique le type des valeurs que le paramètre acceptera. *Attributes* peut être défini par une combinaison de *psSigned*, *psNullable* et *psLong*.
- La propriété *Value* spécifie la valeur du paramètre sélectionné. Ne définissez pas la valeur des paramètres de sortie ou des paramètres de résultat. Pour les paramètres d'entrée ou d'entrée/sortie, vous pouvez laisser vide cette *Value* si votre application fournit les valeurs des paramètres au cours de l'exécution.

Utilisation des paramètres pendant l'exécution

Pour certains ensembles de données, si le nom de la procédure stockée n'est pas spécifié jusqu'au moment de l'exécution, aucun objet *TParam* ne sera créé automatiquement et ils devront être créés par programme. Cela se fera en utilisant la méthode *TParam.Create* ou la méthode *TParams.AddParam* :

```
var
  P1, P2: TParam;
begin
  ...
  with StoredProc1 do begin
    StoredProcName := 'GET_EMP_PROJ';
    Params.Clear;
    P1 := TParam.Create(Params, ptInput);
    P2 := TParam.Create(Params, ptOutput);
    try
      Params[0].Name := 'EMP_NO';
      Params[1].Name := 'PROJ_ID';
      ParamByname('EMP_NO').AsSmallInt := 52;
      ExecProc;
      Edit1.Text := ParamByname('PROJ_ID').AsString;
    finally
      P1.Free;
      P2.Free;
    end;
  end;
  ...
end;
```

Même si vous n'avez pas besoin d'ajouter les objets paramètre individuels à l'exécution, vous pouvez accéder à chacun d'eux en affectant des valeurs aux paramètres d'entrée et en récupérant les valeurs des paramètres de sortie. Vous pouvez utiliser la méthode *ParamByName* de l'ensemble de données, pour accéder aux paramètres individuels par leur nom. Par exemple, le code suivant définit la valeur d'un paramètre d'entrée/sortie, exécute la procédure stockée et récupère la valeur renvoyée :

```
with SQLStoredProc1 do
begin
  ParamByName('IN_OUTVAR').AsInteger := 103;
  ExecProc;
  IntegerVar := ParamByName('IN_OUTVAR').AsInteger;
end;
```

Préparation des procédures stockées

Comme avec les ensembles de données de type requête, les ensembles de données de type procédure stockée doivent être préparés avant d'exécuter la procédure stockée. La préparation de la procédure stockée indique à la couche d'accès aux données et au serveur de base de données d'allouer des ressources pour la procédure stockée et de lier des paramètres. Ces opérations peuvent améliorer les performances.

Si vous tentez d'exécuter une procédure stockée avant de la préparer, l'ensemble de données la prépare pour vous automatiquement et annule la préparation après son exécution. Si vous prévoyez d'exécuter une procédure stockée de nombreuses fois, il est plus efficace de la préparer explicitement en définissant la propriété Prepared par *True*.

```
MyProc.Prepared := True;
```

Lorsque vous préparez l'ensemble de données de manière explicite, les ressources allouées à l'exécution de la procédure stockée ne sont pas libérées tant que vous ne définissez pas *Prepared* par *False*.

Définissez la propriété *Prepared* par *False* si vous voulez que l'ensemble de données soit re-préparé avant son exécution (par exemple, si vous modifiez les paramètres quand vous utilisez les procédures Oracle surchargées).

Exécution de procédures stockées qui ne renvoient pas d'ensemble de résultats

Quand une procédure stockée renvoie un curseur, vous l'exécutez de la même façon que vous remplissez d'enregistrements un ensemble de données : en définissant *Active* par *True* ou en appelant la méthode *Open*.

Mais, de nombreuses procédures stockées ne renvoient pas de données ou renvoient uniquement des paramètres de résultat ou de sortie. L'exécution d'une procédure stockée ne renvoyant pas d'ensemble de résultats est obtenue en appelant la méthode *ExecProc*. Après l'exécution de la procédure stockée, vous pouvez utiliser la méthode *ParamByName* pour lire la valeur du paramètre de résultat ou de tout paramètre de sortie :

```
MyStoredProcedure.ExecProc;  { ne renvoie pas d'ensemble de résultats }
Edit1.Text := MyStoredProcedure.ParamByName('OUTVAR').AsString;
```

Remarque *TADOStoredProc* n'a pas de méthode *ParamByName*. Pour obtenir les valeurs des paramètres de sortie lorsque vous utilisez ADO, accédez aux objets paramètre en utilisant la propriété *Parameters*.

Conseil Si vous exécutez la procédure stockée plusieurs fois, c'est une bonne idée de définir la propriété *Prepared* par *True*.

Lecture de plusieurs ensembles de résultats

Certaines procédures stockées renvoient plusieurs ensembles d'enregistrements. Lorsque vous l'ouvrez, l'ensemble de données ne récupère que le premier ensemble d'enregistrements. Si vous utilisez *TSQLStoredProc* ou *TADOStoredProc*, vous pouvez accéder aux autres ensembles d'enregistrements en appelant la méthode *NextRecordSet* :

```
var
  DataSet2: TCustomSQLDataSet;
begin
  DataSet2 := SQLStoredProc1.NextRecordSet;
  ...
```

Dans *TSQLStoredProc*, *NextRecordSet* renvoie le composant *TCustomSQLDataSet* qui vient d'être créé et fournit l'accès au prochain ensemble d'enregistrements. Dans *TADOStoredProc*, *NextRecordset* renvoie une interface qui peut être affectée à la propriété *RecordSet* d'un ensemble de données ADO existant. Pour n'importe quelle classe, la méthode renvoie le nombre d'enregistrements de l'ensemble de données renvoyé sous forme d'un paramètre de sortie.

La première fois que vous appelez *NextRecordSet*, elle renvoie un ensemble de données pour le second ensemble d'enregistrements. Appeler à nouveau *NextRecordSet* renvoie un troisième ensemble de données, et ainsi de suite jusqu'à ce qu'il n'y ait plus d'ensemble d'enregistrements. Lorsque qu'il n'y a pas de curseur supplémentaire, *NextRecordSet* renvoie **nil**.

Chapitre

19

Manipulation des composants champ

Ce chapitre décrit les propriétés, les événements et les méthodes communes à l'objet *TField* et ses descendants. Les composants champ représentent les différents champs (colonnes) des ensembles de données. Ce chapitre décrit également comment utiliser les composants champ pour contrôler l'affichage et l'édition des données dans des applications.

Les composants champ sont toujours associés à un ensemble de données. Vous n'utilisez jamais un objet *TField* directement dans vos applications. Au lieu de cela, chaque composant champ de votre application est un descendant de *TField* spécifique au type de données d'une colonne d'un ensemble de données. Les composants champ fournissent des contrôles orientés données tels que *TDBEdit* et *TDBGrid* afin d'accéder aux données d'une colonne particulière de l'ensemble de données associé.

D'une façon générale, un composant champ représente les caractéristiques d'une colonne, ou d'un champ, d'un ensemble de données (comme le type de données ou la taille du champ). Il représente également les caractéristiques d'affichage du champ, comme l'alignement, le format d'affichage et le format d'édition. Par exemple, un composant *TFloatField* est doté de quatre propriétés qui affectent directement l'apparence des données :

Tableau 19.1 Propriétés du composant TFloatField affectant l'affichage des données

Propriété	Utilisation
Alignment	Centre ou aligne à droite ou à gauche les valeurs d'un contrôle pour l'affichage.
DisplayWidth	Spécifie le nombre de chiffres affichés dans un contrôle.
DisplayFormat	Spécifie le formatage des données affichées (par exemple, le nombre de décimales).
EditFormat	Contrôle l'aspect des valeurs dans un contrôle au cours d'édition.

Lorsque vous passez d'un enregistrement à un autre dans un ensemble de données, un composant champ vous permet de visualiser et de modifier la valeur du champ dans l'enregistrement en cours.

Les composants champ ont de nombreuses propriétés en commun (comme *DisplayWidth* et *Alignment*) ; ils ont également des propriétés spécifiques à certains types de données (telles que *Precision* pour *TFloatField*). Chacune de ces propriétés affecte l'aspect des données dans une fiche. Certaines d'entre elles, comme *Precision*, peuvent également affecter les données pouvant être saisies par un utilisateur quand il modifie ou saisit des données.

Tous les composants champ d'un ensemble de données sont soit *dynamiques* (automatiquement générés à partir de la structure sous-jacente des tables de bases de données) ou *persistants* (générés à partir des noms de champs et des propriétés définis dans l'éditeur de champs). Les champs dynamiques et les champs persistants ont des caractéristiques différentes et sont appropriés à des situations précises. Les sections suivantes décrivent en détail les champs persistants et les champs dynamiques et indiquent quand les utiliser.

Composants champ dynamique

Les composants champ générés dynamiquement correspondent au comportement par défaut. En fait, tous les composants champ d'un ensemble de données sont créés dynamiquement à chaque fois que vous placez un ensemble de données dans un module de données puis spécifiez comment l'ensemble de données récupère ses données et l'ouvrez. Un composant champ est *dynamique* s'il est créé automatiquement en fonction de la structure physique sous-jacente des données représentées par un ensemble de données. Les ensembles de données génèrent un composant champ pour chaque colonne de données sous-jacentes. Le descendant *TField* créé pour chaque colonne est déterminé par les informations de type de champ envoyées par la base de données ou par le composant fournisseur (pour *TClientDataSet*).

Les champs dynamiques sont temporaires. Leur durée de vie équivaut à la durée d'ouverture d'un ensemble de données. A chaque nouvelle ouverture d'un ensemble de données utilisant des champs dynamiques, il reconstruit un ensemble de composants champ dynamiques à partir de la structure des données sous-jacentes. Si les colonnes des données sous-jacentes changent, à la prochaine ouverture d'un ensemble de données utilisant des composants champ dynamiques, les composants champ générés automatiquement changeront également.

Les champs dynamiques doivent être utilisés dans des applications flexibles en matière d'édition et d'affichage des données. Par exemple, pour créer un outil de visualisation de bases de données comme l'explorateur SQL, vous devez utiliser des champs dynamiques, car chaque table de base de données n'a pas le même nombre de colonnes et ses colonnes sont de types différents. Les champs dynamiques s'utilisent aussi dans des applications où l'interaction utilisateur avec les données se produit à l'intérieur des composants grille et lorsque les ensembles de données utilisés par l'application changent souvent.

Pour utiliser des champs dynamiques dans une application :

1. Placez des ensembles de données et des sources de données dans un module de données.
2. Associez les ensembles de données aux données. Cela suppose l'utilisation d'un fournisseur ou composant connexion pour connecter la source des données et définir les propriétés qui déterminent les données représentées par l'ensemble de données.
3. Associez les sources de données aux ensembles de données.
4. Placez des contrôles orientés données dans les fiches de votre application, ajoutez le module de données dans chaque clause uses des unités de fiches, et associez chaque contrôle orienté données à une source de données du module. Il faut aussi associer un champ à chaque contrôle orienté données le nécessitant. Etant donné que vous utilisez des composants champ dynamiques, l'existence des noms des champs à l'ouverture de l'ensemble de données n'est pas garantie.
5. Ouvrez les ensembles de données.

Les champs dynamiques présentent malgré tout quelques limites. Sans écrire de code, il est impossible de changer leurs paramètres par défaut d'affichage et d'édition, il est difficile de modifier leur ordre d'affichage sans prendre de risque et il est impossible d'empêcher l'accès à des champs de l'ensemble de données. Des champs supplémentaires comme les champs de référence ou les champs calculés ne peuvent pas être créés pour l'ensemble de données et il est impossible de surcharger le type par défaut d'un champ dynamique. Pour un plus grand contrôle des champs de vos applications de bases de données, vous devez appeler l'éditeur de champs et créer des champs persistants.

Champs persistants

Par défaut, les champs des ensembles de données sont des champs dynamiques. Leurs propriétés et leur disponibilité sont automatiquement définies et ne peuvent pas être modifiées. Pour contrôler les propriétés et les événements d'un champ, vous devez créer des champs persistants.

Grâce aux champs persistants, vous pouvez :

- définir ou modifier les caractéristiques d'affichage ou d'édition du champ à la conception ou à l'exécution ;
- créer de nouveaux champs, tels que des champs de référence, des champs calculés et des champs agrégés, dont les valeurs sont basées sur les champs d'un ensemble de données ;
- valider les entrées de données ;
- retirer des composants champ de la liste des composants persistants pour empêcher votre application d'accéder à des colonnes particulières d'une base de données sous-jacente ;

- définir de nouveaux champs pour remplacer des champs existants, d'après les colonnes de la table ou de la requête sous-jacente d'un ensemble de données.

Au moment de la conception vous pouvez, ce qui est conseillé, utiliser l'éditeur de champs pour créer des listes persistantes de composants champ utilisés par les ensembles de données de votre application. Ces listes sont stockées dans votre application et ne changent pas, même si la structure de la base de données sous-jacente d'un ensemble de données est modifiée. Il est ensuite possible de créer des gestionnaires d'événements pour ces champs de façon à ce qu'ils puissent répondre aux modifications de données et aux validations de saisies.

Remarque Lorsque vous créez les champs persistants d'un ensemble de données, seuls ceux sélectionnés sont disponibles pour votre application en mode conception et à l'exécution. Pendant la phase de conception, vous pouvez toujours utiliser l'éditeur de champs de façon à ajouter ou supprimer des champs persistants pour un ensemble de données.

La totalité des champs utilisés dans un ensemble de données sont soit persistants soit dynamiques. Il est impossible de combiner les deux types de champs au sein du même ensemble de données. Si vous avez créé des champs persistants pour un ensemble de données, vous devrez tous les supprimer pour revenir à des champs dynamiques. Pour plus d'informations sur les champs dynamiques, voir "Composants champ dynamique" à la page 19-2.

Remarque L'une des principales utilisations des champs persistants est de contrôler l'aspect et l'affichage des données. Vous pouvez également contrôler l'aspect des colonnes dans les grilles orientées données. Pour savoir comment gérer l'aspect des colonnes dans les grilles, reportez-vous à "Création d'une grille personnalisée" à la page 15-19.

Création de champs persistants

Les composants champ persistants créés avec l'éditeur de champs offrent un accès en programmation aux données sous-jacentes. Ils garantissent qu'à chaque exécution de votre application, celle-ci utilise et affiche toujours les mêmes colonnes, dans le même ordre, même si la structure physique de la base de données sous-jacente change. Les composants orientés données et le code du programme qui dépendent de champs spécifiques fonctionnent toujours comme prévu. Si une colonne dont dépend un composant champ persistant est supprimée ou modifiée, Delphi provoque une exception au lieu de faire fonctionner l'application avec une colonne inexistante ou avec des données sans correspondance.

Pour créer un composant champ persistant pour un ensemble de données :

1. Placez un ensemble de données dans un module de données.
2. Liez l'ensemble de données à ses données sous-jacentes. Pour ce faire, vous devez généralement associer l'ensemble de données à un fournisseur ou composant connexion et spécifier les propriétés permettant de décrire les données. Par exemple, si vous utilisez *TADODataSet*, vous pouvez attribuer à

la propriété *Connection* un composant *TADOConnection* correctement configuré et à la propriété *CommandText* une requête valide.

3 Double-cliquez sur le composant ensemble de données dans le module de données de façon à appeler l'éditeur de champs. Ce dernier contient une barre de titre, des boutons de navigation et une boîte liste.

   La barre de titre de l'éditeur de champs affiche le nom du module de données ou de la fiche contenant l'ensemble de données, ainsi que le nom de celui-ci. Ainsi, si vous ouvrez l'ensemble de données *Customers* dans le module de données *CustomerData*, la barre de titre affiche 'CustomerData.Customers', ou la partie du nom qu'elle peut contenir.

   Sous la barre de titre figure un ensemble de boutons de navigation vous permettant de faire défiler un à un les enregistrements d'un ensemble de données au moment de la conception ; ces boutons vous permettent également de passer directement au premier ou au dernier enregistrement. Les boutons de navigation sont estompés si vous n'avez pas encore créé de composant champ persistant pour l'ensemble de données ou si celui-ci n'est pas actif. Si l'ensemble de données est unidirectionnel, les boutons permettant de se déplacer sur les dernier et précédent enregistrements sont toujours estompés.

   La boîte liste affiche le nom des composants champ persistants de l'ensemble de données. Lorsque vous lancez l'éditeur de champs la première fois pour un nouvel ensemble de données, la liste est vide car les composants champ sont dynamiques et non persistants. Si vous lancez l'éditeur de champs pour un ensemble de données contenant déjà des composants champ persistants, vous verrez leur nom dans la boîte liste.

4 Choisissez l'option d'ajout de champs dans le menu contextuel de l'éditeur de champs.

5 Dans la boîte de dialogue d'ajout de champs, sélectionnez les champs persistants. Par défaut, tous les champs sont sélectionnés à l'ouverture de la boîte de dialogue. Tous les champs que vous sélectionnerez dans cette boîte de dialogue deviendront des champs persistants.

La boîte de dialogue d'ajout de champs se referme et les champs que vous avez sélectionnés apparaissent dans la boîte liste de l'éditeur de champs ; ce sont les champs persistants. Si l'ensemble de données est actif, vous remarquerez également que les boutons de navigation Dernier (si l'ensemble de données n'est pas unidirectionnel) et Suivant, au-dessus de la boîte liste sont activés.

Désormais, à chaque fois que vous ouvrirez l'ensemble de données, il ne créera plus de composant champ dynamique pour chaque colonne de la base de données sous-jacente. Au lieu de cela, il ne créera des composants persistants que pour les champs spécifiés.

A chaque fois que vous ouvrirez l'ensemble de données, il vérifiera que chacun des champs persistants non calculés est présent ou qu'il peut être créé à partir des données de la base. Sinon, il provoquera une exception vous avertissant que le champ n'est pas valide et ne s'ouvrira pas.

Modification de l'ordre des champs persistants

L'ordre dans lequel apparaissent les composants champ persistants dans la boîte liste de l'éditeur de champs est l'ordre par défaut dans lequel ils apparaissent dans un composant grille orienté données. Vous pouvez le modifier en faisant glisser les champs ailleurs dans la boîte liste.

Pour changer l'emplacement de champs :

1. Sélectionnez-les. Vous pouvez sélectionner et déplacer plusieurs champs à la fois.
2. Faites-les glisser au nouvel emplacement.

Si vous sélectionnez un ensemble de champs non contigus et les faites glisser vers un nouvel emplacement, ils sont insérés en tant que bloc contigu. A l'intérieur de ce bloc, l'ordre des champs reste inchangé.

Pour changer l'ordre d'un champ dans la liste, une autre méthode consiste à sélectionner le champ et utiliser les touches *Ctrl+Haut* et *Ctrl+Bas*.

Définition de nouveaux champs persistants

Non seulement vous pouvez sélectionner des composants champ existants à changer en composants champ persistants pour un ensemble de données, mais aussi créer des champs persistants spéciaux supplémentaires ou destinés à remplacer les autres champs persistants de l'ensemble de données.

Les champs persistants que vous créez ne servent que pour l'affichage. Les données qu'ils contiennent au moment de l'exécution ne sont pas conservées, soit parce qu'elles existent déjà ailleurs dans la base de données, soit parce qu'elles sont temporaires. La structure physique des données sous-jacentes de l'ensemble de données reste de toute façon inchangée.

Pour créer un nouveau composant champ persistant, cliquez avec le bouton droit de la souris sur la boîte liste de l'éditeur de champs et choisissez Nouveau champ. La boîte de dialogue Nouveau champ apparaît.

La boîte de dialogue Nouveau champ contient trois boîtes groupe : Propriétés du champ, Type de champ et Définition de la référence.

- La boîte groupe Propriétés du champ vous permet d'entrer des informations générales sur le composant champ. Tapez le nom de champ dans la zone de saisie Nom. Le nom que vous saisissez correspond à la propriété *FieldName* du composant champ. La boîte de dialogue Nouveau champ l'utilise pour construire un nom de composant dans la zone de saisie Composant. Le nom qui s'y affiche correspond à la propriété *Name* du composant champ ; il est fourni uniquement à titre d'information (*Name* est l'identificateur vous permettant de faire référence au composant champ dans le code source). La boîte de dialogue ne tient pas compte de ce que vous entrez directement dans la zone de saisie Composant.

- La boîte à options "Type" vous permet de spécifier le type de données du composant champ. Vous devez obligatoirement indiquer un type de données pour chaque nouveau composant champ créé. Par exemple, pour afficher une valeur monétaire à virgule flottante dans un champ, sélectionnez *Currency* dans la liste déroulante. Utilisez la zone de saisie Taille pour spécifier le nombre maximum de caractères pouvant être affichés ou entrés dans un champ chaîne, ou bien la taille des champs *Bytes* et *VarBytes*. Taille ne s'utilise pour aucun autre type de données.
- La boîte groupe Type de champ vous permet de spécifier le type du nouveau composant champ à créer. Son type par défaut est Données. Si vous choisissez Référence, les zones de saisie Ensemble de données et Champs clé de la boîte groupe Définition de la référence sont activées. Il est aussi possible de créer des champs calculés, et si vous travaillez avec un ensemble de données client, vous pouvez même créer des champs *CalcInterne* ou *Agrégat*. Le tableau suivant dresse la liste des champs pouvant être créés :

Tableau 19.2 Types de champs persistants

Type de champ	Utilisation
Champs de données	Ils remplacent généralement les champs existants (par exemple, pour changer le type des données d'un champ).
Champs calculés	Ils affichent des valeurs calculées lors de l'exécution par le gestionnaire d'événement *OnCalcFields* d'un ensemble de données.
CalcInterne	Ils affichent des valeurs calculées lors de l'exécution par le client ensemble de données et stockées avec ses données.
Champs de référence	Ils extraient des valeurs d'un ensemble de données spécifié lors de l'exécution en fonction des critères de recherche que vous indiquez (non pris en charge par les ensembles de données unidirectionnels).
Agrégat	Affiche une valeur récapitulant les données contenues dans un ensemble d'enregistrements d'un ensemble de données client.

La boîte groupe Définition de la référence ne s'utilise que pour créer des champs de référence. Elle est décrite dans "Définition d'un champ de référence" à la page 19-10.

Définition d'un champ de données

Un champ de données remplace un champ existant dans un ensemble de données. Il peut arriver que pour des raisons liées par exemple à la programmation vous ayez à remplacer *TSmallIntField* par *TIntegerField*. Comme vous ne pouvez pas changer directement son type de données, vous devez définir un nouveau champ pour le remplacer.

Important Même si pour remplacer un champ existant vous en définissez un nouveau, celui-ci doit dériver la valeur de ses données d'une colonne d'une table sous-jacente d'un ensemble de données.

Pour créer un champ de données de remplacement d'un champ de la table sous-jacente à un ensemble de données :

1 Retirez le champ de la liste des champs persistants affectée à l'ensemble de données puis choisissez Nouveau champ dans le menu contextuel.

2 Dans la boîte de dialogue Nouveau champ, entrez dans la zone de saisie Nom le nom d'un champ existant de la table de la base de données. N'entrez pas un nouveau nom de champ car vous spécifiez ici le nom d'un champ existant dont le nouveau champ va dériver ses données.

3 Choisissez un nouveau type de données dans la boîte à options Type. Celui-ci doit être différent de celui du champ que vous remplacez. Il n'est pas possible de remplacer un champ chaîne d'une taille donnée par un champ chaîne d'une autre taille. Même si le type de données doit être différent il doit néanmoins être compatible avec le type de données réel du champ de la table sous-jacente.

4 Le cas échéant, entrez la taille du champ dans la zone de saisie correspondante. Cela ne s'applique qu'aux champs de type *TStringField*, *TBytesField* et *TVarBytesField*.

5 Sélectionnez Données dans la boîte groupe Type de champ.

6 Choisissez OK. La boîte de dialogue Nouveau champ se referme, le nouveau champ remplace celui que vous avez spécifié à l'étape 1 et la déclaration de composant dans **la déclaration de type** du module de données ou de la fiche est mise à jour.

Pour éditer les propriétés ou les événements associés au composant champ, sélectionnez le nom du composant dans la boîte liste de l'éditeur de champs, puis éditez ses propriétés ou événements avec l'inspecteur d'objets. Pour plus de détails sur l'édition des propriétés et des événements d'un composant champ, reportez-vous à "Définition des événements et des propriétés des champs persistants" à la page 19-13.

Définition d'un champ calculé

Un champ calculé, comme son nom l'indique, affiche les valeurs calculées lors de l'exécution par le gestionnaire d'événement *OnCalcFields* d'un ensemble de données. Par exemple, vous pouvez être amené à créer un champ chaîne qui affiche des valeurs concaténées à partir d'autres champs.

Pour créer un champ calculé à partir de la boîte de dialogue Nouveau champ :

1 Saisissez un nom dans la zone de saisie Nom (attention à ne pas entrer le nom d'un champ existant).

2 Choisissez le type de données de ce champ dans la boîte à options Type.

3 Entrez la taille du champ dans la zone de saisie correspondante, le cas échéant. La taille ne s'applique qu'aux champs de type *TStringField*, *TBytesField* et *TVarBytesField*.

4 Sélectionnez Calculé ou CalcInterne dans la boîte groupe Type de champ. InternalCalc est uniquement disponible si vous utilisez un ensemble de données client. La principale différence entre les deux types de champs calculés est que les valeurs calculées pour un champ CalcInterne sont stockées et extraites des données de l'ensemble de données client.

5 Choisissez OK. Le nouveau champ calculé est ajouté automatiquement à la fin de la liste des champs persistants de la boîte liste de l'éditeur de champs et la déclaration du composant est ajoutée automatiquement à **la déclaration de type** du module de données ou de la fiche.

6 Introduisez le code qui calcule les valeurs du champ dans le gestionnaire d'événement *OnCalcFields* de l'ensemble de données. Pour savoir comment écrire du code pour calculer la valeur des champs, voir "Programmation d'un champ calculé" à la page 19-9.

Remarque Pour éditer les propriétés ou événements associés au composant champ, sélectionnez le nom du composant dans la boîte liste de l'éditeur de champs, puis procédez à l'édition avec l'inspecteur d'objets. Pour en savoir plus à ce sujet, reportez-vous à "Définition des événements et des propriétés des champs persistants" à la page 19-13.

Programmation d'un champ calculé

Après avoir défini un champ calculé, vous devez écrire le code permettant d'en calculer la valeur, sinon il aura toujours une valeur NULL. Le code d'un champ calculé doit être placé dans l'événement *OnCalcFields* de son ensemble de données.

Pour programmer la valeur d'un champ calculé :

1 Sélectionnez le composant ensemble de données dans la liste déroulante de l'inspecteur d'objets.

2 Choisissez la page Evénements dans l'inspecteur d'objets.

3 Double-cliquez sur la propriété *OnCalcFields* pour lancer ou créer une procédure *CalcFields* pour le composant ensemble de données.

4 Ecrivez le code définissant les valeurs et les autres propriétés du champ calculé.

Supposons par exemple que vous ayez créé un champ calculé *CityStateZip* pour la table *Customers* dans le module de données *CustomerData*. *CityStateZip* affichera le nom de la ville et de l'état où se trouve l'entreprise et le code postal sur une même ligne d'un contrôle orienté données.

Pour ajouter du code à la procédure *CalcFields* pour la table *Customers*, sélectionnez celle-ci dans la liste déroulante de l'inspecteur d'objets, passez à la page Evénements et double-cliquez sur la propriété *OnCalcFields*.

La procédure *TCustomerData.CustomersCalcFields* apparaît dans la fenêtre de code source de l'unité. Ajoutez le code suivant à la procédure pour calculer le champ :

```
CustomersCityStateZip.Value := CustomersCity.Value + ', ' + CustomersState.Value
  + ' ' + CustomersZip.Value;
```

Remarque Lorsque vous écrivez le gestionnaire d'événement *OnCalcFields* pour un champ calculé de façon interne, vous pouvez améliorer les performances en consultant la propriété *State* de l'ensemble de données client et en recalculant la valeur uniquement si *State* vaut *dsInternalCalc*. Pour plus d'informations, voir

"Utilisation de champs calculés de façon interne dans les ensembles de données client" à la page 23-13.

Définition d'un champ de référence

Un champ de référence est un champ en lecture seule qui affiche des valeurs à l'exécution en fonction du critère de recherche ayant été spécifié. Dans sa forme la plus simple, on transmet à un champ de référence le nom du champ à rechercher, la valeur à rechercher et le champ de l'ensemble de données de référence dont la valeur doit s'afficher.

Prenons par exemple une application de VPC permettant à un opérateur d'utiliser un champ de référence pour déterminer automatiquement la ville et l'état correspondant à un code postal fourni par le client. Dans ce cas, la colonne sur laquelle doit porter la recherche peut s'appeler *ZipTable.Zip*, la valeur à rechercher étant le code postal du client tel qu'il a été entré dans *Order.CustZip* et les valeurs à renvoyer étant celles des colonnes *ZipTable.City* et *ZipTable.State* de l'enregistrement où *ZipTable.Zip* correspond à la valeur actuelle du champ *Order.CustZip*.

Remarque Les ensembles de données unidirectionnels ne prennent pas en charge les champs de référence.

Pour créer un champ de référence dans la boîte de dialogue Nouveau champ :

1. Saisissez le nom du champ de référence dans la zone de saisie Nom. Attention à ne pas saisir un nom de champ existant.
2. Dans la boîte à options Type, choisissez un type de données pour le champ.
3. Le cas échéant, entrez la taille du champ dans la zone de saisie correspondante. La taille n'est à spécifier que pour les champs de type *TStringField*, *TBytesField* et *TVarBytesField*.
4. Sélectionnez Référence dans la boîte groupe Type de champ pour activer les boîtes à options Ensemble de données et Champs clé.
5. Choisissez dans la liste déroulante de la boîte à options "Ensemble de données" l'ensemble de données sur lequel doit porter la recherche des valeurs de champ. L'ensemble de référence doit être différent de celui du composant champ lui-même, sinon une exception de référence circulaire sera provoquée au moment de l'exécution. En spécifiant un ensemble de données de référence, vous activez les boîtes à options Clés de référence et Champ résultat.
6. Dans la liste déroulante Champs clé, choisissez dans l'ensemble de données actif le champ dont les valeurs doivent se correspondre. Pour faire correspondre plusieurs champs, entrez leur nom directement au lieu de les choisir dans la liste déroulante. Séparez leur nom par des points-virgules. Si vous utilisez plusieurs champs, il faut employer des composants champ persistants.
7. Dans la liste déroulante Clés de référence, choisissez un champ de l'ensemble de données de référence devant correspondre au champ source spécifié à l'étape 6. Si vous avez spécifié plusieurs champs clé, il faut spécifier le même

nombre de clés de référence. Pour spécifier plusieurs champs, entrez leur nom directement. Séparez leur nom par des points-virgules.

8 Dans la liste déroulante Champ résultat, choisissez un champ de l'ensemble de données de référence à renvoyer comme valeur du champ de référence que vous êtes en train de créer.

Lorsque vous concevez et exécutez votre application, les valeurs des champs de référence sont déterminées avant celles des champs calculés. Il est possible de créer des champs calculés basés sur des champs de référence mais il est impossible de créer des champs de référence basés sur des champs calculés.

Vous pouvez affiner à l'aide de la propriété *LookupCache* la façon dont les champs de référence sont déterminés. Cette propriété détermine si les valeurs d'un champ de référence sont placées en mémoire cache lors de la première ouverture d'un ensemble de données ou si elles sont référencées dynamiquement à chaque modification de l'enregistrement en cours dans l'ensemble de données. Mettez *LookupCache* à *True* pour cacher les valeurs d'un champ de référence lorsque la propriété *LookupDataSet* ne risque aucune modification et que le nombre de valeurs de référence est faible. La mise en mémoire cache des valeurs de référence permet d'accélérer les performances, car les valeurs de référence relatives à chaque ensemble de valeurs de la propriété *LookupKeyFields* sont préchargées à l'ouverture de l'ensemble de données. En cas de modification de l'enregistrement en cours dans l'ensemble de données, l'objet champ peut localiser sa valeur (*Value*) dans la mémoire cache, plutôt que d'accéder à *LookupDataSet*. Cette amélioration des performances est appréciable si l'ensemble de données de référence (*LookupDataSet*) est sur un réseau présentant des temps d'accès lents.

Conseil Vous pouvez utiliser une mémoire cache de référence pour proposer les valeurs de référence par programme au lieu d'utiliser un ensemble de données secondaire. Vérifiez que la propriété *LookupDataSet* vaut nil. Utilisez ensuite la méthode *Add* de la propriété *LookupList* pour la remplir avec les valeurs de référence. Affectez la valeur *True* à la propriété *LookupCache*. Le champ utilise la liste de référence spécifiée sans redéfinir ses valeurs avec celles d'un ensemble de données de référence.

Si chaque enregistrement de l'ensemble de données (*DataSet*) comporte des valeurs différentes pour *KeyFields*, le temps nécessaire à la localisation des valeurs dans le cache peut être supérieur aux gains de temps obtenus grâce au placement en mémoire cache. Plus le nombre de valeurs distinctes dans *KeyFields* est élevé, plus le temps de traitement passé à localiser des valeurs dans la mémoire cache augmente.

Si *LookupDataSet* risque de changer, le fait de placer des valeurs de référence en mémoire cache peut provoquer des résultats erronés. Appelez *RefreshLookupList* pour mettre à jour les valeurs dans la mémoire cache. *RefreshLookupList* régénère la propriété *LookupList* qui contient la valeur de la propriété *LookupResultField* de chaque ensemble de valeurs *LookupKeyFields*.

Si *LookupCache* est définie à l'exécution, appelez *RefreshLookupList* pour initialiser la mémoire cache.

Définition d'un champ agrégat

Un champ agrégat affiche les valeurs issues d'un calcul portant sur un ensemble de données client. Un agrégat est la somme des données contenues dans un ensemble d'enregistrements. Voir "Utilisation des agrégats maintenus" à la page 23-13 pour plus d'informations sur les agrégats maintenus.

Pour créer un champ agrégat dans la boîte de dialogue Nouveau champ :

1. Entrez un nom pour le champ agrégat dans la zone de saisie Nom. N'entrez pas le nom d'un champ existant.
2. Choisissez le type de données d'agrégat pour le champ dans la boîte à options Type.
3. Sélectionnez l'option d'agrégat dans la boîte groupe Type de champ.
4. Choisissez OK. Le champ agrégat nouvellement défini est automatiquement ajouté à l'ensemble de données client et la propriété *Aggregates* de l'ensemble de données client est automatiquement mise à jour pour inclure la spécification d'agrégat appropriée.
5. Placez le calcul de la somme dans la propriété *ExprText* du champ agrégat nouvellement créé. Pour plus d'informations sur la définition d'une somme, voir "Spécification d'agrégats" à la page 23-14.

Une fois qu'un composant persistant TAggregateField est créé, un contrôle TDBText peut être lié au champ agrégat. Le contrôle TDBText affiche alors la valeur du champ agrégat en fonction de l'enregistrement en cours de l'ensemble de données client sous-jacent.

Suppression de champs persistants

La suppression de champs persistants peut permettre d'accéder à un sous-ensemble de colonnes d'une table et de définir vos propres champs persistants afin de remplacer une colonne. Pour supprimer un ou plusieurs champs persistants d'un ensemble de données :

1. Sélectionnez les champs à supprimer dans la boîte liste de l'éditeur de champs.
2. Appuyez sur *Suppr*.

Remarque Vous pouvez également supprimer les champs sélectionnés en choisissant Supprimer dans le menu contextuel de l'éditeur de champs.

Les champs supprimés ne sont plus disponibles pour l'ensemble de données et ne peuvent plus être affichés par les contrôles orientés données. Vous avez toujours la possibilité de recréer un composant champ persistant supprimé par accident, mais les modifications préalablement apportées à leurs propriétés ou événements seront perdues. Pour plus de détails, reportez-vous à "Création de champs persistants" à la page 19-4.

Remarque Si vous supprimez tous les composants champ persistants d'un ensemble de données, l'ensemble de données recommence à utiliser des composants champ dynamiques pour chaque colonne de la table de base de données sous-jacente.

Définition des événements et des propriétés des champs persistants

Vous pouvez définir les propriétés et personnaliser les événements des composants champ persistants lors de la conception. Les propriétés contrôlent la façon dont un champ est affiché par un composant orienté données (elles déterminent par exemple s'il doit apparaître dans un composant *TDBGrid* ou bien si sa valeur peut être modifiée). Les événements contrôlent ce qui se passe quand les données d'un champ sont extraites, modifiées, définies ou validées.

Pour définir les propriétés d'un composant champ ou écrire des gestionnaires d'événements personnalisés pour celui-ci, sélectionnez-le dans l'éditeur de champs ou dans la liste de l'inspecteur d'objets.

Définition des propriétés d'affichage et d'édition en mode conception

Pour éditer les propriétés d'affichage d'un composant champ sélectionné, accédez à la page Propriétés de l'inspecteur d'objets. Le tableau suivant dresse la liste des propriétés d'affichage pouvant être éditées.

Tableau 19.3 Propriétés du composant champ

Propriété	Finalité
Alignment	Aligne le contenu d'un champ à gauche, à droite ou au centre dans un composant orienté données.
ConstraintErrorMessage	Spécifie le texte à afficher en cas de condition de contrainte.
CustomConstraint	Spécifie une contrainte locale à appliquer aux données lors de l'édition.
Currency	Champs numériques seulement. *True* : affiche des valeurs monétaires. *False* (par défaut) : n'affiche pas les valeurs monétaires.
DisplayFormat	Spécifie le format d'affichage dans un composant orienté données.
DisplayLabel	Spécifie le nom de colonne d'un champ dans un composant grille orienté données.
DisplayWidth	Spécifie la largeur, en caractères, d'une colonne de grille affichant le contenu de ce champ.
EditFormat	Spécifie le format d'édition dans un composant orienté données.
EditMask	Limite l'entrée de données dans un champ de saisie aux types et aux étendues de caractères spécifiés. Spécifie également les caractères spéciaux, non éditables, qui apparaissent dans le champ (tirets, parenthèses, etc.).
FieldKind	Spécifie le type du champ à créer.
FieldName	Spécifie le nom réel d'une colonne de la table dont le champ dérive sa valeur et son type de données.

Tableau 19.3 Propriétés du composant champ (suite)

Propriété	Finalité
HasConstraints	Indique si des conditions de contrainte ont été imposées à un champ.
ImportedConstraint	Spécifie une contrainte SQL importée du dictionnaire de données ou d'un serveur SQL.
Index	Spécifie la position du champ dans un ensemble de données.
LookupDataSet	Spécifie la table utilisée pour référencer les valeurs de champs lorsque *Lookup* est à *True*.
LookupKeyFields	Spécifie le champ ou les champs de l'ensemble de données de référence devant se correspondre lors d'un référencement.
LookupResultField	Spécifie le champ de l'ensemble de données de référence dont la valeur doit être copiée dans le champ de référence
MaxValue	Champs numériques seulement. Spécifie la valeur maximum que l'utilisateur peut entrer dans le champ.
MinValue	Champs numériques seulement. Spécifie la valeur minimum que l'utilisateur peut entrer dans le champ.
Name	Spécifie le nom du composant utilisé pour faire référence du composant champ dans Delphi.
Origin	Spécifie le nom du champ tel qu'il apparaît dans la base de données sous-jacente.
Precision	Champs numériques uniquement. Spécifie le nombre de chiffres significatifs.
ReadOnly	*True* : affiche les valeurs de champ dans les contrôles orientés données, mais en interdit l'édition. *False* (par défaut) : autorise l'affichage et l'édition des valeurs du champ.
Size	Spécifie le nombre maximum de caractères pouvant être affichés ou entrés dans un champ chaîne, ou bien la taille en octets des champs *TBytesField* et *TVarBytesField*.
Tag	Compartiment entier long que le programmeur peut utiliser dans tous les composants.
Transliterate	*True* (par défaut) : spécifie qu'une traduction sera effectuée vers et depuis les locales respectives au fur et à mesure que des données sont transférées entre un ensemble de données et une base de données. *False* : spécifie que ces traductions ne seront pas effectuées.
Visible	*True* (valeur par défaut) : permet l'affichage du champ dans une grille orientée données. *False* : empêche l'affichage du champ dans un composant grille orienté données. Les composants définis par l'utilisateur peuvent afficher les décisions prises en fonction de cette propriété.

Toutes les propriétés ne sont pas disponibles pour l'ensemble des composants champ. Par exemple, un composant champ de type *TStringField* ne peut pas avoir les propriétés *Currency*, *MaxValue* ou *DisplayFormat* et un composant de type *TFloatField* ne peut pas avoir de propriété *Size*.

Alors que le rôle de la plupart des propriétés est évident, certaines comme *Calculated* nécessitent des étapes de programmation supplémentaires. D'autres, comme *DisplayFormat*, *EditFormat* et *EditMask* sont reliées entre elles ; leur configuration doit être coordonnée. Pour plus de détails sur l'utilisation des propriétés *DisplayFormat*, *EditFormat* et *EditMask*, voir "Contrôle ou dissimulation de la saisie utilisateur" à la page 19-17.

Définition des propriétés des composants champ à l'exécution

Les propriétés des composants champ peuvent aussi être utilisées et manipulées à l'exécution. Accédez aux composants champ persistants par leur nom, celui-ci étant obtenu en concaténant le nom du champ et le nom de l'ensemble de données.

Par exemple, le code ci-dessous donne la valeur *True* à la propriété *ReadOnly* du champ *CityStateZip* de la table *Customers* :

```
CustomersCityStateZip.ReadOnly := True;
```

L'instruction suivante change l'ordre des champs en donnant la valeur 3 à la propriété *Index* du champ *CityStateZip* de la table *Customers* :

```
CustomersCityStateZip.Index := 3;
```

Création des ensembles d'attributs pour les composants champ

Lorsque plusieurs champs des ensembles de données utilisés par votre application partagent des propriétés de formatage (telles que *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, et ainsi de suite), il est plus pratique de les définir pour un seul champ, puis de les stocker comme ensemble d'attributs dans le dictionnaire des données. Ils peuvent alors être appliqués facilement à d'autres champs.

Remarque Les ensembles d'attributs et le dictionnaire de données ne sont disponibles que pour les ensembles de données BDE.

Pour créer un ensemble d'attributs d'après un composant champ d'un ensemble de données :

1. Double-cliquez sur l'ensemble de données pour lancer l'éditeur de champs.
2. Sélectionnez le champ dont vous voulez définir les propriétés.
3. Définissez les propriétés voulues pour ce champ dans l'inspecteur d'objets.
4. Cliquez avec le bouton droit de la souris sur la boîte liste de l'éditeur de champs pour ouvrir le menu contextuel.
5. Choisissez Enregistrer les attributs pour enregistrer la propriété du champ en cours comme ensemble d'attributs dans le dictionnaire de données.

Par défaut, l'ensemble d'attributs prend le nom du champ en cours. Vous pouvez toutefois spécifier un nom différent en choisissant "Enregistrer les attributs sous" au lieu de la commande "Enregistrer les attributs".

Lorsque vous avez créé un nouvel ensemble d'attributs et que vous l'avez ajouté au dictionnaire de données, vous pouvez l'associer à d'autres composants champ

persistants. Même si vous supprimez cette associationultérieurement, l'ensemble d'attributs reste défini dans le dictionnaire de données.

Remarque Vous pouvez également créer des ensembles d'attributs directement depuis l'explorateur SQL. Lorsque vous créez un ensemble d'attributs avec l'explorateur SQL, il est ajouté au dictionnaire de données mais il n'est pas encore appliqué à un champ. L'explorateur SQL vous permet de spécifier deux attributs supplémentaires : un type de champ (tel que *TFloatField*, *TStringField*, etc.) et un contrôle orienté données (*TDBEdit*, *TDBCheckBox*, etc.) placé automatiquement sur une fiche lorsque vous y faites glisser un champ basé sur l'ensemble d'attributs. Pour en savoir plus à ce sujet, reportez-vous à l'aide en ligne de l'explorateur SQL.

Association des ensembles d'attributs aux composants champ

Lorsque plusieurs champs des ensembles de données utilisés par votre application partagent des propriétés de formatage (*Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, etc.) et si vous avez enregistré ces propriétés en tant qu'ensembles d'attributs dans le dictionnaire de données, vous pouvez facilement appliquer ceux-ci aux champs sans avoir à les recréer manuellement pour chacun d'eux. En outre, si vous changez ultérieurement la configuration des attributs dans le dictionnaire de données, ces modifications seront répercutées automatiquement à chaque champ associé, la prochaine fois que des composants champ seront ajoutés à l'ensemble de données.

Pour attribuer un ensemble d'attributs à un composant champ :

1. Double-cliquez sur l'ensemble de données pour lancer l'éditeur de champs.
2. Sélectionnez le champ auquel l'ensemble d'attributs doit s'appliquer.
3. Cliquez avec le bouton droit de la souris sur la boîte liste et choisissez Associer les attributs.
4. Sélectionnez ou entrez l'ensemble d'attributs à appliquer depuis la boîte de dialogue Association d'attributs. Si un ensemble d'attributs porte ce nom dans le dictionnaire de données, il apparaît dans la boîte de saisie.

Important Si l'ensemble d'attributs défini dans le dictionnaire de données est modifié par la suite, vous devrez à nouveau l'appliquer à chaque composant champ qui l'utilise. Vous pouvez appeler l'éditeur de champs et sélectionner plusieurs composants champ de l'ensemble de données lorsque vous ré-appliquez les attributs.

Suppression des associations d'ensembles d'attributs

Si vous changez d'avis concernant l'association d'un ensemble d'attributs à un champ, vous pouvez facilement supprimer l'association en procédant comme suit :

1. Appelez l'éditeur de champs correspondant depuis l'ensemble de données contenant le champ.
2. Sélectionnez le champ pour lequel vous voulez supprimer l'ensemble d'attributs.

3 Appelez le menu contextuel de l'éditeur de champs et choisissez Dissocier les attributs.

Important Le fait de dissocier un ensemble d'attributs ne modifie pas les propriétés du champ. Le champ garde les paramètres qu'il avait lorsque l'ensemble d'attributs lui a été affecté. Pour modifier ces propriétés, sélectionnez le champ dans l'éditeur de champs et définissez les propriétés dans l'inspecteur d'objets.

Contrôle ou dissimulation de la saisie utilisateur

La propriété *EditMask* permet de contrôler le type et la portée des valeurs qu'un utilisateur peut entrer dans un composant orienté données associé à des composants *TStringField*, *TDateField*, *TTimeField*, *TDateTimeField* et *TSQLTimeStampField*. Vous pouvez soit utiliser des masques existants ou créer les vôtres. La manière la plus simple consiste à faire appel à l'éditeur de masques de saisie. Cependant, vous avez aussi la possibilité d'entrer les masques directement dans le champ *EditMask* de l'inspecteur d'objets.

Remarque Pour les composants *TStringField*, la propriété *EditMask* définit aussi le format d'affichage.

Pour entrer un masque de saisie, appelez l'éditeur de masques de saisie d'un composant champ :

1 Sélectionnez le composant dans l'éditeur de champs ou dans l'inspecteur d'objets.

2 Cliquez sur la page Propriétés de l'inspecteur d'objets.

3 Double-cliquez sur la colonne de la propriété *EditMask* dans l'inspecteur d'objets ou cliquez sur le bouton à points de suspension. L'éditeur de masques de saisie s'ouvre.

La zone de saisie "Masque" vous permet de créer et d'éditer un format de masque. La grille Masques exemple vous permet de sélectionner des masques prédéfinis. Si vous en sélectionnez un, son format apparaît dans la zone de saisie et vous pouvez alors soit le modifier, soit l'utiliser tel quel. La zone Test de saisie vous permet également de tester les entrées utilisateur.

Le bouton Masques vous permet de charger un ensemble de masques personnalisé (si vous en avez créé un) dans la grille Masques exemple.

Utilisation des formats par défaut pour les champs numériques, date et heure

Delphi fournit des routines intégrées d'affichage et d'édition ainsi qu'un formatage par défaut intelligent pour les composants *TFloatField*, *TCurrencyField*, *TBCDField*, *TFMTBCDField*, *TIntegerField*, *TSmallIntField*, *TWordField*, *TDateField*, *TDateTimeField*, *TTimeField* et *TSQLTimeStampField*. Vous n'avez donc rien à faire pour utiliser ces routines.

Le formatage par défaut est effectué par les routines suivantes :

Tableau 19.4 Routines de formatage des composants champ

Routine	Utilisée par . . .
FormatFloat	*TFloatField, TCurrencyField*
FormatDateTime	*TDateField, TTimeField, TDateTimeField*
SQLTimeStampToString	*TSQLTimeStampField*
FormatCurr	*TCurrencyField, TBCDField*
BcdToStrF	TFMTBcdField

Seules les propriétés de format appropriées au type de données d'un composant champ sont disponibles pour un composant donné.

Les conventions de format par défaut des valeurs de date, d'heure, monétaire et numériques reposent sur les propriétés des Options régionales définies dans le Panneau de configuration. Par exemple, si vous utilisez les paramètres par défaut des Etats-Unis, la colonne *TFloatField* dont la propriété *Currency* est à *True* remplace la valeur 1234,56 de la propriété *DisplayFormat* par $1234.56, tandis que la valeur de *EditFormat* devient 1234.56.

Au moment de la conception ou de l'exécution, vous pouvez éditer les propriétés *DisplayFormat* et *EditFormat* d'un composant champ pour ignorer la configuration d'affichage par défaut de ce champ. Vous pouvez également écrire des gestionnaires d'événements *OnGetText* et *OnSetText* pour effectuer un formatage personnalisé des composants champ lors de l'exécution.

Gestion des événements

A l'instar de tous les composants, les composants champ ont des gestionnaires d'événements associés. Il est possible d'affecter des méthodes comme gestionnaire de ces événements. Ceux-ci vous permettent de contrôler les événements affectant les données saisies dans les champs au moyen de contrôles orientés données et d'effectuer des actions spécifiques. Le tableau suivant dresse la liste des événements associés aux composants champ :

Tableau 19.5 Evénements des composants champ

Evénement	Finalité
OnChange	Appelé lors du changement de la valeur d'un champ.
OnGetText	Appelé lors de l'extraction de la valeur d'un composant champ pour affichage ou édition.
OnSetText	Appelé lors de la définition de la valeur d'un composant champ.
OnValidate	Appelé pour valider la valeur d'un composant champ à chaque fois qu'elle est modifiée suite à une édition ou une insertion.

Les événements *OnGetText* et *OnSetText* sont surtout utiles aux programmeurs qui veulent aller au-delà des fonctions de formatage intégrées. *OnChange* permet d'effectuer des tâches spécifiques à une application et associées aux modifications

de données, comme l'activation ou la désactivation de menus ou de contrôles visuels. *OnValidate* est utile pour valider la saisie de données dans votre application avant de renvoyer les valeurs à un serveur de base de données.

Pour écrire un gestionnaire d'événement pour un composant champ :

1. Sélectionnez le composant.
2. Sélectionnez la page Evénements dans l'inspecteur d'objets.
3. Double-cliquez sur la colonne des valeurs du gestionnaire d'événement pour ouvrir la fenêtre de code source correspondante.
4. Créez ou éditez le code du gestionnaire.

Manipulation des méthodes de champ lors de l'exécution

Les méthodes des composants champ disponibles au moment de l'exécution vous permettent de convertir les valeurs des champs d'un type en un autre et de placer la focalisation sur le premier contrôle orienté données d'une fiche associée à un composant champ.

Le contrôle de la focalisation des composants orientés données associés à un champ est important quand votre application effectue la validation de données orientées enregistrement dans un gestionnaire d'événement d'ensemble de données (comme *BeforePost)*. Il est possible de procéder à la validation des champs d'un enregistrement, que son contrôle orienté données associé ait la focalisation ou non. Si la validation échoue pour un champ particulier de l'enregistrement, vous devez faire en sorte que le contrôle contenant les données à l'origine de l'erreur reçoive la focalisation afin que l'utilisateur puisse entrer les corrections.

Pour contrôler la focalisation des composants orientés données d'un champ, vous devez recourir à la méthode*FocusControl*. Celle-ci place la focalisation sur le premier contrôle orienté données d'une fiche associé à un champ. Un gestionnaire d'événement doit faire appel à la méthode *FocusControl* d'un champ avant de le valider. Le code ci-dessous montre comment faire appel à la méthode *FocusControl* pour le champ *Company* de la table *Customers* :

```
CustomersCompany.FocusControl;
```

Le tableau suivant dresse la liste des autres méthodes des composants champ et présente leur utilisation. Pour une liste complète des méthodes et des informations détaillées sur leur utilisation, voir les entrées sur *TField* et ses descendants dans la *Référence VCL*.

Tableau 19.6 Méthodes des composants champ

Méthode	Utilisation
AssignValue	Affecte à une valeur de champ une valeur spécifiée en utilisant une fonction de conversion basée sur le type du champ.
Clear	Efface le champ et met sa valeur à NULL.

Tableau 19.6 Méthodes des composants champ (suite)

Méthode	Utilisation
GetData	Extrait du champ des données non formatées.
IsValidChar	Détermine si un caractère saisi par un utilisateur dans un contrôle orienté données dans le but de définir une valeur est autorisé.
SetData	Affecte des données “brutes” au champ.

Affichage, conversion et accès aux valeurs des champs

Les contrôles orientés données comme *TDBEdit* et *TDBGrid* affichent automatiquement les valeurs associées aux composants champ. Si l’édition est activée pour l’ensemble de données et les contrôles, les contrôles orientés données peuvent aussi envoyer des valeurs nouvelles et modifiées dans la base de données. En général, les propriétés et les méthodes incorporées des contrôles orientés données leur permettent de se connecter à des ensembles de données, d’afficher des valeurs et d’effectuer des mises à jour sans que cela requière de programmation supplémentaire de votre part. Utilisez-les autant que possible dans vos applications de bases de données. Pour plus d’informations sur les contrôles orientés données, voir chapitre 15, “Utilisation de contrôles de données”.

Les contrôles standard peuvent aussi afficher et éditer des valeurs de bases de données associées à des composants champ. L’utilisation de contrôles standard peut toutefois nécessiter un supplément de programmation de votre part. Par exemple, lorsque votre application utilise des contrôles standard, elle doit déterminer à quel moment les mettre à jour car les valeurs de champ évoluent. Si l’ensemble de données possède un composant source de données, vous pouvez utiliser ses événements à cet effet. En particulier, l’événement *OnDataChange* vous permet de savoir à quel moment vous devez mettre à jour la valeur d’un contrôle et l’événement *OnStateChange* peut vous aider à déterminer à quel moment vous devez activer ou désactiver les contrôles. Pour plus d’informations sur ces événements, voir “Réponse aux modifications effectuées par le biais de la source de données” à la page 15-5.

Les rubriques suivantes décrivent la manipulation de valeurs de champ afin de les afficher dans des contrôles standard.

Affichage de valeurs dans les contrôles standard

Une application peut accéder à la valeur d’une colonne d’ensemble de données au moyen de la propriété *Value* d’un composant champ. Le gestionnaire d’événement *OnDataChange* ci-dessous, par exemple, met à jour le texte contenu dans un contrôle *TEdit* car la valeur du champ *CustomersCompany* a peut-être évolué :

```
procedure TForm1.CustomersDataChange(Sender: TObject, Field: TField);
begin
  Edit3.Text := CustomersCompany.Value;
end;
```

Cette méthode fonctionne bien pour les valeurs chaîne, mais elle peut nécessiter un surcroît de programmation pour pouvoir gérer les conversions d'autres types de données. Heureusement, les composants champ ont des propriétés intégrées leur permettant de le faire.

Remarque Vous pouvez également utiliser des Variants pour accéder aux valeurs des champs et les définir. Pour plus de détails sur l'utilisation des variants pour accéder aux valeurs des champs et les définir, reportez-vous à "Accès à des valeurs par la propriété par défaut d'un ensemble de données" à la page 19-23.

Conversion des valeurs de champs

Le rôle de ces propriétés est de convertir un type de données en un autre. Par exemple, la fonction propriété *AsString* convertit les valeurs numériques et booléennes en représentations chaîne. Le tableau suivant donne la liste des propriétés de conversion des composants champ et indique celles conseillées en fonction de leur classe :

	AsVariant	AsString	AsInteger	AsFloat AsCurrency AsBCD	AsDateTime AsSQLTimeStamp	AsBoolean
TStringField	Oui	N/D	Oui	Oui	Oui	Oui
TWideStringField	Oui	Oui	Oui	Oui	Oui	Oui
TIntegerField	Oui	Oui	N/D	Oui		
TSmallIntField	Oui	Oui	Oui	Oui		
TWordField	Oui	Oui	Oui	Oui		
TLargeintField	Oui	Oui	Oui	Oui		
TFloatField	Oui	Oui	Oui	Oui		
TCurrencyField	Oui	Oui	Oui	Oui		
TBCDField	Oui	Oui	Oui	Oui		
TFMTBCDField	Oui	Oui	Oui	Oui		
TDateTimeField	Oui	Oui		Oui	Oui	
TDateField	Oui	Oui		Oui	Oui	
TTimeField	Oui	Oui		Oui	Oui	
TSQLTimeStampField	Oui	Oui		Oui	Oui	
TBooleanField	Oui	Oui				
TBytesField	Oui	Oui				
TVarBytesField	Oui	Oui				
TBlobField	Oui	Oui				
TMemoField	Oui	Oui				
TGraphicField	Oui	Oui				
TVariantField	N/D	Oui	Oui	Oui	Oui	Oui
TAggregateField	Oui	Oui				

Vous remarquerez que certaines colonnes du tableau font référence à plusieurs propriétés de conversion (telles que *AsFloat*, *AsCurrency* et *AsBCD*). En effet, tous les types de données de champ qui prennent en charge l'une de ces propriétés prennent également en charge les autres.

Vous remarquerez également que la propriété *AsVariant* peut traduire tous les types de données. Pour tout type de données ne figurant pas dans la liste ci-dessus, *AsVariant* est également disponible (et constitue, de fait, la seule option). Utilisez-la en cas de doute.

Il est des cas où la conversion n'est pas possible. Par exemple, *AsDateTime* permet de convertir une chaîne au format date, heure ou date/heure seulement si la valeur de la chaîne est dans un format date/heure identifiable. L'échec d'une tentative de conversion a pour effet de provoquer une exception.

Dans certains autres cas, la conversion est possible, mais les résultats sont parfois imprévisibles. Par exemple, que signifie la conversion d'une valeur *TDateTimeField* au format flottant ? *AsFloat* convertit la partie date du champ en nombre de jours à partir du 31/12/1899 et la partie heure du champ en une fraction de 24 heures. Le tableau suivant donne la liste des conversions autorisées qui donnent des résultats spéciaux :

Tableau 19.7 Résultats de conversion spéciaux

Conversion	Résultat
De String à Boolean	Convertit "True", "False", "Yes" et "No" en valeur booléenne. Les autres valeurs provoquent une exception.
De Float à Integer	Arrondit la valeur à virgule flottante à l'entier le plus proche.
De DateTime ou SQLTimeStamp à Float	Convertit la date en nombre de jours à partir du 31 décembre 1899 et l'heure en fraction de 24 heures.
De Boolean à String	Convertit toute valeur booléenne en "True" ou "False".

Dans les autres cas, la conversion est totalement impossible, toute tentative provoquant une exception.

La conversion a toujours lieu avant l'affectation. Par exemple, l'instruction ci-dessous convertit la valeur de *CustomersCustNo* en chaîne et affecte celle-ci au texte d'un contrôle de saisie :

```
Edit1.Text := CustomersCustNo.AsString;
```

A l'inverse, l'instruction suivante affecte le texte d'un contrôle de saisie au champ *CustomersCustNo* en tant qu'entier :

```
MyTableMyField.AsInteger := StrToInt(Edit1.Text);
```

Accès à des valeurs par la propriété par défaut d'un ensemble de données

La méthode la plus générale pour accéder à la valeur d'un champ consiste à recourir à la propriété *FieldValues*. Par exemple, l'instruction suivante prend la valeur d'une zone de saisie et la transfère dans le champ *CustNo* de la table *Customers* :

```
Customers.FieldValues['CustNo'] := Edit2.Text;
```

Etant donné que la propriété FieldValues est de type Variant, elle convertit automatiquement les autres types de données en une valeur Variant.

Pour en savoir plus sur les Variants, voir l'aide en ligne.

Accès à des valeurs par la propriété Fields d'un ensemble de données

Vous pouvez accéder à la valeur d'un champ par la propriété *Fields* du composant ensemble de données auquel appartient le champ. *Fields* gère une liste indexée de tous les champs de l'ensemble de données. C'est utile si vous devez parcourir un certain nombre de colonnes ou si votre application utilise des tables non disponibles lors de la conception.

Pour utiliser la propriété *Fields*, vous devez connaître l'ordre et le type de données d'un champ de l'ensemble de données. Pour spécifier le champ auquel vous voulez accéder, vous devez utiliser un nombre ordinal. Le premier champ d'un ensemble de données porte le numéro 0. La valeur des champs doit être convertie correctement à l'aide des propriétés de conversion de chaque composant champ. Pour plus de détails sur les propriétés de conversion, reportez-vous à "Conversion des valeurs de champs" à la page 19-21.

Par exemple, l'instruction ci-dessous affecte la valeur actuelle de la septième colonne (pays) de la table *Customers* à un contrôle de saisie :

```
Edit1.Text := CustTable.Fields[6].AsString;
```

A l'inverse, vous pouvez affecter une valeur à un champ en définissant la propriété *Fields* de l'ensemble de données pour le champ voulu. Exemple :

```
begin
  Customers.Edit;
  Customers.Fields[6].AsString := Edit1.Text;
  Customers.Post;
end;
```

Accès à des valeurs par la méthode FieldByName d'un ensemble de données

Vous pouvez également accéder à la valeur d'un champ par la méthode *FieldByName* d'un ensemble de données. Cette méthode est utile si vous connaissez le nom du champ auquel vous voulez accéder, mais si vous n'avez pas accès à la table sous-jacente au moment de la conception.

Pour utiliser *FieldByName*, vous devez connaître l'ensemble de données et le nom du champ auxquels vous voulez accéder pour transmettre le nom du champ à la méthode en tant qu'argument. Pour accéder à la valeur du champ ou la modifier, vous devez convertir le résultat avec la propriété de conversion de composant champ appropriée, comme par exemple *AsString* ou *AsInteger*. Par exemple, l'instruction suivante affecte la valeur du champ *CustNo* de l'ensemble de données *Customers* à un contrôle de saisie :

```
Edit2.Text := Customers.FieldByName('CustNo').AsString;
```

A l'inverse, vous pouvez affecter une valeur à un champ :

```
begin
  Customers.Edit;
  Customers.FieldByName('CustNo').AsString := Edit2.Text;
  Customers.Post;
end;
```

Définition de la valeur par défaut d'un champ

Vous pouvez spécifier la façon dont la valeur par défaut d'un champ dans un ensemble de données client ou un ensemble de données BDE doit être calculée à l'exécution en utilisant la propriété *DefaultExpression*. Cette propriété peut être toute expression SQL valide ne faisant pas référence aux valeurs des champs. Si l'expression contient des valeurs littérales autres que des valeurs numériques, elles doivent être entourées de guillemets. Par exemple, si un champ heure contient une valeur par défaut équivalente à midi, elle apparaît ainsi :

```
'12:00:00'
```

Remarque Si la table de base de données sous-jacente définit une valeur par défaut pour le champ, la valeur par défaut que vous spécifiez dans *DefaultExpression* est prioritaire. En effet, *DefaultExpression* s'applique lorsque l'ensemble de données valide l'enregistrement contenant le champ, c'est-à-dire avant l'application de l'enregistrement modifié au serveur de base de données.

Utilisation de contraintes

Les composants champ des ensembles de données ou des ensembles de données BDE peuvent utiliser des contraintes SQL importées du serveur. De plus, vos applications peuvent créer et utiliser des contraintes personnalisées appliquées à

ces ensembles de donnés localement par rapport à votre application. Toutes les contraintes sont des règles ou des conditions qui imposent une limite à la portée ou à l'intervalle des valeurs que le champ peut stocker.

Création de contrainte personnalisée

Une contrainte personnalisée n'est pas importée du serveur comme les autres contraintes. C'est une contrainte que vous déclarez, implémentez et rendez effective dans votre application locale. Ainsi, les contraintes personnalisées permettent une pré-validation des données saisies mais elles ne peuvent pas être appliquées aux données reçues d'une application serveur ou émises vers celle-ci

Pour créer une contrainte personnalisée, définissez la propriété *CustomConstraint* de façon à ce qu'elle spécifie une condition de contrainte et affectez à la propriété *ConstraintErrorMessage* le texte du message affiché lorsqu'un utilisateur effectue une violation de contrainte à l'exécution.

CustomConstraint est une chaîne SQL qui spécifie une contrainte propre à l'application et imposée à la valeur du champ. Définissez *CustomConstraint* de façon à limiter les valeurs pouvant être saisies par un utilisateur dans un champ. *CustomConstraint* peut être toute expression de recherche SQL valide telle que :

```
x > 0 and x < 100
```

Le nom utilisé pour faire référence à la valeur du champ peut être toute chaîne autre qu'un mot SQL réservé, tant qu'elle est utilisée avec homogénéité à l'intérieur de l'expression de contrainte.

Remarque Les contraintes spécialisées ne sont disponibles que pour les ensembles de données BDE et client.

Les contraintes spécialisées s'ajoutent à toute contrainte défini par le serveur quant aux valeurs des données du champ. Pour connaître les contraintes imposées par le serveur, lisez la valeur de la propriété *ImportedConstraint*.

Utilisation des contraintes du serveur

La plupart des bases de données SQL d'exploitation utilisent les contraintes pour imposer des conditions sur les valeurs possibles d'un champ. Par exemple, un champ peut ne pas autoriser de valeurs NULL, peut exiger qu'une valeur soit unique pour une colonne et que ces valeurs soient comprises entre *0* et *150*. Comme vous devez dupliquer de telles conditions dans vos applications client, les ensembles de données client et les ensembles de données BDE possèdent une propriété *ImportedConstraint* permettant de diffuser localement les contraintes du serveur.

ImportedConstraint est une propriété en lecture seule qui spécifie une clause SQL limitant les valeurs de champ d'une certaine manière. Par exemple :

```
Value > 0 and Value < 100
```

Ne modifiez pas la valeur de *ImportedConstraint*, sauf si vous voulez modifier des commandes SQL non standard ou propres à un serveur qui ont été importées en tant que commentaires, car le moteur de bases de données n'a pas pu les interpréter.

Pour ajouter des contraintes supplémentaires sur la valeur d'un champ, utilisez la propriété *CustomConstraint*. Les contraintes personnalisées sont imposées en plus des contraintes importées. Si les contraintes du serveur changent, la valeur de *ImportedConstraint* change aussi, mais les contraintes introduites dans la propriété *CustomConstraint* persistent.

La suppression des contraintes de la propriété *ImportedConstraint* n'entraîne pas la validité des valeurs de champs qui sont en violation avec ces contraintes. La suppression des contraintes signifie simplement que la vérification est effectuée par le serveur et non plus localement. Lorsque les contraintes sont vérifiées localement, le message d'erreur spécifié dans la propriété *ConstraintErrorMessage* apparaît au moment où les violations sont détectées (sinon, les messages d'erreur affichés proviennent du serveur).

Utilisation des champs objet

Les champs objet représentent un composé d'autres types de données plus simples, parmi lesquels les champs ADT (Abstract Data Type), tableau, ensemble de données et référence. Ces types de champ contiennent des champs enfants ou d'autres ensembles de données ou y font référence.

Les champs ADT et tableau contiennent des champs enfants. Les champs enfants d'un champ ADT peuvent être de n'importe quel type scalaire ou type d'objet (c'est-à-dire, de tout autre type de champ). Ces champs enfants peuvent varier quant à leur type. Un champ tableau contient un tableau de champs enfants de type identique.

Les champs ensemble de données et de référence sont des champs qui accèdent à d'autres ensembles de données. Un champ ensemble de données permet d'accéder à un ensemble de données (détail) imbriqué. Un champ de référence stocke un pointeur (référence) sur un autre objet persistant (ADT).

Tableau 19.8 Types de composants champ objet

Nom du composant	Utilisation
TADTField	Représente un champ ADT (Abstract Data Type).
TArrayField	Représente un champ tableau.
TDataSetField	Représente un champ contenant une référence à un ensemble de données imbriqué.
TReferenceField	Représente un champ REF, pointant sur un champ ADT.

Quand vous ajoutez des champs avec l'éditeur de champs à un ensemble de données contenant des champs objet, des champs objet persistants du type correct sont automatiquement créés pour vous. Lorsque vous ajoutez des champs objet persistants à un ensemble de données, la propriété *ObjectView* de

l'ensemble de données prend automatiquement la valeur *True*, ce qui amène l'ensemble de données à stocker ces champs de façon hiérarchisée, plutôt qu'à les aplanir comme si les champs enfants constitutifs étaient des champs séparés indépendants.

Les propriétés suivantes sont communes à tous les champs objet et permettent de gérer les ensembles de données et les champs enfants.

Tableau 19.9 Propriétés communes des descendants de champs objet

Propriété	Utilisation
Fields	Contient les champs enfants du champ objet.
ObjectType	Classe le champ objet.
FieldCount	Nombre de champs enfants appartenant au champ objet.
FieldValues	Permet d'accéder aux valeurs des champs enfants.

Affichage des champs ADT et tableau

Les champs ADT et tableau contiennent des champs enfants qui peuvent être affichés par le biais de contrôles orientés données.

Les contrôles orientés données tels que *TDBEdit* qui représentent une valeur de champ unique affichent les valeurs des champs enfants dans une chaîne non modifiable, séparés par des virgules. En outre, si vous attribuez à la propriété *DataField* du contrôle le champ enfant au lieu du champ objet, le champ enfant peut être visualisé et modifié comme tout autre champ de données normal.

Un contrôle *TDBGrid* affiche différemment les données de champs ADT et tableau, suivant la valeur de la propriété *ObjectView* de l'ensemble de données. Lorsque *ObjectView* vaut *False*, chaque champ enfant apparaît dans une seule colonne. Lorsque *ObjectView* vaut *True*, un champ ADT ou tableau peut être développé et réduit en cliquant sur la flèche dans la barre de titre de la colonne. Lorsque le champ est développé, chaque champ enfant apparaît dans sa propre colonne et barre de titre, et tous sous la barre de titre du champ ADT ou tableau. Lorsque le champ ADT ou tableau est réduit, seule une colonne apparaît, contenant une chaîne non modifiable des champs enfants, séparés par des virgules.

Utilisation des champs ADT

Les types de champs ADT sont définis par l'utilisateur et créés sur le serveur. Ils s'apparentent au type d'enregistrement. Un type de champ ADT peut contenir la plupart des champ scalaires, des champs tableau, des champs de référence et des champs ADT imbriqués.

Il existe plusieurs façons d'accéder aux données des types de champ ADT. Elles sont illustrés dans les exemples suivants, qui affectent une valeur de champ

enfant à une zone de saisie appelée *CityEdit* et utilisent la structure ADT suivante :

```
Address
  Street
  City
  State
  Zip
```

Utilisation de composants champ persistant

La façon la plus facile d'accéder à des valeurs de champ ADT consiste à utiliser des composants champ persistant. Dans le cas de la structure ADT ci-dessus, les champs persistants suivants peuvent être ajoutés à la table *Customer* à l'aide de l'éditeur de champs :

```
CustomerAddress: TADTField;
CustomerAddrStreet: TStringField;
CustomerAddrCity: TStringField;
CustomerAddrState: TStringField;
CustomerAddrZip: TStringField;
```

A partir de ces champs persistants, vous pouvez accéder simplement aux champs enfants d'un champ ADT par nom :

```
CityEdit.Text := CustomerAddrCity.AsString;
```

Bien que les champs persistants représentent la façon la plus facile d'accéder aux champs enfants ADT, il est impossible de les utiliser si la structure de l'ensemble de données est inconnue lors de la conception. Lorsque vous accédez à des champs enfants ADT sans utiliser de champs persistants, vous devez attribuer la valeur *True* à la propriété *ObjectView* de l'ensemble de données.

Utilisation de la méthode FieldByName d'un ensemble de données

Vous pouvez accéder aux enfants d'un champ ADT à l'aide de la méthode *FieldByName* de l'ensemble de données en qualifiant le nom du champ enfant à partir du nom du champ ADT :

```
CityEdit.Text := Customer.FieldByName('Address.City').AsString;
```

Utilisation de la propriété FieldValues d'un ensemble de données

Vous pouvez également utiliser des noms de champ qualifiés avec la propriété *FieldValues* d'un ensemble de données :

```
CityEdit.Text := Customer['Address.City'];
```

Vous pouvez ignorer le nom de la propriété (FieldValues) car FieldValues est la propriété par défaut de l'ensemble de données.

Remarque A la différence des autres méthodes disponibles à l'exécution pour accéder aux valeurs de champs enfants ADT, la propriété *FieldValues* fonctionne même si la propriété *ObjectView* de l'ensemble de données vaut *False*.

Utilisation de la propriété FieldValues d'un champ ADT

Vous pouvez accéder à la valeur d'un champ enfant avec la propriété *FieldValues* de TADTField. *FieldValues* accepte et renvoie un *Variant* ; elle peut dont traiter et convertir des champs de n'importe quel type. Le paramètre d'index accepte une valeur entière qui spécifie le décalage du champ.

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).FieldValues[1];
```

FieldValues étant la propriété par défaut de *TADTField*, le nom de propriété (FieldValues) peut être ignoré. Par conséquent, l'instruction suivante est équivalente à la précédente :

```
CityEdit.Text := TADTField(Customer.FieldByName('Address'))[1];
```

Utilisation de la propriété Fields d'un champ ADT

Chaque champ ADT possède une propriété *Fields* analogue à la propriété *Fields* d'un ensemble de données. De même que la propriété *Fields* d'un ensemble de données, vous pouvez l'utiliser pour accéder aux champs enfants par position :

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).Fields[1].AsString;
```

ou par nom :

```
CityEdit.Text :=
TADTField(Customer.FieldByName('Address')).Fields.FieldByName('City').AsString;
```

Utilisation des champs tableau

Les champs tableau se composent d'un ensemble de champs de même type. Les types de champ peuvent être scalaires (par exemple à virgule flottante, chaîne) ou non scalaires (ADT), mais un champ tableau de tableaux n'est pas autorisé. La propriété SparseArrays de *TDataSet* détermine si un objet unique *TField* est créé pour chaque élément du champ tableau.

Il existe plusieurs façons d'accéder aux données contenues dans les types de champs tableau. Si vous n'utilisez pas de champs persistants, vous devez attribuer la valeur *True* à la propriété *ObjectView* de l'ensemble de données afin d'accéder aux éléments d'un champ tableau.

Utilisation de champs persistants

Vous pouvez mapper les champs persistants avec les différents éléments de tableau d'un champ tableau. Par exemple, supposons un champ tableau *TelNos_Array*, qui est un tableau de chaînes composé de six éléments. Les champs persistants suivants créés pour le composant table *Customer* représente le champ *TelNos_Array* et ses six éléments :

```
CustomerTelNos_Array: TArrayField;
CustomerTelNos_Array0: TStringField;
CustomerTelNos_Array1: TStringField;
CustomerTelNos_Array2: TStringField;
CustomerTelNos_Array3: TStringField;
CustomerTelNos_Array4: TStringField;
CustomerTelNos_Array5: TStringField;
```

A partir de ces champs persistants, le code suivant utilise un champ persistant pour affecter une valeur d'élément de tableau à une zone de saisie nommée *TelEdit*.

```
TelEdit.Text := CustomerTelNos_Array0.AsString;
```

Utilisation de la propriété FieldValues d'un champ tableau

Vous pouvez accéder à la valeur d'un champ enfant avec la propriété *FieldValues* du champ tableau. *FieldValues* accepte et renvoie un *Variant* ; elle peut dont traiter et convertir des champs enfants de n'importe quel type. Par exemple,

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array')).FieldValues[1];
```

FieldValues étant la propriété par défaut de *TArrayField*, vous pouvez aussi écrire le code suivant :

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array'))[1];
```

Utilisation de la propriété Fields d'un champ tableau

TArrayField possède une propriété *Fields* qui vous permet d'accéder aux différents sous-champs. Cette possibilité est illustrée dans le code suivant, dans lequel un champ tableau (*OrderDates*) est utilisé pour remplir une boîte liste avec tous les éléments de tableau n'ayant pas NULL pour valeur :

```
for I := 0 to OrderDates.Size - 1 do
begin
  if not OrderDates.Fields[I].IsNull then
    OrderDateListBox.Items.Add(OrderDates[I]);
end;
```

Utilisation des champs ensemble de données

Les champs ensemble de données permettent d'accéder aux données stockées dans un ensemble de données imbriqué. La propriété NestedDataSet contient des références à l'ensemble de données imbriqué. Les données contenues dans l'ensemble de données imbriqué sont accessibles par le biais des objets champ de cet ensemble de données.

Affichage des champs ensemble de données

Les contrôles TDBGrid permettent l'affichage des données stockées dans les champs ensemble de données. Dans un contrôle *TDBGrid*, un champ ensemble de données est désigné dans chaque cellule de la colonne de l'ensemble de données avec la chaîne "(DataSet)" et, à l'exécution, avec un bouton points de suspension sur la droite. Le fait de cliquer sur ce bouton appelle une nouvelle fiche dont la grille affiche l'ensemble de données associé au champ ensemble de données de l'enregistrement en cours. Cette fiche peut aussi être appelée par programmation avec la méthode *ShowPopupEditor*. Par exemple, si la septième colonne de la grille représente un champ ensemble de données, le code suivant affichera l'ensemble de données associé à ce champ pour l'enregistrement en cours.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

Accès aux données d'un ensemble de données imbriqué

Normalement un champ ensemble de données n'est pas directement lié à un contrôle orienté données. En fait, comme un champ ensemble de données imbriqué n'est rien d'autre qu'un ensemble de données, vous utilisez un descendant de *TDataSet* pour accéder à ses données. Le type d'ensemble de données utilisé est déterminé par l'ensemble de données parent (celui possédant le champ ensemble de données). Par exemple, un ensemble de données BDE utilise *TNestedTable* pour représenter les données des champs ensemble de données, alors que les ensembles de données client utilisent d'autres ensembles de données client.

Pour accéder aux données d'un champ ensemble de données :

1. Créez un objet *TDataSetField* persistant en appelant l'éditeur de champ de l'ensemble de données parent.
2. Créez un ensemble de données représentant les valeurs de ce champ ensemble de données. Il doit avoir un type compatible avec l'ensemble de données parent.
3. Affectez à la propriété *DataSetField* de l'ensemble de données créé à l'étape 2 le champ ensemble de données persistant créé à l'étape 1.

Si le champ ensemble de données imbriqué de l'enregistrement en cours possède une valeur, le composant ensemble de données détail contient des enregistrements avec les données imbriquées ; sinon, l'ensemble de données détail est vide.

Avant d'insérer des enregistrements dans un ensemble de données imbriqué, vous devez avoir posté l'enregistrement correspondant dans la table maître, s'il vient juste d'être inséré. Si l'enregistrement inséré n'a pas été posté, il sera automatiquement posté avant que ne soit posté l'ensemble de données imbriqué.

Utilisation de champs de référence

Les champs de référence stockent un pointeur ou une référence vers un autre objet ADT. Cet objet ADT est un enregistrement unique d'une autre table objet. Les champs de référence font toujours référence à un enregistrement unique d'un ensemble de données (table objet). Les données contenues dans l'objet référencé sont renvoyées dans un ensemble de données imbriqué mais sont aussi accessibles via la propriété *Fields* sur le composant *TReferenceField*.

Affichage des champs de référence

Dans un contrôle *TDBGrid*, un champ de référence est désigné dans chaque cellule de la colonne de l'ensemble de données avec (Reference) et, à l'exécution, avec un bouton à points de suspension sur la droite. A l'exécution, le fait de cliquer sur ce bouton appelle une nouvelle fiche dont la grille affiche l'objet associé au champ de référence de l'enregistrement en cours.

Cette fiche peut aussi être appelée par programmation avec la méthode *ShowPopupEditor* de la grille DB. Par exemple, si la septième colonne de la grille

représente un champ de référence, le code suivant affiche l'objet associé à ce champ pour l'enregistrement en cours.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

Accès aux données d'un champ de référence

Vous pouvez accéder aux données d'un champ de référence de la même façon que vous accédez à un ensemble de données imbriqué :

1. Créez un objet *TDataSetField* persistant en appelant l'éditeur de champ de l'ensemble de données parent.
2. Créez un ensemble de données représentant la valeur de ce champ ensemble de données.
3. Affectez à la propriété *DataSetField* de l'ensemble de données créé à l'étape 2 le champ ensemble de données persistant créé à l'étape 1.

Si la référence est attribuée, l'ensemble de données de référence contient un enregistrement unique, avec les données référencées. Si la référence vaut null, l'ensemble de données de référence est vide.

Vous pouvez également utiliser la propriété Fields du champ de référence pour accéder aux données d'un champ de référence. Par exemple, les lignes suivantes sont équivalentes et affectent des données du champ de référence *CustomerRefCity* à une zone de saisie appelée *CityEdit* :

```
CityEdit.Text := CustomerRefCity.Fields[1].AsString;
CityEdit.Text := CustomerRefCity.NestedDataSet.Fields[1].AsString;
```

Lorsque les données d'un champ référence sont modifiées, c'est en fait les données référencées qui sont modifiées.

Pour affecter une valeur à un champ référence, vous devez d'abord utiliser une instruction SELECT pour sélectionner la référence dans la table et ensuite lui attribuer la valeur. Par exemple :

```
var
  AddressQuery: TQuery;
  CustomerAddressRef: TReferenceField;
begin
  AddressQuery.SQL.Text := 'SELECT REF(A) FROM AddressTable A WHERE A.City = ''San
  Francisco''';
  AddressQuery.Open;
  CustomerAddressRef.Assign(AddressQuery.Fields[0]);
end;
```

Chapitre

20

Utilisation du moteur de bases de données Borland

Le moteur de bases de données Borland (BDE) est un mécanisme d'accès aux données pouvant être partagé entre plusieurs applications. Le BDE définit une puissante bibliothèque d'appels API qui peuvent créer, restructurer, mettre à jour, interroger ou manipuler des serveurs de bases de données locaux ou distants. Le BDE fournit une interface uniforme permettant d'accéder à une grande variété de serveurs, en utilisant des pilotes pour se connecter aux différentes bases de données. Selon votre version de Delphi, vous pouvez utiliser les pilotes pour base de données locale (Paradox, dBASE, FoxPro et Access), les pilotes SQL Links pour base de données distante telles que InterBase, Oracle, Sybase, Informix, Microsoft SQL server ou DB2, et un adaptateur ODBC qui vous permet de fournir vos propres pilotes ODBC.

Durant son déploiement, vous devez inclure le BDE avec votre application. Bien que cela augmente la taille de l'application et la complexité du déploiement, le BDE peut être partagé avec d'autres applications BDE, et il fournit un large support pour la manipulation des bases de données. Même s'il est possible d'utiliser l'API du BDE directement dans votre application, les composants de la page BDE de la palette des composants regroupent toutes ces fonctionnalités.

Remarque Pour plus d'informations sur l'API du BDE, voir son fichier d'aide, BDE32.hlp, qui se trouve dans le répertoire où vous vous avez installé le moteur de bases de données Borland.

Architecture BDE

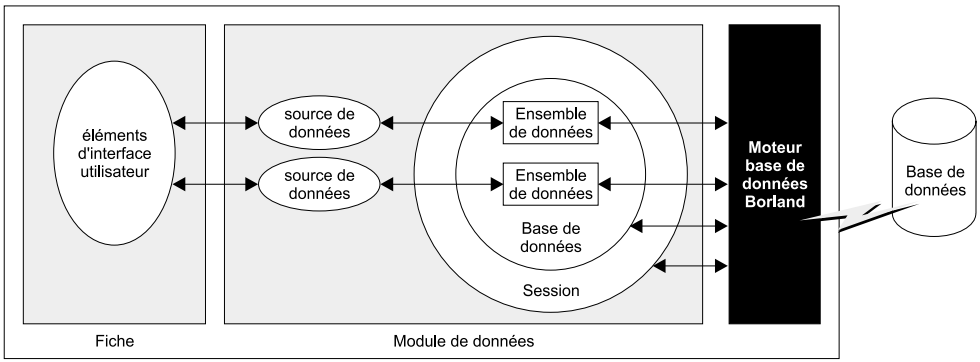
Avec le BDE, votre application utilise une variante de l'architecture générale de base de données décrite dans "Architecture des bases de données" à la page 14-6. En plus des éléments de l'interface utilisateur, de la source de données et des

ensembles de données communs à toutes les applications de bases de données Delphi, une application BDE peut inclure :

- Un ou plusieurs composants pour contrôler les transactions et gérer les connexions aux bases de données.
- Un ou plusieurs composants de session pour isoler les opérations d'accès aux données telles que les connexions aux bases de données, et pour gérer des groupes de bases de données.

Les relations entre les composants d'une application BDE sont illustrées dans la figure suivante :

Figure 20.1 Composants d'une application BDE



Utilisation d'ensembles de données BDE

Les ensembles de données BDE utilisent le moteur de bases de données Borland pour accéder aux données. Ils héritent des capacités de l'ensemble de données commun, décrites au chapitre 18, "Présentation des ensembles de données", en utilisant le BDE pour fournir l'implémentation. De plus, tous les ensembles de données BDE ajoutent des propriétés, des événements et des méthodes pour :

- Association d'un ensemble de données avec les connexions de bases de données et de session.
- Mise en cache des BLOBS.
- Obtention d'un handle BDE.

Il existe trois ensembles de données BDE :

- *TTable*, un ensemble de données de type table qui représente toutes les lignes et colonnes d'une table de base de données. Voir "Utilisation d'ensembles de données de type table" à la page 18-29 pour la description des caractéristiques communes aux ensembles de données de type table. Voir "Utilisation de TTable" à la page 20-5 pour la description des caractéristiques spécifiques à *TTable*.
- *TQuery*, un ensemble de données de type requête qui encapsule une instruction SQL et permet aux applications d'accéder aux enregistrements trouvés, s'il y en a. Voir "Utilisation d'ensembles de données de type requête"

à la page 18-49 pour la description des caractéristiques communes aux ensembles de données de type requête. Voir "Utilisation de TQuery" à la page 20-9 pour la description des caractéristiques spécifiques à *TQuery*.

- *TStoredProc*, un ensemble de données de type procédure stockée qui exécute une procédure stockée définie sur un serveur de base de données. Voir "Utilisation d'ensembles de données de type procédure stockée" à la page 18-58 pour la description des caractéristiques communes aux ensembles de données de type procédure stockée. Voir "Utilisation de TStoredProc" à la page 20-13 pour la description des caractéristiques spécifiques à *TStoredProc*.

Remarque En plus des trois types d'ensemble de données BDE, il existe un ensemble de données BDE client (*TBDEClientDataSet*) qui peut être utilisé pour les mises à jour en mémoire cache . Pour plus d'informations sur les mises à jour en mémoire cache, voir "Utilisation d'un ensemble de données client pour mettre en cache les mises à jour" à la page 23-18.

Association d'un ensemble de données avec les connexions de bases de données et de session

Pour qu'un ensemble de données BDE puisse récupérer des données sur un serveur, il doit utiliser à la fois une base de données et une session.

- Les bases de données représentent les connexions aux serveurs spécifiques de bases de données. La base de données identifie un pilote BDE, un serveur particulier de base de données utilisant ce pilote et un jeu de paramètres de connexion pour accéder à ce serveur. Chaque base de données est représentée par un composant *TDatabase*. Vous pouvez soit associer vos ensembles de données avec un composant *TDatabase* inséré dans un formulaire ou un module de données, soit identifier simplement le serveur de base de données par son nom et laisser Delphi générer un composant base de données implicite à votre place. Il est toutefois recommandé d'utiliser un composant *TDatabase* explicite dans la plupart des applications, parce qu'il vous donne un plus grand contrôle sur la façon d'établir la connexion, y compris sur le processus de connexion, et vous permet de créer et d'utiliser des transactions.

  Pour associer un ensemble de données BDE à une base de données, utilisez la propriété*DatabaseName*. DatabaseName est une chaîne qui contient des informations différentes selon que vous utilisez ou non un composant explicite de base de données et, si ce n'est pas le cas, le type de base de données utilisé :

  - Si vous utilisez un composant *TDatabase* explicite, *DatabaseName* est la valeur de la propriété *DatabaseName* du composant base de données.
  - Si vous désirez utiliser un composant base de données implicite et si la base de données possède un alias BDE, vous pouvez spécifier celui-ci comme valeur pour *DatabaseName*. Un alias BDE représente une base de données et ses informations de configuration. Les informations de configuration associées avec les alias diffèrent avec le type de la base de données (Oracle, Sybase, InterBase, Paradox, dBASE, etc.). Servez-vous de l'outil d'administration BDE ou de l'explorateur SQL pour créer et gérer les alias BDE.

  - Si vous désirez utiliser un composant base de données implicite pour une base de données Paradox ou dBASE, vous pouvez simplement spécifier avec *DatabaseName* le répertoire où se trouvent les tables de la base.

- Unesession fournit la gestion globale d'un groupe de connexions aux bases de données dans une application. Quand vous ajoutez des ensembles de données BDE dans votre application, celle-ci contient automatiquement un composant session nommé *Session*. Quand vous ajoutez des bases de données et des composants ensemble de données à votre application, ils sont automatiquement associés à cette session par défaut. Ce composant gère aussi l'accès aux fichiers Paradox protégés par mot de passe, et spécifie les répertoires pouvant partager des fichiers Paradox sur un réseau. Vous pouvez gérer les connexions aux bases de données et accéder aux fichiers Paradox en utilisant les propriétés, les événements et les méthodes de la session.

  Vous pouvez vous servir de la session par défaut pour gérer toutes les connexions aux bases de données dans votre application. Vous pouvez aussi ajouter des sessions supplémentaires au moment de la conception, ou les créer dynamiquement lors de l'exécution pour gérer un sous-ensemble de connexions. Pour associer votre ensemble de données avec un composant session explicitement créé, utilisez la propriété *SessionName*. Si vous n'utilisez pas de composant explicite de session dans votre application, il n'est pas nécessaire de fournir une valeur à cette propriété. Que vous utilisiez la session par défaut ou spécifiez une session explicite avec la propriété *SessionName*, vous pouvez accéder à la session associée à un ensemble de données en lisant la propriété *DBSession*.

Remarque Si vous utilisez un composant session, la propriété *SessionName* d'un ensemble de données doit concorder avec la propriété *SessionName* du composant base de données auquel cet ensemble de données est associé.

Pour plus d'informations sur *TDatabase* et *TSession*, voir "Connexion aux bases de données avec TDatabase" à la page 20-14 et "Gestion des sessions de bases de données" à la page 20-18.

Mise en cache des BLOBS

Les ensembles de données BDE possèdent tous une propriété *CacheBlobs* qui détermine si les champs BLOB sont mis localement en cache par le BDE quand une application lit les enregistrements BLOB. Par défaut, *CacheBlobs* a pour valeur *True*, ce qui signifie que le BDE met en cache une copie locale des champs BLOB. Cette mise en cache améliore les performances de l'application en permettant au BDE de stocker des copies locales des BLOBS au lieu d'aller les chercher sans cesse sur le serveur à mesure que l'utilisateur parcourt les enregistrements.

Dans les applications et les environnements où les BOLBS sont fréquemment mis à jour ou remplacés, et dans lesquels une vue actualisée des données BLOB est plus importante que les performances de l'application, vous pouvez mettre *CacheBlobs* à *False* pour vous assurer que l'application voit toujours la dernière version du champ BLOB.

Obtention d'un handle BDE

Vous pouvez très bien utiliser les ensembles de données BDE sans jamais devoir faire d'appel API direct au moteur de bases de données Borland. Les ensembles de données BDE, en combinaison avec les composants base de données et session, comprennent une bonne partie des fonctionnalités du BDE. Cependant, si vous devez faire des appels API directs au BDE, vous aurez besoin de handles pour les ressources gérées par le BDE. De nombreuses API du BDE requièrent ces handles sous la forme de paramètres.

Tous les ensembles de données BDE comprennent trois propriétés en lecture seule pour accéder aux handles du BDE durant l'exécution :

- *Handle* est un handle vers le curseur BDE qui accède aux enregistrements de l'ensemble de données.
- *DBHandle* est un handle vers la base de données qui contient les tables sous-jacentes ou la procédure stockée.
- *DBLocale* est un handle vers le pilote du langage BDE pour l'ensemble de données. La locale contrôle l'ordre de tri et le jeu de caractères utilisé pour la chaîne de données.

Ces propriétés sont automatiquement assignées à un ensemble de données quand il est connecté à un serveur de base de données à travers le BDE. Pour plus d'informations sur l'API du BDE, voir le fichier d'aide en ligne, BDE32.HLP.

Utilisation de TTable

TTable encapsule toute la structure et les données d'une table de base de données sous-jacente. Elle implémente toutes les fonctionnalités de base introduites par *TDataSet*, ainsi que toutes les caractéristiques spéciales typiques des ensembles de données de type table. Avant d'étudier les possibilités uniques offertes par *TTable*, vous devriez vous familiariser avec les caractéristiques communes des bases de données décrites dans "Présentation des ensembles de données" à la page 18-1, y compris la section sur les ensembles de données de type table qui commence page 18-29.

Comme *TTable* est un ensemble de données BDE, elle doit êtreassociée avec une base de données et une session. "Association d'un ensemble de données avec les connexions de bases de données et de session" à la page 20-3 décrit la façon de former ces associations. Une fois l'ensemble de données associé avec une base de données et une session, vous pouvez le lier à une table particulière en définissant la propriété *TableName* et, si vous utilisez une table Paradox, dBASE, FoxPro ou ASCII délimité par des virgules, la propriété *TableType*.

Remarque La table doit être fermée quand vous modifiez son association à une base de données, une session ou à une autre table, ou quand vous définissez la propriété *TableType*. Cependant, avant de fermer la table pour modifier ces propriétés, validez ou annulez toute modification en cours. Si les mises à jour en mémoire cache sont activées, appelez la méthode *ApplyUpdates* pour écrire les modifications validées dans la base de données.

Les composants *TTable* offrent un support unique aux tables des bases de données locales (Paradox, dBASE, FoxPro, ASCII délimité par des virgules). De plus, les composants *TTable* peuvent bénéficier du support du BDE pour les opérations groupées (opérations consistant à ajouter, mettre à jour, supprimer ou copier des groupes entiers d'enregistrements). Ce support est décrit dans "Importation des données d'une autre table" à la page 20-8.

Spécification du type d'une table locale

Si une application accède à une table Paradox, dBASE, FoxPro, ou ASCII délimité par des virgules, le BDE utilise la propriété *TableType* pour déterminer le type de la table (sa structure présumée). *TableType* n'est pas utilisée quand *TTable* représente une base de données SQL sur un serveur de base de données.

Par défaut, *TableType* est défini sur *ttDefault*. Dans ce cas, le BDE détermine le type de la table d'après l'extension de son nom de fichier. Le tableau suivant résume les extensions de fichier reconnues par le BDE et les types présumés correspondants :

Tableau 20.1 Types de tables reconnues par le BDE d'après les extensions

Extension	Type de table
Pas d'extension	Paradox
.DB	Paradox
.DBF	dBASE
.TXT	Texte ASCII

Si votre table locale Paradox, dBASE ou ASCII utilise les extensions décrites dans le tableau précédent, vous pouvez laisser *TableType* sur *ttDefault*. Sinon, votre application doit définir *TableType* pour indiquer le type réel de la table. Le tableau suivant indique les valeurs que vous pouvez assigner à *TableType* :

Tableau 20.2 Valeurs de TableType

Valeur	Type de table
ttDefault	Type de table déterminé automatiquement par le BDE
ttParadox	Paradox
ttDBase	dBASE
ttFoxPro	FoxPro
ttASCII	Texte ASCII délimité par des virgules

Contrôle d'accès en lecture/écriture aux tables locales

Comme tout ensemble de données de type table, *TTable* vous permet de contrôler l'accès en lecture et en écriture par votre application grâce à la propriété *ReadOnly*.

De plus, pour les tables Paradox, dBASE et FoxPro, *TTable* peut vous permettre de contrôler l'accès aux tables en lecture et en écriture par d'autres applications. La propriété *Exclusive* détermine si votre application obtient l'accès exclusif en

lecture/écriture à une base Paradox, dBASE ou FoxPro. Pour l'obtenir, mettez la propriété *Exclusive* du composant table à *True* avant d'ouvrir la table. Si vous réussissez à ouvrir la table avec un accès exclusif, les autres applications ne peuvent plus ni lire ni écrire de données dans la table. La requête d'accès exclusif ne peut être honorée que si la table n'est pas déjà en utilisation au moment où vous l'ouvrez.

Les instructions suivantes ouvrent une table en accès exclusif :

```
CustomersTable.Exclusive := True; {Etablit la requête de verrouillage exclusif}
CustomersTable.Active := True; {Ouvre la table}
```

Remarque Vous pouvez essayer d'appliquer *Exclusive* aux tables SQL, mais certains serveurs ne supportent pas le verrouillage exclusif au niveau des tables. D'autres peuvent admettre le verrouillage exclusif, mais permettre tout de même à d'autres applications de lire des données dans la table. Pour plus d'informations sur le verrouillage exclusif de tables de bases de données, voir la documentation de votre serveur.

Spécification d'un fichier d'index dBASE

Sur la plupart des serveurs, pour spécifier un index, vous utilisez les méthodes communes à tous les ensembles de données de type table. Ces méthodes sont décrites dans "Tri des enregistrements avec des index" à la page 18-30.

Cependant, pour les tables dBASE utilisant des fichiers d'index autres que le fichier d'index MDX d'exploitation ou des index dBASE III PLUS (*.NDX), utilisez les propriétés *IndexFiles et IndexName.* Assignez à la propriété *IndexFiles* le nom d'un fichier d'index autre que le fichier MDX d'exploitation ou listez les fichiers .NDX. Spécifiez ensuite un index dans la propriété *IndexName* pour qu'il trie effectivement l'ensemble de données.

Lors de la conception, cliquez sur le bouton point de suspension dans la valeur de la propriété *IndexFiles*, dans l'inspecteur d'objets, pour invoquer l'éditeur de fichiers d'index. Pour ajouter un fichier d'index .MDX ou .NDX, cliquez sur le bouton Ajouter dans la boîte de dialogue Fichiers d'index et sélectionnez le fichier dans la boîte de dialogue d'ouverture. Répétez ce processus pour chaque fichier d'index .MDX ou .NDX. Cliquez sur le bouton OK dans la boîte de dialogue Fichiers d'index après avoir ajouté tous les index voulus.

Cette même opération peut être exécutée par programmation lors de l'exécution. Pour cela, accédez à la propriété *IndexFiles* en utilisant les propriétés et méthodes des listes de chaînes. Quand vous ajoutez un nouvel ensemble d'index, appelez d'abord la méthode *Clear* de la propriété *IndexFiles* de la table pour supprimer toute entrée existante. Appelez la méthode *Add* pour ajouter chaque fichier d'index .MDX ou .NDX :

```
with Table2.IndexFiles do begin
  Clear;
  Add('Bystate.ndx');
  Add('Byzip.ndx');
  Add('Fullname.ndx');
  Add('St_name.ndx');
end;
```

Après avoir ajouté des fichiers d'index .MDX ou .NDX, les noms des index individuels du fichier d'index sont accessibles et modifiables par la propriété *IndexName*. Les repères d'index sont aussi listés en utilisant la méthode *GetIndexNames* et en inspectant les définitions d'index à travers les objets *TIndexDef* dans la propriété *IndexDefs*. Les fichiers .NDX correctement listés sont automatiquement mis à jour lorsque des données sont ajoutées, modifiées ou supprimées dans la table (qu'un index donné soit ou non utilisé dans la propriété *IndexName*).

Dans l'exemple ci-dessous, le fichier d'index ANIMALS.MDX est affecté à la propriété *IndexFiles* du composant table *AnimalsTable*, et le repère d'index "NAME" est affecté à sa propriété *IndexName* :

```
AnimalsTable.IndexFiles.Add('ANIMALS.MDX');
AnimalsTable.IndexName := 'NAME';
```

Une fois le fichier d'index spécifié, les index MDX ou NDX fonctionnent comme n'importe quel autre index. Spécifier un nom d'index trie les données de la table et rend l'index disponible pour les recherches, les portées et (pour les index MDX) les liaisons maître-détail. Voir "Utilisation d'ensembles de données de type table" à la page 18-29 pour plus de détail sur l'utilisation d'index.

En utilisant des index .NDX dBASE III PLUS avec les composants *TTable*, deux considérations particulières sont à prendre en compte. La première est que les fichiers .NDX ne peuvent pas être utilisés comme base pour les liaisons maître-détail. La seconde est que lors de l'activation d'un index .NDX avec la propriété *IndexName*, vous devez inclure l'extension .NDX dans le nom de l'index dans la valeur de la propriété :

```
with Table1 do begin
  IndexName := 'ByState.NDX';
  FindKey(['CA']);
end;
```

Renommer une table locale

Pour renommer une table Paradox ou dBASE au moment de la conception, faites un clic droit sur le composant table et sélectionnez l'option adéquate dans le menu contextuel.

Pour renommer une table Paradox ou dBASE à l'exécution, appelez la méthode *RenameTable* de la table. Par exemple, l'instruction suivante renomme la table Customer en CustInfo :

```
Customer.RenameTable('CustInfo');
```

Importation des données d'une autre table

Vous pouvez utiliser la méthode *BatchMove* pour importer les données d'une autre table. *BatchMove* peut :

- Copier des enregistrements d'une autre table dans la table en cours.
- Mettre à jour les enregistrements de la table en cours qui apparaissent dans une autre table.

- Ajouter des enregistrements d'une autre table à la fin de la table en cours.
- Supprimer dans la table en cours les enregistrements qui apparaissent dans une autre table.

BatchMove prend deux paramètres : le nom de la table depuis laquelle importer les données, et une spécification de mode qui détermine quelle opération d'importation effectuer.Le tableau suivant indique les valeurs possibles pour la spécification de mode :

Tableau 20.3 Modes d'importation de BatchMove

Valeur	Signification
batAppend	Ajoute tous les enregistrements de la table source à la fin de la table en cours.
batAppendUpdate	Ajoute tous les enregistrements de la table source à la fin de la table en cours et met à jour les enregistrements existants dans cette table avec les enregistrements correspondants de la table source.
batCopy	Copie tous les enregistrements de la table source dans la table en cours.
batDelete	Supprime tous les enregistrements de la table en cours qui apparaissent aussi dans la table source.
batUpdate	Met à jour les enregistrements existants de la table en cours avec les enregistrements correspondants de la table source.

Par exemple, le code suivant met à jour tous les enregistrements de la table en cours avec les enregistrements de la table *Customer* qui possèdent les mêmes valeurs pour les champs de l'index en cours :

```
Table1.BatchMove('CUSTOMER.DB', batUpdate);
```

BatchMove renvoie le nombre d'enregistrements importés avec succès.

Attention L'importation d'enregistrements en utilisant le mode *batCopy* écrase les enregistrements existants. Pour préserver ces derniers, utilisez plutôt *batAppend*.

BatchMove n'effectue que certaines opérations groupées supportées par le BDE. D'autres fonctions sont disponibles par le composant *TBatchMove*. Si vous devez déplacer une grosse masse de données dans les tables, utilisez *TBatchMove* au lieu d'appeler la méthode de table *BatchMove*. Pour plus d'informations sur l'utilisation de *TBatchMove*, voir "Utilisation de TBatchMove" à la page 20-55.

Utilisation de TQuery

TQuery représente une unique instruction en DDL (Data Definition Language) ou DML (Data Manipulation Language), par exemple une commande SELECT, INSERT, DELETE, UPDATE, CREATE INDEX ou ALTER TABLE. Le langage utilisé dans les commandes est spécifique au serveur, mais se conforme généralement au standard SQL-92. *TQuery* implémente toutes les fonctionnalités de base introduites par *TDataSet*, ainsi que toutes les spécificités des ensembles de données de type requête. Avant d'étudier les caractéristiques particulières de *TQuery*, familiarisez-vous avec les fonctions communes des bases de données

décrites au chapitre 18, “Présentation des ensembles de données”, y compris la section sur les ensembles de données de type requête qui commence page 18-49.

Comme *TQuery* est un ensemble de données BDE, il doit habituellement être associé avec une base de données et une session (l'exception survient quand vous utilisez *TQuery* pour une requête hétérogène). Consultez “Association d'un ensemble de données avec les connexions de bases de données et de session” à la page 20-3 pour savoir comment former de telles associations. Vous spécifiez l'instruction SQL de la requête en définissant la propriété *SQL*.

Un composant*TQuery* peut accéder aux données dans :

- Les tables Paradox ou dBASE, en utilisant Local SQL, qui fait partie du BDE. Local SQL est un sous-ensemble de la spécification SQL-92. La plus grande partie de DML et suffisamment de syntaxe DDL sont supportés pour travailler avec ces types de tables. Consultez l'aide locale SQL, LOCALSQL.HLP, pour plus de détails sur la syntaxe SQL supportée.
- Les bases de données locales InterBase Server, à l'aide du moteur InterBase. Pour plus d'informations sur le support de syntaxe SQL-92 d'InterBase et sur le support de syntaxe étendue, voir la référence au langage InterBase.
- Les bases de données sur serveurs distants, comme Oracle, Sybase, MS-SQL Server, Informix, DB2, et InterBase. Vous devez installer le pilote SQL Link et un logiciel client spécifique(fourni par le vendeur) pour accéder au serveur de base de données. Toute syntaxe SQL supportée par ces serveurs est admise. Pour plus d'informations sur la syntaxe, les limitations et les extensions SQL, voir la documentation de votre serveur.

Création de requêtes hétérogènes

TQuery supporte les requêtes hétérogènes sur plusieurs types de serveur ou de table (par exemple, des données d'une table Oracle et d'une table Paradox). Quand vous exécutez une requête hétérogène, le BDE analyse et traite la requête en utilisant Local SQL. Pour cette raison, la syntaxe SQL spécifique au serveur et étendue n'est pas supportée.

Pour effectuer une requête hétérogène, suivez les étapes ci-dessous :

1. Définissez des alias BDE distincts pour chaque base de données concernée dans la requête, en utilisant l'outil d'administration BDE ou l'explorateur SQL.
2. Laissez vide la propriété *DatabaseName* de *TQuery* ; les noms des bases de données seront spécifiés dans l'instruction SQL.
3. Dans la propriété SQL, spécifiez l'instruction SQL à exécuter. Faites-y précéder le nom de chaque table par l'alias BDE de la base de données de cette table, encadré par des caractères deux-points. La référence entière est elle-même encadrée de guillemets.
4. Définissez les paramètres de la requête dans la propriété *Params*.
5. Appelez *Prepare* pour préparer la requête avant sa première exécution.
6. Appelez *Open* ou *ExecSQL* selon le type de requête que vous exécutez.

Par exemple, vous définissez un alias nommé *Oracle1* pour une base de données Oracle comprenant une table CUSTOMER, et un alias nommé *Sybase1* pour une base de données Sybase comprenant une table ORDERS. Une requête simple sur ces deux tables pourrait s'écrire :

```
SELECT Customer.CustNo, Orders.OrderNo
FROM ":Oracle1:CUSTOMER"
  JOIN ":Sybase1:ORDERS"
    ON (Customer.CustNo = Orders.CustNo)
WHERE (Customer.CustNo = 1503)
```

Dans une requête hétérogène, au lieu de spécifier la base de données à l'aide d'un alias, vous pouvez utiliser un composant *TDatabase*. Configurez *TDatabase* afin qu'il pointe sur la base de données, affectez à la propriété *TDatabase.DatabaseName* une valeur arbitraire mais unique, et utilisez ensuite cette valeur dans l'instruction SQL au lieu d'un nom d'alias BDE.

Obtention d'un ensemble de résultats modifiable

Pour obtenir un ensemble de résultats que l'utilisateur puisse éditer dans des contrôles orientés données, définissez la propriété *RequestLive* d'un composant requête à *True*. Définir *RequestLive* à *True* ne garantit pas un ensemble de résultats dynamique, mais le BDE essaie d'honorer la requête chaque fois que possible. Il existe quelques restrictions sur les requêtes d'ensemble de résultats dynamiques, selon que la requête utilise l'analyseur SQL local ou l'analyseur SQL d'un serveur.

- Les requêtes dans lesquelles les noms des tables sont précédés d'un alias de base de données BDE (comme les requêtes hétérogènes) et les requêtes exécutées sur Paradox ou dBASE sont analysées par le BDE en utilisant le SQL local. Quand une requête utilise l'analyseur SQL local, le BDE offre une gestion étendue des ensembles de résultats dynamiques et modifiables, aussi bien dans les requêtes monotables que multitables. Lors de l'utilisation du SQL local, un ensemble de résultats dynamique d'une requête sur une vue ou une table unique est renvoyé si la requête ne contient aucun des éléments suivants :
  - DISTINCT dans la clause SELECT
  - Jointures (interne, externe ou UNION)
  - Fonctions d'agrégat avec ou sans clauses GROUP BY ou HAVING
  - Vues ou tables de base qui ne peuvent pas être mises à jour
  - Sous-requêtes
  - Clauses ORDER BY non basées sur un index
- Les requêtes sur un serveur de base de données distant sont analysées par le serveur. Si la propriété *RequestLive* a pour valeur *True*, l'instruction SQL doit rester dans les standards Local SQL et se conformer de plus à toute restriction imposée par le serveur, car le BDE en a besoin pour communiquer les modifications de données à la table. Un ensemble de résultats dynamique d'une requête sur une vue ou une table unique est renvoyé si la requête ne contient aucun des éléments suivants :
  - Clause DISTINCT dans l'instruction SELECT

- Fonctions d'agrégat avec ou sans clauses GROUP BY ou HAVING
- Références à plusieurs tables de base ou vues modifiables (jointures)
- Sous-requêtes qui référencent la table dans la clause FROM ou d'autres tables

Si une application demande et reçoit un ensemble de résultats dynamique, la propriété *CanModify* du composant requête est définie à *True*. Même si la requête renvoie un ensemble de résultats dynamique, il n'est pas sûr que vous puissiez mettre à jour l'ensemble de résultats directement s'il contient des champs liés ou si vous changez d'index avant de tenter une mise à jour. Si ces conditions se présentent, vous devez considérer l'ensemble de résultats comme étant en lecture seule, et le mettre à jour en conséquence.

Si une application demande un ensemble de résultats dynamique, mais que la syntaxe de l'instruction SELECT ne le permette pas, le BDE retourne :

- Soit un ensemble de résultats en lecture seule pour les requêtes effectuées sur Paradox ou dBASE.
- Soit un code d'erreur pour les requêtes SQL effectuées sur un serveur distant.

Mise à jour des ensembles de résultats en lecture seule

Les applications peuvent mettre à jour les données renvoyées dans un ensemble de résultats en lecture si elles utilisent les mises à jour en mémoire cache.

Si vous utilisez un ensemble de données client pour mettre en mémoire cache les mises à jour, celui-ci ou son fournisseur associé peut générer automatiquement le code SQL pour appliquer les mises à jour, à moins que la requête ne représente des tables multiples. Dans ce dernier cas, vous devez indiquer comment appliquer les mises à jour :

- Si toutes les mises à jour sont appliquées à une table de base de données unique, vous pouvez indiquer la table sous-jacente à mettre à jour dans un gestionnaire d'événement *OnGetTableName*.
- Si vous avez besoin de davantage de contrôle sur l'application des mises à jour, vous pouvez associer la requête avec un objet de mise à jour (*TUpdateSQL*). Un fournisseur utilise automatiquement cet objet de mise à jour pour appliquer les mises à jour :
  1. Associez l'objet de mise à jour à la requête en définissant la propriété *UpdateObject* de la requête à l'objet *TUpdateSQL* que vous utilisez.
  2. Affectez aux propriétés *ModifySQL*, *InsertSQL* et *DeleteSQL* de l'objet de mise à jour les instructions SQL qui effectuent les mises à jour appropriées aux données de votre requête.

Si vous utilisez le BDE pour mettre en mémoire cache les mises à jour, vous devez utiliser un objet de mise à jour.

Remarque Pour plus d'informations sur les objets de mise à jour, voir "Utilisation d'objets mise à jour pour mettre à jour un ensemble de données" à la page 20-45.

Utilisation de TStoredProc

TStoredProc représente une procédure stockée. Elle implémente toutes les fonctionnalités introduites par *TDataSet*, ainsi que la plupart des fonctionnalités spéciales typiques des ensembles de données de type procédure stockée. Avant d'étudier les caractéristiques particulières de *TStoredProc*, familiarisez-vous avec les fonctions communes des bases de données décrites au chapitre 18, "Présentation des ensembles de données", y compris la section sur les ensembles de données de type procédure stockée qui commence en page 18-58.

Puisque *TStoredProc* est un ensemble de données BDE, il doit être associé à une base de données et une session. Voir "Association d'un ensemble de données avec les connexions de bases de données et de session" à la page 20-3 pour savoir comment former de telles associations. Une fois l'ensemble de données associé à une base de données et une session, vous pouvez le lier à une procédure stockée particulière en définissant la propriété *StoredProcName*.

TStoredProc diffère des autres ensembles de données de type procédure stockée par les points suivants :

- Il vous donne un plus grand contrôle sur la façon de lier les paramètres .
- Il fournit la gestion des procédures stockées surchargées Oracle.

Liaison des paramètres

Quand vous préparez et exécutez une procédure stockée, ses paramètres d'entrée sont automatiquement liés aux paramètres sur le serveur.

TStoredProc vous permet d'utiliser la propriété *ParamBindMode* pour spécifier comment les paramètres doivent être liés à ceux du serveur. Par défaut, *ParamBindMode* est défini à *pbByName*, ce qui signifie que les paramètres du composant procédure stockée correspondent à ceux du serveur par leur nom. C'est la méthode la plus simple pour lier les paramètres.

Certains serveurs supportent aussi la liaison des paramètres par valeur ordinale, c'est-à-dire l'ordre dans lequel ils apparaissent dans la procédure stockée. Dans ce cas, l'ordre dans lequel vous spécifiez les paramètres dans l'éditeur de collection de paramètres est important. Le premier paramètre spécifié correspond au premier paramètre d'entrée sur le serveur, et ainsi de suite. Si votre serveur supporte ce type de liaison de paramètre, définissez *ParamBindMode* à *pbByNumber*.

Astuce Si vous voulez définir *ParamBindMode* à *pbByNumber*, vous devez spécifier des paramètres de type correct dans l'ordre correct. Vous pouvez consulter le code source d'une procédure stockée d'un serveur dans l'explorateur SQL pour déterminer l'ordre correct et le type des paramètres à indiquer.

Manipulation des procédures stockées redéfinies d'Oracle

Les serveurs Oracle permettent la redéfinition de procédures stockées ; les procédures redéfinies sont des procédures différentes avec le même nom. La propriété *Overload* du composant procédure stockée permet à l'application de spécifier la procédure à exécuter.

Si *Overload* a pour valeur zéro (valeur par défaut), il n'y a pas de redéfinition. Si *Overload* a pour valeur 1, le composant procédure stockée exécute la première procédure stockée qu'il trouve sur le serveur Oracle ayant le nom redéfini ; si *Overload* a pour valeur 2, il exécute la seconde, et ainsi de suite.

Remarque Les procédures stockées redéfinies peuvent prendre des paramètres d'entrée et de sortie différents. Voir la documentation de votre serveur Oracle pour plus d'informations.

Connexion aux bases de données avec TDatabase

Quand une application Delphi utilise le moteur de bases de données Borland (BDE) pour se connecter à une base de données, cette connexion est encapsulée par un composant *TDatabase*. Un composant base de données représente la connexion à une base de données unique dans le contexte d'une session BDE.

TDatabase effectue la plupart des mêmes tâches que les autres composants de connexion de base de données et partage avec eux de nombreuses propriétés méthodes et événements communs. Ces fonctionnalités sont décrites dans le chapitre 17, "Connexion aux bases de données".

En plus des propriétés, méthodes et événements communs, *TDatabase* introduit de nombreuses fonctionnalités spécifiques au BDE, décrites dans les rubriques suivantes .

Association d'un composant base de données à une session

Tous les composants base de données doivent être associés à une session BDE. Utilisez *SessionName* pour établir cette association. Quand vous créez un composant base de données au moment de la conception, la propriété *SessionName* est définie à "Default", ce qui signifie qu'il est associé au composant de session par défaut référencé par la variable globale *Session*.

Les applications BDE multithreads ou réentrantes peuvent nécessiter plusieurs sessions. Si vous devez utiliser des sessions multiples, ajoutez des composants *TSession* pour chaque session. Associez alors votre ensemble de données à un composant session en affectant à la propriété *SessionName* la valeur de la propriété *SessionName* du composant session.

Durant l'exécution, vous pouvez accéder au composant session avec lequel la base de données est associée en lisant la propriété *Session*. Si *SessionName* est vide ou a pour valeur "Default", la propriété *Session* référence l'instance *TSession* référencée par la variable globale *Session*. *Session* permet aux applications d'accéder aux propriétés, méthodes et événements d'un composant session parent d'un composant de base de données sans connaître le nom réel de la session.

Pour plus d'informations sur les sessions BDE, voir "Gestion des sessions de bases de données" à la page 20-18.

Si vous utilisez un composant base de données implicite, la session de ce composant base de données est celle spécifiée par la propriété *SessionName* de l'ensemble de données.

Interactions entre les composants base de données et session

En général, les propriétés de composants session fournissent des comportements globaux par défaut qui s'appliquent à tous les composants base de données créés à l'exécution. Par exemple, la propriété *KeepConnections* de la session de contrôle détermine si une connexion de base de données est maintenue même si ses ensembles de données associés sont fermés (cas par défaut), ou si la connexion est interrompue quand tous ses ensembles de données sont fermés. De même, l'événement par défaut *OnPassword* d'une session garantit que l'application affiche la boîte de dialogue standard de saisie de mot de passe si elle tente de s'attacher à une base de données sur un serveur nécessitant un mot de passe.

Les méthodes de session s'appliquent un peu différemment. Les méthodes *TSession* affectent tous les composants base de données, qu'ils soient créés explicitement ou instanciés implicitement par un ensemble de données. Par exemple, la méthode de session *DropConnections* ferme tous les ensembles de données appartenant aux composants base de données d'une session, puis interrompt toutes les connexions de bases de données, même si la propriété *KeepConnection* de composants base de données individuels a pour valeur *True*.

Les méthodes de composants de bases de données ne s'appliquent qu'aux ensembles de données associés à un composant base de données donné. Par exemple, supposons que le composant *Database1* est associé à la session par défaut. *Database1.CloseDataSets()* ne ferme que les ensembles de données associés à *Database1*. Les ensembles de données ouverts appartenant à d'autres composants de bases de données dans la session par défaut restent ouverts.

Identification de la base de données

AliasName et *DriverName* sont des propriétés mutuellement exclusives qui identifient le serveur de base de données auquel le composant *TDatabase* est connecté.

- *AliasName* spécifie le nom d'un alias BDE existant à utiliser pour le composant de base de données. L'alias apparaît ensuite dans les listes déroulantes pour les composants ensemble de données, afin que vous puissiez les lier à un composant base de données particulier. Si vous spécifiez *AliasName* pour un composant base de données, toute valeur déjà affectée à *DriverName* est effacée, parce qu'un nom de pilote fait toujours partie d'un alias BDE.

  Vous créer et éditez les alias BDE en utilisant l'explorateur de bases de données ou l'utilitaire d'administration BDE. Pour plus d'informations sur la création et la maintenance des alias BDE, voir la documentation en ligne de ces utilitaires.

- *DriverName*est le nom d'un pilote BDE. Un nom de pilote est un paramètre d'un alias BDE, mais vous pouvez spécifier un nom de pilote au lieu d'un alias quand vous créez un alias BDE local pour un composant base de données en utilisant la propriété *DatabaseName*. Si vous spécifiez *DriverName*,

toute valeur déjà affectée à *AliasName* est effacée pour éviter tout conflit potentiel entre le nom du pilote que vous spécifiez et le nom du pilote faisant partie de l'alias BDE identifié par *AliasName*.

DatabaseName vous permet de fournir votre propre nom pour une connexion de base de données. Ce nom s'ajoute alors à *AliasName* ou *DriverName*, et est local à votre application. *DatabaseName* peut être un alias BDE ou, pour les fichiers Paradox et dBASE, un chemin d'accès qualifié. Comme *AliasName*, *DatabaseName* apparaît ensuite dans les listes déroulantes pour les composants ensemble de données afin de vous permettre de les lier aux composants base de données.

Au moment de la conception, pour spécifier un alias BDE, affecter un pilote BDE ou créer un alias BDE local, double-cliquez sur un composant base de données pour appeler l'éditeur de propriétés de bases de données.

Vous pouvez entrer un *DatabaseName* dans la zone de saisie Nom de l'éditeur de propriétés. Vous pouvez entrer un nom d'alias BDE existant dans la boîte à options Nom d'alias pour la propriété *Alias*, ou choisir un alias existant dans la liste déroulante. La boîte à options Nom de pilote vous permet de saisir le nom d'un pilote BDE existant pour la propriété *DriverName*, mais vous pouvez aussi en choisir un dans la liste déroulante des pilotes existants.

Remarque L'éditeur de propriétés de bases de données vous permet aussi de visualiser et de définir les paramètres de connexion BDE, et de définir l'état des propriétés *LoginPrompt* et *KeepConnection*. Pour plus d'informations sur les paramètres de connexion, voir "Définition des paramètres d'alias BDE" ci-dessous. Pour plus d'informations sur *LoginPrompt*, voir "Contrôle de la connexion au serveur" à la page 17-4. Pour plus d'informations sur *KeepConnection*, voir "Ouverture d'une connexion avec TDatabase" à la page 20-17.

Définition des paramètres d'alias BDE

Au moment de la conception, vous pouvez créer ou éditer les paramètres de connexion de trois manières :

- Utilisez l'explorateur de bases de données ou l'utilitaire d'administration BDE pour créer ou modifier des alias BDE, y compris les paramètres. Pour plus d'informations sur ces utilitaires, voir les fichiers d'aide en ligne.
- Double-cliquez sur la propriété *Params* dans l'inspecteur d'objets pour appeler éditeur de liste de chaînes .
- Double-cliquez sur un composant base de données dans un module de données ou une fiche pour appeler l'éditeur de propriétés de bases de données.

Ces trois méthodes éditent la propriété *Params* du composant base de données. *Params* est une liste de chaînes contentant les paramètres de connexion de base de données pour l'alias BDE associé à un composant base de données. Certains paramètres de connexion typiques comprennent le chemin d'accès, le nom du serveur, la taille de cache du schéma, le pilote de langage et le mode de requête SQL.

Quand vous appelez l'éditeur de propriétés de bases de données pour la première fois, les paramètres de l'alias BDE ne sont pas visibles. Pour afficher les paramètres en cours, cliquez sur Défaut. Les paramètres en cours s'affichent dans la zone mémo Paramètres de connexion. Vous pouvez éditer les entrées existantes ou en ajouter de nouvelles. Pour effacer les paramètres existants, cliquez sur Effacer. Les modifications ne prennent effet que lorsque vous cliquez sur OK.

A l'exécution, l'application ne peut définir les paramètres d'alias qu'en modifiant directement la propriété *Params*. Pour plus d'informations sur les paramètres spécifiques à l'utilisation de pilotes SQL Links avec le BDE, voir le fichier d'aide en ligne SQL Links.

Ouverture d'une connexion avec TDatabase

Comme avec tous les composants de connexion de base de données, pour vous connecter à une base de données à l'aide de *TDatabase*, définissez la propriété *Connected* à *True* ou appelez la méthode *Open*. Ce processus est décrit dans "Connexion à un serveur de bases de données" à la page 17-3. Une fois la connexion de base de données établie, elle est maintenue aussi longtemps qu'il reste au moins un ensemble de données actif. Quand il n'en reste plus, la connexion est interrompue, à moins que la propriété *KeepConnection* du composant base de données soit à *True*.

Quand vous vous connectez à partir d'une application sur un serveur de base de données distant, l'application utilise le BDE et le pilote SQL Links Borland pour établir la connexion. Le BDE peut aussi communiquer avec un pilote ODBC fourni par vos soins. Vous devrez configurer le pilote ODBC ou SQL Links pour votre application avant d'effectuer la connexion. Les paramètres ODBC et SQL Links sont stockés dans la propriété *Params* d'un composant base de données. Pour plus d'informations sur les paramètres SQL Links, voir le guide de l'utilisateur en ligne *SQL Links*. Pour modifier la propriété *Params*, voir "Définition des paramètres d'alias BDE" à la page 20-16.

Manipulation des protocoles réseau

Durant la configuration du pilote ODBC ou SQL Links approprié, il peut être nécessaire de spécifier le protocole réseau utilisé par le serveur, tel que SPX/IPX ou TCP/IP, en fonction des options de configuration du pilote. Dans la plupart des cas, la configuration du protocole réseau est prise en charge par un logiciel de configuration client du serveur. Pour ODBC, il peut également être nécessaire de vérifier la configuration du pilote avec le gestionnaire de pilote ODBC.

L'établissement d'une connexion initiale entre client et serveur peut s'avérer problématique. La liste de dépannage suivante pourra vous aider si vous rencontrez des difficultés :

- La connexion côté client du serveur est-elle correctement configurée ?
- Toutes les DLL pour la connexion et les pilotes de bases de données sont-elles dans le chemin de recherche ?

- Si vous utilisez TCP/IP :
  - Le logiciel de communication TCP/IP est-il installé ? La bonne WINSOCK.DLL est-elle installée ?
  - L'adresse IP du serveur est-elle recensée dans le fichier HOSTS du client ?
  - DNS (Domain Name Services) est-il correctement configuré ?
  - Pouvez-vous effectuer un ping du serveur ?

Pour d'autres conseils de dépannage, voir le guide de l'utilisateur *SQL Links* en ligne et la documentation de votre serveur.

Utilisation d'ODBC

Une application peut utiliser les sources de données ODBC (par exemple, Btrieve). Une connexion par pilote ODBC requiert :

- Un pilote ODBC fourni par le vendeur.
- Microsoft ODBC Driver Manager.
- L'utilitaire d'administration BDE.

Pour configurer un alias BDE pour une connexion par pilote ODBC, utilisez l'utilitaire d'administration BDE. Pour plus d'informations, voir le fichier d'aide en ligne de l'utilitaire d'administration BDE.

Utilisation des composants base de données dans les modules de données

Vous pouvez placer sans risque des composants base de données dans des modules de données. Cependant, si vous mettez un module de données contenant un composant base de données dans le référentiel d'objets, et si vous voulez que d'autres utilisateurs puissent en hériter, vous devez définir la propriété *HandleShared* du composant base de données à *True* pour éviter tout conflit de domaine d'appellation global.

Gestion des sessions de bases de données

Les requêtes, curseurs, pilotes et connexions de bases de données d'une application BDE sont maintenues dans le contexte d'une ou plusieurs sessions BDE. Les sessions isolent un ensemble d'opérations d'accès à une base de données, comme les connexions de bases de données, sans qu'il soit nécessaire de lancer une autre instance de l'application.

Toutes les applications BDE comprennent automatiquement un composant session par défaut, nommé *Session*, qui encapsule la session BDE par défaut. Quand les composants base de données sont ajoutés à l'application, ils sont automatiquement associés à la session par défaut (notez que son *SessionName* est "Default"). La session par défaut fournit un contrôle global sur tous les composants base de données non associés à une autre session, qu'ils soient implicites (créés par la session à l'exécution quand vous ouvrez un ensemble de données non associé à un composant base de données que vous avez créé) ou persistant (créé explicitement par votre application). La session par défaut n'est

pas visible dans votre module de données ou votre fiche au moment de la conception, mais vous pouvez accéder par code à ses propriétés et à ses méthodes lors de l'exécution.

Pour utiliser la session par défaut, vous n'avez pas à écrire de code, à moins que votre application doive :

- Explicitement activer ou désactiver une session, en activant ou désactivant la capacité d'ouverture de base de données de la session.
- Modifier les propriétés de la session, par exemple spécifier les propriétés par défaut pour les composants base de données générés implicitement.
- Exécuter les méthodes d'une session, comme gérer les connexions aux bases de données (par exemple ouvrir et fermer des connexions de bases de données en réponse aux actions de l'utilisateur).
- Répondre aux événements de session, comme quand l'application essaie d'accéder à une table Paradox ou dBASE protégée par mot de passe.
- Définir des répertoires Paradox comme avec la propriété *NetFileDir* pour accéder aux tables Paradox sur un réseau et avec la propriété *PrivateDir* sur un disque dur local pour accélérer les performances.
- Gérer les alias BDE qui décrivent les configurations de connexion possibles pour les bases de données et les ensembles de données qui utilisent la session.

Que vous ajoutiez les composants base de données à une application au moment de la conception ou que vous les créiez dynamiquement à l'exécution, ils sont automatiquement associés à la session par défaut, à moins que vous ne les affectiez spécifiquement à une session différente. Si vous ouvrez un ensemble de données non associé à un composant base de données, Delphi va automatiquement :

- Créer un composant base de données pour cet ensemble de données à l'exécution.
- Associer le composant base de données à la session par défaut.
- Initialiser certaines propriétés-clés du composant base de données, basées sur celles de la session par défaut. Parmi les plus importantes de ces propriétés se trouve *KeepConnections*, qui détermine quand les connexions de bases de données sont maintenues ou interrompues par l'application.

La session par défaut procure un large ensemble de valeurs par défaut pouvant être utilisées par la plupart des applications. Il n'est nécessaire d'associer un composant base de données à une session explicitement nommée que si le composant exécute une requête simultanée sur une base de données déjà ouverte par la session par défaut. Dans ce cas, chaque requête concurrente doit fonctionner dans sa propre session. Les applications de bases de données multithreads requièrent aussi des sessions multiples, dans lesquelles chaque thread possède sa propre session.

Les applications peuvent créer des composants session supplémentaires selon leurs besoins. Les applications de bases de données BDE comprennent automatiquement un composant liste de session, nommé *Sessions*, que vous

pouvez utiliser pour gérer tous vos composants session. Pour plus d'informations sur la gestion de sessions multiples, voir "Gestion de sessions multiples" à la page 20-32.

Vous pouvez placer sans risque des composants session dans des modules de données. Cependant, si vous mettez un module de données contenant un ou plusieurs composants session dans le référentiel d'objets, assurez-vous que la propriété *AutoSessionName* a pour valeur *True* pour éviter tout conflit d'espace d'appellation quand les utilisateurs en héritent.

Activation d'une session

Active est une propriété booléenne qui détermine si les composants base de données et ensemble de données associés à une session sont ouverts. Vous pouvez utiliser cette propriété pour lire l'état en cours des connexions de bases de données et d'ensemble de données d'une session, ou changer cet état. Si *Active* a pour valeur *False* (valeur par défaut), toutes les bases de données et ensembles de données associés à la session sont fermés. Si *Active* a pour valeur *True*, les bases de données et ensembles de données sont ouverts.

Une session est activée dès qu'elle est créée, puis chaque fois que sa propriété *Active* est changée de *False* à *True* (par exemple, quand une base de données ou un ensemble de données associé à une session est ouverte alors qu'il n'y a aucun autre ensemble de données ou base de données ouvert). Mettre *Active* à *True* déclenche l'événement de session *OnStartup*, recense les répertoires Paradox avec le BDE, et recense la propriété *ConfigMode*, qui détermine quels alias BDE sont disponibles dans cette session. Vous pouvez écrire un gestionnaire d'événement *OnStartup* pour initialiser les propriétés *NetFileDir*, *PrivateDir* et *ConfigMode* avant qu'elles ne soient recensées avec le BDE, ou pour effectuer d'autres activités spécifiques de démarrage de session. Pour plus d'informations sur les propriétés *NetFileDir* et *PrivateDir*, voir "Spécification des répertoires Paradox" à la page 20-27. Pour plus d'informations sur ConfigMode, voir "Manipulation des alias BDE" à la page 20-28.

Une fois la session active, vous pouvez ouvrir ses connexions de bases de données en appelant la méthode *OpenDatabase*.

Pour les composants session placés dans un module de données ou une fiche, mettre *Active* à *False* quand il existe des bases de données ou des ensembles de données ouverts ferme ceux-ci. A l'exécution, fermer des bases de données ou des ensembles de données peut déclencher les événements associés.

Remarque Il est impossible de mettre *Active* à *False* pour la session par défaut au moment de la conception. Bien qu'il soit possible de fermer la session par défaut à l'exécution, ce n'est pas recommandé.

Vous pouvez aussi utiliser les méthodes *Open* et *Close* d'une session pour activer ou désactiver les sessions autres que la session par défaut à l'exécution. Par exemple, la simple ligne de code suivante ferme toutes les bases de données et ensembles de données ouverts pour une session :

```
Session1.Close;
```

Ce code met la propriété *Active* de Session1 à *False*. Quand la propriété *Active* d'une session a pour valeur *False*, tout essai par l'application d'ouvrir une base de données ou un ensemble de données remet *Active* à *True* et appelle le gestionnaire d'événement *OnStartup* de la session, s'il existe. Vous pouvez aussi coder explicitement la réactivation de la session à l'exécution. Le code suivant réactive *Session1* :

```
Session1.Open;
```

Remarque Si une session est active, vous pouvez aussi ouvrir et fermer des connexions individuelles de bases de données. Pour plus d'informations, voir "Fermeture des connexions de bases de données" à la page 20-22.

Spécification du comportement de la connexion de base de données par défaut

KeepConnections fournit la valeur par défaut de la propriété *KeepConnection* des composants base de données implicites créés à l'exécution. *KeepConnection* spécifie ce qu'il advient d'une connexion à une base de données établie pour un composant base de données quand tous ses ensembles de données sont fermés. Si elle a pour valeur *True* (valeur par défaut), une connexion de base de données constante, ou *persistante*, est maintenue même si aucun ensemble de données n'est actif. Si elle a pour valeur *False*, une connexion de base de données est interrompue dès que tous ses ensembles de données sont fermés.

Remarque La persistance de la connexion pour un composant base de données placé explicitement dans un module de données ou une fiche est contrôlée par la propriété *KeepConnection* de ce composant base de données. Si elle est définie différemment, la propriété *KeepConnection* d'un composant base de données prend toujours le pas sur la propriété *KeepConnections* de la session. Pour plus d'informations sur le contrôle des connexions individuelles de bases de données au sein d'une session, voir "Gestion des connexions de bases de données" à la page 20-22.

KeepConnections doit être à *True* pour les applications qui ouvrent et ferment souvent tous les ensembles de données associés à une base de données sur un serveur distant. Ce réglage réduit le trafic du réseau et accélère l'accès aux données, parce qu'il signifie qu'une connexion ne doit être ouverte et fermée qu'une seule fois durant la vie de la session. Sinon, chaque fois que l'application ferme ou rétablit une connexion, cela implique la surcharge d'attacher et de détacher la base de données.

Remarque Même avec *KeepConnections* à *True* pour une session, vous pouvez fermer et libérer les connexions de bases de données inactives pour tous les composants base de données implicites en appelant la méthode *DropConnections*. Pour plus d'informations sur *DropConnections*, voir "Interruption des connexions de bases de données inactives" à la page 20-23.

Gestion des connexions de bases de données

Vous pouvez utiliser un composant session pour gérer les connexions de bases de données en son sein. Le composant session inclut des propriétés et des méthodes pour :

- Ouvrir des connexions de bases de données.
- Fermer des connexions de bases de données.
- Fermer et libérer toutes les connexions de bases de données temporaires et inactives.
- Localiser des connexions de bases de données spécifiques.
- Parcourir toutes les connexions de bases de données ouvertes.

Ouverture de connexions de bases de données

Pour ouvrir une connexion de base de données au sein d'une session, appelez la méthode *OpenDatabase*. *OpenDatabase* prend un paramètre, le nom de la base de données à ouvrir. Ce nom est un alias BDE ou le nom d'un composant base de données. Pour Paradox ou dBASE, le nom peut aussi être un chemin d'accès qualifié. Par exemple, l'instruction suivante utilise la session par défaut et essaie d'ouvrir une connexion de base de données pour la base de données pointée par l'alias DBDEMOS :

```
var
  DBDemosDatabase: TDatabase;
begin
  DBDemosDatabase := Session.OpenDatabase('DBDEMOS');
  ...
```

OpenDatabase active la session si elle ne l'est pas déjà, puis vérifie si le nom de base de données spécifié concorde avec la propriété *DatabaseName* de l'un des composants base de données de la session. Si le nom ne correspond à aucun composant base de données existant, *OpenDatabase* crée un composant base de données temporaire en utilisant le nom spécifié. Finalement, *OpenDatabase* appelle la méthode *Open* du composant base de données pour se connecter au serveur. Chaque appel à *OpenDatabase* incrémente d'une unité un compteur de référence pour la base de données. Tant que ce compteur reste supérieur à 0, la base de données est ouverte.

Fermeture des connexions de bases de données

Pour fermer une connexion de base de données individuelle, appelez la méthode *CloseDatabase*. Quand vous appelez *CloseDatabase*, le compteur de référence de la base de données, incrémenté lors de l'appel de *OpenDatabase*, est décrémenté d'une unité. Quand le compteur de référence d'une base de données arrive à 0, la base de données est fermée. *CloseDatabase* prend un paramètre, la base de données à fermer. Si vous avez ouvert la base de données avec la méthode *OpenDatabase*, ce paramètre peut être défini sur la valeur renvoyée par celle-ci.

```
Session.CloseDatabase(DBDemosDatabase);
```

Si le nom de base de données spécifié est associé à un composant base de données temporaire (implicite), et si la propriété *KeepConnections* de la session a

pour valeur *False*, le composant base de données est libéré, fermant effectivement la connexion.

Remarque Si *KeepConnections* a pour valeur *False,* les composants base de données temporaires sont fermés et libérés automatiquement quand le dernier ensemble de données associé au composant base de données est fermé. Une application peut toujours appeler *CloseDatabase* avant ce moment pour forcer la fermeture. Pour libérer les composants base de données temporaires quand *KeepConnections* a pour valeur *True*, appelez la méthode *Close* du composant base de données, puis la méthode *DropConnections* de la session.

Remarque L'appel de *CloseDatabase* pour un composant base de données persistant ne ferme pas réellement la connexion. Pour cela, appelez directement la méthode *Close* du composant base de données.

Il existe deux façons de fermer toutes les connexions de bases de données au sein de la session :

- Mettre la propriété *Active* de la session à *False*.
- Appeler la méthode *Close* pour la session.

Quand vous mettez *Active* à *False*, Delphi appelle automatiquement la méthode *Close*. *Close* déconnecte toutes les bases de données actives en libérant les composants base de données temporaires et en appelant la méthode *Close* de chaque composant base de données persistant. Finalement, *Close* met le handle BDE de la session à **nil**.

Interruption des connexions de bases de données inactives

Si la propriété *KeepConnections* d'une session a pour valeur *True* (valeur par défaut), les connexions de bases de données pour les composants temporaires base de données sont maintenues même si tous les ensembles de données utilisés par le composant sont fermés. Vous pouvez éliminer ces connexions et libérer tous les composants base de données temporaires inactifs pour une session en appelant la méthode *DropConnections*. Par exemple, les lignes suivantes libèrent tous les composants base de données temporaires inactifs pour la session par défaut :

```
Session.DropConnections;
```

Les composants base de données temporaires pour lesquels un ou plusieurs ensembles de données sont actifs ne sont ni supprimés ni libérés par cet appel. Pour libérer ces composants, appelez *Close*.

Recherche d'une connexion de base de données

Utilisez la méthode *FindDatabase* d'une session pour déterminer si un composant base de données spécifique est déjà associé à une session. *FindDatabase* prend un paramètre, le nom de la base de données à rechercher. Ce nom est un alias BDE ou le nom d'un composant base de données. Pour Paradox ou dBASE, il peut aussi être un chemin d'accès qualifié.

FindDatabase renvoie le composant base de données s'il trouve une concordance. Sinon, il renvoie **nil**.

Le code suivant recherche dans la session par défaut un composant base de données en utilisant l'alias DBDEMOS, et, s'il n'en trouve pas, le crée et l'ouvre :

```
var
  DB: TDatabase;
begin
  DB := Session.FindDatabase('DBDEMOS');
  if (DB = nil) then                          { La base de données n'existe pas}
    DB := Session.OpenDatabase('DBDEMOS');    { la créer et l'ouvrir}
  if Assigned(DB) and DB.Connected then begin
    DB.StartTransaction;
    ...
  end;
end;
```

Parcourir les composants base de données d'une session

Vous pouvez utiliser deux propriétés de composant de session, *Databases* et *DatabaseCount*, pour parcourir tous les composants base de données actifs associés à une session.

Databases est un tableau de tous les composants base de données actifs et associés à une session. *DatabaseCount* est le nombre de bases de données de ce tableau. A mesure que les connexions sont ouvertes et fermées durant la vie d'une session, les valeurs de *Databases* et *DatabaseCount* changent. Par exemple, si la propriété *KeepConnections* d'une session a pour valeur *False* et si tous les composants base de données sont créés selon les besoins à l'exécution, chaque fois qu'une base de données est ouverte, *DatabaseCount* est incrémentée d'une unité. Chaque fois qu'une base de données est fermée, *DatabaseCount* est décrémentée d'une unité. Si *DatabaseCount* a pour valeur zéro, il n'y a actuellement plus de composant base de données actif pour la session.

Le code suivant définit la propriété *KeepConnection* de chaque base de données active dans la session par défaut à *True* :

```
var
  MaxDbCount: Integer;
begin
  with Session do
    if (DatabaseCount > 0) then
      for MaxDbCount := 0 to (DatabaseCount - 1) do
        Databases[MaxDbCount].KeepConnection := True;
end;
```

Manipulation des tables Paradox et dBASE protégées par mot de passe

Un composant session peut stocker des mots de passe pour les tables Paradox et dBASE protégées par mot de passe. Une fois un mot de passe ajouté à la session, votre application peut ouvrir les tables protégées par ce mot passe. Si vous retirez le mot de passe de la session, l'application ne pourra plus ouvrir les tables qui l'utilisent tant que vous ne l'ajoutez pas de nouveau à la session.

Utilisation de la méthode AddPassword

La méthode *AddPassword* offre une alternative pour qu'une application puisse fournir un mot de passe pour une session avant d'ouvrir une table Paradox ou dBase cryptée qui requiert un mot de passe pour son accès. Si vous n'ajoutez pas le mot de passe à la session, quand l'application tente d'ouvrir une table protégée par mot de passe, une boîte de dialogue demande le mot de passe à l'utilisateur.

AddPassword prend un paramètre, une chaîne contenant le mot de passe à utiliser. Vous pouvez appeler *AddPassword* autant de fois que nécessaire pour ajouter des mots de passe (un à la fois) pour accéder à des tables protégées par des mots de passe différents.

```
var
  Passwrd: String;
begin
  Passwrd := InputBox('Entrez le mot de passe', 'Mot de passe:', '');
  Session.AddPassword(Passwrd);
  try
    Table1.Open;
  except
    ShowMessage('Impossible d'ouvrir la table !');
    Application.Terminate;
  end;
end;
```

Remarque L'emploi de la fonction *InputBox*, ci-dessus, est pour démonstration seulement. Dans une application réelle, utilisez les fonctions de saisie de mot de passe, qui masquent ce dernier à mesure de sa frappe, comme la fonction *PasswordDialog* ou une fiche personnalisée.

Le bouton Ajouter de la boîte de dialogue de la fonction *PasswordDialog* a le même effet que la méthode *AddPassword*.

```
if PasswordDialog(Session) then
  Table1.Open
else
  ShowMessage('Aucun mot de passe fourni, impossible d'ouvrir la table !');
end;
```

Utilisation des méthodes RemovePassword et RemoveAllPasswords

RemovePassword supprime de la mémoire un mot de passe précédemment ajouté. *RemovePassword* prend un paramètre, une chaîne contenant le mot de passe à supprimer.

```
Session.RemovePassword('secret');
```

RemoveAllPasswords supprime de la mémoire tous les mots de passe précédemment ajoutés.

```
Session.RemoveAllPasswords;
```

Utilisation de la méthode GetPassword et de l'événement OnPassword

L'événement *OnPassword* vous permet de contrôler la façon dont votre application fournit les mots de passe pour les tables Paradox et dBASE quand c'est nécessaire. Fournissez un gestionnaire pour l'événement *OnPassword* si vous ne voulez pas du comportement par défaut. Si vous n'en fournissez pas, Delphi présente une boîte de dialogue par défaut pour l'entrée du mot de passe, sans comportement spécial -- si l'essai d'ouverture de la table ne réussit pas, une exception est déclenchée.

Si vous fournissez un gestionnaire pour l'événement *OnPassword*, vous devez faire deux choses dans ce gestionnaire : appeler la méthode *AddPassword* et définir le paramètre *Continue* du gestionnaire d'événement à True. La méthode *AddPassword* passe à la session une chaîne à utiliser comme mot de passe pour la table. Le paramètre *Continue* indique à Delphi qu'il n'est plus nécessaire d'inviter l'utilisateur à saisir le mot de passe pour cet essai d'ouverture de table. La valeur par défaut de *Continue* a pour valeur False, et il faut donc la mettre explicitement à True. Si *Continue* a pour valeur False après que le gestionnaire d'événement a fini son exécution, un événement *OnPassword* est de nouveau déclenché, même si un mot de passe valide a été passé par *AddPassword*. Si *Continue* a pour valeur True après l'exécution du gestionnaire d'événement et si la chaîne passée avec *AddPassword* n'est pas un mot de passe valide, l'essai d'ouverture de la table échoue et une exception est déclenchée.

OnPassword peut être déclenchée dans deux circonstances. La première est un essai d'ouverture d'une table protégée par mot de passe (dBASE ou Paradox) quand un mot de passe valide n'a pas encore été fourni à la session (si un mot de passe valide a déjà été fourni, l'événement *OnPassword* ne survient pas).

L'autre circonstance est un appel à la méthode *GetPassword*. *GetPassword* génère un événement *OnPassword*, ou, si la session n'a pas de gestionnaire d'événement *OnPassword*, affiche une boîte de dialogue de mot de passe par défaut. Elle renvoie True si le gestionnaire d'événement *OnPassword* ou la boîte de dialogue par défaut a ajouté un mot de passe à la session, et False si aucune entrée n'a été effectuée.

Dans l'exemple suivant, la méthode *Password* est désignée comme le gestionnaire d'événement *OnPassword* pour la session par défaut, en l'affectant à la propriété *OnPassword* de l'objet global *Session*.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Session.OnPassword := Password;
end;
```

Dans la méthode *Password*, la fonction *InputBox* demande un mot de passe à l'utilisateur. La méthode *AddPassword* fournit alors par programme le mot de passe saisi dans la boîte de dialogue à la session.

```
procedure TForm1.Password(Sender: TObject; var Continue: Boolean);
var
  Passwrd: String;
begin
  Passwrd := InputBox('Entrez le mot de passe', 'Mot de passe:', '');
```

```
    Continue := (Passwrd > '');
    Session.AddPassword(Passwrd);
  end;
```

L'événement *OnPassword* (et donc le gestionnaire d'événement *Password*) est déclenché par un essai d'ouverture d'une table protégée par mot de passe, comme présenté ci-dessus. Même si l'utilisateur est invité à fournir un mot de passe dans le gestionnaire d'événement *OnPassword*, l'essai d'ouverture peut échouer en cas de saisie d'un mot de passe incorrect ou si une autre erreur survient.

```
procedure TForm1.OpenTableBtnClick(Sender: TObject);
const
  CRLF = #13 + #10;
begin
  try
    Table1.Open;                          { cette ligne déclenche l'événement OnPassword}
  except
    on E:Exception do begin               { exception si impossible d'ouvrir la table }
      ShowMessage('Error!' + CRLF +       { afficher l'erreur expliquant le problème }
        E.Message + CRLF +
        'Terminating application...');
      Application.Terminate;              { fin de l'application }
    end;
  end;
end;
```

Spécification des répertoires Paradox

Deux propriétés de composant de session, *NetFileDir* et *PrivateDir*, sont spécifiques aux applications qui fonctionnent avec les tables Paradox.

NetFileDir spécifie le répertoire qui contient le fichier de contrôle de réseau Paradox, PDOXUSRS.NET. Ce fichier régente le partage des tables Paradox sur les lecteurs réseau. Toutes les applications qui ont besoin de partager des tables doivent spécifier le même répertoire pour le fichier de contrôle de réseau (en général, un répertoire situé sur un serveur de fichiers réseau). Delphi calcule une valeur pour *NetFileDir* à partir du fichier de configuration du moteur de bases de données Borland (BDE) pour un alias de base de données donné. Si vous définissez vous-même *NetFileDir*, la valeur que vous fournissez supplante le paramètre de configuration BDE ; assurez-vous donc de valider la nouvelle valeur.

Au moment de la conception, vous pouvez spécifier une valeur pour *NetFileDir* dans l'inspecteur d'objets. Vous pouvez aussi définir ou modifier *NetFileDir* par code à l'exécution. La ligne suivante définit *NetFileDir* pour la session par défaut à l'emplacement du répertoire depuis lequel l'application s'exécute :

```
Session.NetFileDir := ExtractFilePath(Application.EXEName);
```

Remarque *NetFileDir* ne peut être modifiée que lorsqu'une application n'a ouvert aucun fichier Paradox. Si vous modifiez *NetFileDir* à l'exécution, vérifiez qu'elle pointe vers un répertoire réseau valide partagé par vos utilisateurs réseau.

PrivateDir spécifie le répertoire de stockage des fichiers temporaires de traitement des tables, comme ceux générés par le BDE pour manipuler les instructions SQL locales. Si aucune valeur n'est spécifiée pour la propriété *PrivateDir*, le BDE utilise automatiquement le répertoire en cours au moment de son initialisation. Si votre application s'exécute directement sur un serveur de fichiers réseau, vous pouvez améliorer ses performances à l'exécution en définissant *PrivateDir* au disque dur local de l'utilisateur avant d'ouvrir la base de données.

Remarque Il ne faut pas définir *PrivateDir* au moment de la conception puis ouvrir la session dans l'EDI. Cela aurait pour conséquence de générer une erreur spécifiant que le répertoire est occupé lors de l'exécution de votre application depuis l'EDI.

Le code suivant définit le paramétrage de la propriété *PrivateDir* de la session par défaut sur le répertoire C:\TEMP d'un utilisateur :

```
Session.PrivateDir := 'C:\TEMP';
```

Important Ne définissez pas *PrivateDir* sur le répertoire racine d'un disque. Utilisez toujours un sous-répertoire.

Manipulation des alias BDE

Chaque composant de base de données associé à une session possède un alias BDE (même si un chemin d'accès qualifié peut être substitué à un alias pour accéder aux tables Paradox et dBASE). Une session peut créer, modifier et supprimer des alias durant son existence.

La méthode *AddAlias* crée un nouvel alias BDE pour un serveur de base de données SQL. *AddAlias* prend trois paramètres : une chaîne contenant le nom de l'alias, une chaîne qui spécifie le pilote SQL Links à utiliser, et une liste de chaînes contenant les paramètres de l'alias. Par exemple, les instructions suivantes utilisent *AddAlias* pour ajouter à la session par défaut un nouvel alias pour l'accès à un serveur InterBase :

```
var
  AliasParams: TStringList;
begin
  AliasParams := TStringList.Create;
  try
    with AliasParams do begin
      Add('OPEN MODE=READ');
      Add('USER NAME=TOMSTOPPARD');
      Add('SERVER NAME=ANIMALS:/CATS/PEDIGREE.GDB');
    end;
    Session.AddAlias('CATS', 'INTRBASE', AliasParams);
    ...
  finally
    AliasParams.Free;
  end;
end;
```

AddStandardAlias crée un nouvel alias BDE pour les tables Paradox, dBASE ou ASCII. *AddStandardAlias* prend trois paramètres de type chaîne : le nom de l'alias, le chemin d'accès qualifié vers la table Paradox ou dBASE, et le nom du pilote

par défaut à utiliser lors de l'essai d'ouverture d'une table dont le nom ne possède pas d'extension. Par exemple, l'instruction suivante utilise *AddStandardAlias* pour créer un nouvel alias pour l'accès à une table Paradox :

```
AddStandardAlias('MYDBDEMOS', 'C:\TESTING\DEMOS\', 'Paradox');
```

Quand vous ajoutez un alias à une session, le BDE stocke une copie de l'alias en mémoire, où il est seulement disponible à cette session et à toute autre session pour laquelle *cfmPersistent* est inclus dans la propriété *ConfigMode*. *ConfigMode* est un ensemble qui décrit quels types d'alias peuvent être utilisés par les bases de données dans la session. Le paramétrage par défaut est *cmAll*, qui se traduit par l'ensemble [*cfmVirtual*, *cfmPersistent*, *cfmSession*]. Si *ConfigMode* a pour valeur *cmAll*, une session peut voir tous les alias créés dans cette session (*cfmSession*), tous les alias du fichier de configuration BDE du système de l'utilisateur (*cfmPersistent*), et tous les alias que le BDE maintient en mémoire (*cfmVirtual*). Vous pouvez modifier *ConfigMode* pour restreindre les alias que les bases de données d'une session peuvent utiliser. Par exemple, définir *ConfigMode* à *cfmSession* restreint la vue d'une session aux seuls alias créés au sein de cette session. Tous les autres alias du fichier de configuration BDE ou en mémoire ne sont pas disponibles.

Pour rendre un alias nouvellement créé disponible à toutes les sessions et aux autres applications, utilisez la méthode *SaveConfigFile* de la session. *SaveConfigFile* écrit les alias en mémoire dans le fichier de configuration BDE, où ils peuvent être lus et utilisés par les autres applications BDE.

Après avoir créé un alias, vous pouvez modifier ses paramètres en appelant *ModifyAlias*. *ModifyAlias* prend deux paramètres : le nom de l'alias à modifier et une liste de chaînes contenant les paramètres à modifier et leurs valeurs. Par exemple, les instructions suivantes utilisent *ModifyAlias* pour changer le paramètre OPEN MODE pour l'alias CATS en READ/WRITE dans la session par défaut :

```
var
  List: TStringList;
begin
  List := TStringList.Create;
  with List do begin
    Clear;
    Add('OPEN MODE=READ/WRITE');
  end;
  Session.ModifyAlias('CATS', List);
  List.Free;
  ...
```

Pour supprimer un alias précédemment créé dans une session, appelez la méthode *DeleteAlias*. *DeleteAlias* prend un seul paramètre, le nom de l'alias à supprimer. *DeleteAlias* rend un alias indisponible à la session.

Remarque *DeleteAlias* ne supprime pas un alias du fichier de configuration BDE si l'alias y avait été écrit par un appel préalable à *SaveConfigFile*. Pour supprimer l'alias du fichier de configuration après l'appel à *DeleteAlias*, appelez *SaveConfigFile* de nouveau.

Les composants session fournissent cinq méthodes pour la récupération d'informations sur les alias BDE, y compris les informations de paramètre et de pilote. Ce sont :

- *GetAliasNames*, pour lister les alias auxquels une session a accès.
- *GetAliasParams*, pour lister les paramètres d'un alias spécifié.
- *GetAliasDriverName*, pour retourner le nom du pilote BDE utilisé par l'alias.
- *GetDriverNames*, pour retourner une liste de tous les pilotes BDE disponibles pour la session.
- *GetDriverParams*, pour retourner les paramètres de pilote pour un pilote spécifié.

Pour plus d'informations sur l'utilisation des méthodes d'information d'une session, voir "Récupération des informations d'une session" ci-dessous. Pour plus d'informations sur les alias BDE et les pilotes SQL Links avec lesquels ils fonctionnent, voir le fichier d'aide BDE en ligne, BDE32.HLP.

Récupération des informations d'une session

Vous pouvez récupérer les informations sur une session et ses composants base de données en utilisant les méthodes informatives de la session. Par exemple, une méthode récupère le nom de tous les alias connus de la session, et une autre méthode récupère le nom des tables associées à un composant base de données spécifique utilisé par la session. Le tableau suivant résume les méthodes informatives d'un composant session :

Tableau 20.4 Méthodes informatives de bases de données pour les composants session

Méthode	Action
GetAliasDriverName	Récupère le pilote BDE pour un alias de base de données spécifié.
GetAliasNames	Récupère la liste des alias BDE d'une base de données.
GetAliasParams	Récupère la liste des paramètres de l' alias BDE spécifié d'une base de données.
GetConfigParams	Récupère les informations de configuration à partir du fichier de configuration BDE.
GetDatabaseNames	Récupère la liste des alias BDE et les noms de tous les composants *TDatabase* en cours d'utilisation.
GetDriverNames	Récupère le nom de tous les pilotes BDE actuellement installés.
GetDriverParams	Récupère la liste des paramètres pour le pilote BDE spécifié.
GetStoredProcNames	Récupère les noms de toutes les procédures stockées pour la base de données spécifiée.
GetTableNames	Récupère les noms de toutes les tables correspondant au modèle spécifié pour la base de données spécifiée.
GetFieldNames	Récupère les noms de tous les champs de la table spécifiée d'une base de données spécifiée.

Sauf pour *GetAliasDriverName*, ces méthodes renvoient un ensemble de valeurs dans une liste de chaînes déclarée et maintenue par votre application. *GetAliasDriverName* renvoie une chaîne unique, le nom du pilote BDE en cours pour un composant de base de données particulier utilisé par la session.

Par exemple, le code suivant récupère les noms de tous les composants base de données et tous les alias connus de la session par défaut :

```
var
  List: TStringList;
begin
  List := TStringList.Create;
  try
    Session.GetDatabaseNames(List);
    ...
  finally
    List.Free;
  end;
end;
```

Création de sessions supplémentaires

Vous pouvez créer des sessions qui s'ajoutent à la session par défaut. Au moment de la conception, vous pouvez placer des sessions supplémentaires dans un module de données (ou une fiche), définir leurs propriétés dans l'inspecteur d'objets, leur écrire des gestionnaires d'événements, et écrire le code qui appelle leurs méthodes. Vous pouvez aussi créer des sessions, définir leurs propriétés et appeler leurs méthodes à l'exécution.

Remarque Créer des sessions supplémentaires est facultatif, à moins qu'une application n'exécute des requêtes simultanées sur une base de données, ou qu'elle soit multithread.

Pour permettre la création dynamique d'un composant session à l'exécution, suivez les étapes ci-dessous :

1. Déclarez une variable *TSession*.
2. Instanciez une nouvelle session en appelant la méthode *Create*. Le constructeur définit une liste vide de composants base de données pour la session, définit la propriété *KeepConnections* à *True*, et ajoute la session à la liste des sessions maintenue par le composant liste de sessions de l'application.
3. Définissez la propriété *SessionName* de la nouvelle session à un nom unique. Cette propriété est utilisée pour associer les composants base de données à la session. Pour plus d'informations sur la propriété *SessionName*, voir "Affectation d'un nom à une session" à la page 20-32.
4. Activez la session et ajustez ses propriétés si nécessaire.

Vous pouvez aussi créer et ouvrir des sessions avec la méthode *OpenSession* de *TSessionList*. L'emploi de *OpenSession* est plus sûr que l'appel de *Create*, parce que *OpenSession* ne crée la session que si elle n'existe pas déjà. Pour plus d'informations sur *OpenSession*, voir "Gestion de sessions multiples" à la page 20-32.

Le code suivant crée un nouveau composant session, lui affecte un nom et ouvre la session pour les opérations de bases de données qui suivent (non présentées ici). Après usage, la session est détruite par un appel à la méthode *Free*.

Remarque Ne détruisez jamais la session par défaut.

```
var
  SecondSession: TSession;
begin
  SecondSession := TSession.Create(Form1);
  with SecondSession do
    try
      SessionName := 'SecondSession';
      KeepConnections := False;
      Open;
      ...
    finally
      SecondSession.Free;
    end;
end;
```

Affectation d'un nom à une session

La propriété *SessionName* d'une session sert à nommer la session afin que vous puissiez lui associer des bases de données et des ensembles de données. Pour la session par défaut, *SessionName* est "Default" ; vous devez donner à la propriété *SessionName* de chaque nouveau composant que vous créez une valeur unique.

Les composants base de données et ensemble de données possèdent une propriété *SessionName* qui correspond à la propriété *SessionName* d'un composant session. Si vous laissez vide la propriété *SessionName* d'un composant base de données et ensemble de données, celui-ci est automatiquement associé à la session par défaut. Vous pouvez aussi définir *SessionName* pour un composant base de données ou ensemble de données à un nom correspondant à la propriété *SessionName* d'un composant session que vous créez.

Le code suivant utilise la méthode *OpenSession* du composant *TSessionList par défaut*, *Sessions*, pour ouvrir un nouveau composant session, définir sa propriété *SessionName* à "InterBaseSession", activer la session et associer un composant base de données existant *Database1* à cette session :

```
var
  IBSession: TSession;
  ...
begin
  IBSession := Sessions.OpenSession('InterBaseSession');
  Database1.SessionName := 'InterBaseSession';
end;
```

Gestion de sessions multiples

Si vous créez une application unique utilisant plusieurs threads pour effectuer des opérations de bases de données, vous devez créer une session supplémentaire pour chaque thread. La page BDE de la palette des composants contient un composant session que vous pouvez placer dans un module de données ou une fiche au moment de la conception.

Important Quand vous placez un composant session, vous devez aussi définir sa propriété *SessionName* à une valeur unique afin d'éviter les conflits avec la propriété *SessionName* de la session par défaut.

Placer un composant session au moment de la conception présuppose que le nombre de threads (et donc de sessions) requis par l'application à l'exécution est statique. Cependant, il est plus probable qu'une application doive créer des sessions dynamiquement. Pour cela, appelez la méthode *OpenSession* de l'objet global *Sessions* à l'exécution.

OpenSession ne demande qu'un seul paramètre, un nom de session qui doit être unique parmi tous les noms de sessions de l'application. Le code suivant crée dynamiquement et active une nouvelle session avec un nom généré unique :

```
Sessions.OpenSession('RunTimeSession' + IntToStr(Sessions.Count + 1));
```

Cette instruction génère un nom unique pour une nouvelle session en récupérant le nombre de sessions en cours et en l'augmentant de 1. Notez que si vous créez et détruisez dynamiquement des sessions à l'exécution, ce code exemple ne fonctionnera comme prévu. Néanmoins, il montre comment utiliser les propriétés et les méthodes de *Sessions* pour gérer les sessions multiples.

Sessions est une variable de type *TSessionList* qui est automatiquement instanciée pour les applications de bases de données BDE. Vous pouvez utiliser les propriétés et les méthodes de *Sessions* pour tracer les sessions multiples d'une application de base de données multithread. Le tableau suivant résume les propriétés et les méthodes du composant *TSessionList* :

Tableau 20.5 Propriétés et méthodes de TSessionList

Propriété ou Méthode	But
Count	Renvoie le nombre de sessions, actives et inactives, dans la liste des sessions.
FindSession	Recherche une session d'un nom spécifié et renvoie un pointeur dessus, ou **nil** s'il n'y en a pas. Si on lui passe un nom de session vide, *FindSession* renvoie un pointeur sur la session par défaut, *Session*.
GetSessionNames	Remplit une liste de chaînes avec les noms de tous les composants session couramment instanciés. Cette procédure ajoute toujours au moins une chaîne, "Default" pour la session par défaut.
List	Renvoie le composant session pour le nom de session spécifié. S'il n'y a pas de session avec ce nom,une exception est déclenchée.
OpenSession	Crée et active une nouvelle session et réactive une session existante pour le nom de session spécifié.
Sessions	Accéde à la liste des sessions par valeur ordinale.

En exemple à l'utilisation des propriétés et des méthodes de *Sessions* dans une application multithread, considérez ce qui se passe quand vous voulez ouvrir une connexion de base de données. Pour déterminer si une connexion existe déjà, utilisez la propriété *Sessions* pour parcourir chaque session de la liste de sessions, en commençant par la session par défaut. Examinez la propriété *Databases* de chaque composant session pour voir si la base de données en question est ouverte. Si vous découvrez qu'un autre thread utilise déjà la base de données désirée, examinez la session suivante de la liste.

Si aucun thread existant n'utilise la base de données, vous pouvez ouvrir la connexion avec cette session.

En revanche, si tous les threads existants utilisent la base de données, vous devez ouvrir une nouvelle session dans laquelle vous ouvrirez une autre connexion de base de données.

Si vous répliquez un module de données contenant une session dans une application multithread, où chaque thread contient sa propre copie du module de données, vous pouvez utiliser la propriété *AutoSessionName* pour vous assurer que tous les ensembles de données de ce module de données utilisent la session adéquate. Définir *AutoSessionName* à *True* permet à la session de générer dynamiquement son propre nom unique quand elle est créée à l'exécution. Elle affecte ensuite ce nom à chaque ensemble de données du module de données, redéfinissant tout nom de session explicitement défini. Cela assure que chaque thread possède sa propre session, et que chaque ensemble de données utilise la session dans son propre module de données.

Utilisation des transactions avec le BDE

Par défaut, le BDE fournit un contrôle de transaction implicite pour vos applications. Quand une application est sous contrôle de transaction implicite, une transaction distincte est utilisée pour chaque enregistrement d'un ensemble de données écrit dans la base de données sous-jacente. Les transactions implicites garantissent à la fois un minimum de conflits de mise à jour des enregistrements et une vue cohérente de la base de données. En revanche, comme chaque ligne de données écrite dans une base de données prend place dans sa propre transaction, le contrôle de transaction implicite peut conduire à un trafic réseau excessif et réduire les performances de l'application. De plus, le contrôle de transaction implicite ne protège pas les opérations logiques qui recouvrent plusieurs enregistrements.

Si vous contrôlez explicitement les transactions, vous pouvez choisir le meilleur moment pour lancer, valider ou annuler vos transactions. Quand vous développez des applications en environnement multi-utilisateur, surtout quand elles fonctionnent sur un serveur SQL distant, il vaut mieux contrôler les transactions explicitement.

Il existe deux façons, mutuellement exclusives de contrôler les transactions explicitement dans une application de base de données BDE :

- Utiliser le composant base de données pour contrôler les transactions. Le principal avantage de l'emploi des méthodes et des propriétés d'un composant base de données est que cela fournit une application propre, portable, qui ne dépend pas d'une base de données ou d'un serveur particulier. Ce type de contrôle de transaction est supporté par tous les composants connexion de bases de données et est décrit dans "Gestion des transactions" à la page 17-6.
- Utiliser le SQL transparent dans un composant requête pour passer des instructions SQL directement aux serveurs distants SQL ou ODBC. Le principal avantage du SQL transparent est que vous pouvez utiliser les

capacités de gestion avancée des transactions d'un serveur de base de données particulier, comme la mise en cache de schéma. Pour comprendre les avantages du modèle de gestion de transaction de votre serveur, voir la documentation de votre serveur de base de données. Pour plus d'informations sur l'utilisation du SQL transparent, voir "Utilisation du SQL transparent" ci-dessous.

Quand vous travaillez sur des bases de données locales, vous pouvez utiliser seulement le composant base de données pour créer des transactions explicites (les bases de données locales ne supportent pas le SQL transparent). Cependant, il existe des limitations à l'emploi des transactions locales. Pour plus d'informations sur l'utilisation des transactions locales, voir "Utilisation de transactions locales" à la page 20-36.

Remarque Vous pouvez minimiser le nombre des transactions nécessaires en mettant les mises à jour en mémoire cache. Pour plus d'informations sur les mises à jour en mémoire cache, voir "Utilisation d'un ensemble de données client pour mettre en cache les mises à jour" à la page 23-18 et "Utilisation du BDE pour placer en mémoire cache les mises à jour" à la page 20-37.

Utilisation du SQL transparent

Avec le SQL transparent, utilisez un composant *TQuery*, *TStoredProc* ou *TUpdateSQL* pour envoyer une instruction de contrôle de transaction SQL directement à un serveur de base de données distant. Le BDE ne traite pas l'instruction SQL. L'emploi du SQL transparent vous permet de tirer directement avantage des contrôles de transaction offerts par votre serveur, surtout quand ces contrôles ne sont pas standard.

Pour utiliser le SQL transparent pour contrôler une transaction, vous devez :

- Installer les pilotes SQL Links appropriés. Si vous avez choisi l'installation "typique" de Delphi, tous les pilotes SQL Links sont déjà correctement installés.
- Configurer votre protocole de réseau. Consultez votre administrateur réseau pour plus d'informations.
- Avoir accès à une base de données sur un serveur distant.
- Définir SQLPASSTHRU MODE à NOT SHARED avec l'explorateur SQL. SQLPASSTHRU MODE spécifie si les instructions BDE et SQL transparent peuvent partager les mêmes connexions de bases de données. Dans la plupart des cas, SQLPASSTHRU MODE est défini à SHARED AUTOCOMMIT. Cependant, vous ne pouvez pas partager les connexions de bases de données quand vous utilisez des instructions de contrôle de transaction. Pour plus d'informations sur les modes SQLPASSTHRU, voir le fichier d'aide de l'utilitaire d'administration BDE.

Remarque Quand SQLPASSTHRU MODE a pour valeur NOT SHARED, vous devez utiliser des composants base de données distincts pour les ensembles de données qui

passent des instructions de transaction SQL au serveur et pour les ensembles de données qui ne le font pas.

Utilisation de transactions locales

Le BDE supporte les transactions locales sur les tables Paradox, dBASE, Access et FoxPro. D'un point de vue programmation, il n'y a pas de différence pour vous entre une transaction locale et une transaction sur un serveur de base de données distant.

Remarque Quand vous utilisez des transactions avec des tables locales Paradox, dBASE, Access et FoxPro, définissez *TransIsolation* à *tiDirtyRead* au lieu d'utiliser la valeur par défaut *tiReadCommitted*. Une erreur BDE est renvoyée si *TransIsolation* est définie à une valeur autre que *tiDirtyRead* pour les tables locales.

Quand une transaction est démarrée sur une table locale, les mises à jour effectuées sur la table sont consignées. Chaque enregistrement du journal contient l'ancien tampon d'un enregistrement. Quand une transaction est active, les enregistrements mis à jour sont verrouillés jusqu'à ce que la transaction soit validée ou annulée. En cas d'annulation, les anciens tampons d'enregistrements sont appliqués au lieu des nouveaux, afin de les restaurer dans leur état antérieur.

Les transactions locales sont plus limitées que les transactions sur les serveurs SQL ou les pilotes ODBC. En particulier, les restrictions suivantes s'appliquent aux transactions locales :

- La récupération automatique en cas de problème n'est pas fournie.
- Les instructions de définition de données ne sont pas supportées.
- Les transactions ne peuvent pas être effectuées sur des tables temporaires.
- Le niveau *TransIsolation* doit être défini à *tiDirtyRead*.
- Pour Paradox, les transactions locales ne peuvent être effectuées que sur les tables comportant des index valides. Il est impossible d'annuler les modifications sur des tables Paradox qui n'ont pas d'index.
- Un nombre limité d'enregistrements peut être verrouillé et modifié. Avec les tables Paradox, vous êtes limité à 255 enregistrements. Avec dBASE la limite est de 100.
- Les transactions ne peuvent être effectuées sur le pilote ASCII BDE.
- Fermer un curseur sur une table durant une transaction annule celle-ci sauf si :
  - Plusieurs tables sont ouvertes.
  - Le curseur est fermé sur une table dans laquelle aucune modification n'a été effectuée.

Utilisation du BDE pour placer en mémoire cache les mises à jour

L'approche recommandée pour la mise en mémoire cache des mises à jour consiste à utiliser un ensemble de données client (*TBDEClientDataSet*) ou à connecter l'ensemble de données BDE à un ensemble de données client à l'aide d'un fournisseur d'ensemble de données. Les avantages de l'ensemble de données client sont traités dans "Utilisation d'un ensemble de données client pour mettre en cache les mises à jour" à la page 23-18.

Dans les cas les plus simples, vous pouvez choisir d'utiliser le BDE pour la mise en mémoire cache des mises à jour. Les ensembles de données BDE et les composants *TDatabase* fournissent des propriétés, des méthodes et des événements pour la gestion des mises à jour en mémoire cache. La plupart correspondent directement aux propriétés, méthodes et événements que vous utilisez avec les ensembles de données client et les fournisseurs d'ensembles de données lors de l'utilisation d'un ensemble de données client pour la mise en mémoire cache des mises à jour. Le tableau suivant présente ces propriétés, méthodes et événements, ainsi que leurs correspondances dans *TBDEClientDataSet* :

Tableau 20.6 Propriétés, méthodes et événements pour les mises à jour en mémoire cache

Sur ensembles de données BDE (ou TDatabase)	Sur TBDEClientDataSet	Fonction
CachedUpdates	Inutile pour les ensembles de données clients, qui pratiquent toujours la mise en mémoire cache des mises à jour.	Détermine si les mise à jour en mémoire cache sont activées pour l'ensemble de données.
UpdateObject	Utilise un gestionnaire d'événement *BeforeUpdateRecord*, ou, si l'on emploie *TClientDataSet*, utilise la propriété *UpdateObject* sur l'ensemble de données BDE source.	Spécifie l'objet mise à jour pour mettre à jour les ensembles de données en lecture seule.
UpdatesPending	*ChangeCount*	Indique si le cache local contient des enregistrements modifiés qu'il faut appliquer à la base de données.
UpdateRecordTypes	*StatusFilter*	Indique le type d'enregistrements mis à jour à rendre visibles en appliquant les mises à jour en mémoire cache.
UpdateStatus	*UpdateStatus*	Indique si un enregistrement est inchangé, modifié, inséré ou supprimé.
OnUpdateError	*OnReconcileError*	Un événement pour la gestion des erreurs de mise à jour, enregistrement par enregistrement.

Tableau 20.6 Propriétés, méthodes et événements pour les mises à jour en mémoire cache (suite)

Sur ensembles de données BDE (ou TDatabase)	Sur TBDEClientDataSet	Fonction
OnUpdateRecord	*BeforeUpdateRecord*	Un événement pour le traitement des mises à jour, enregistrement par enregistrement.
ApplyUpdates *ApplyUpdates* (base de données)	*ApplyUpdates*	Applique à la base de données les enregistrements du cache local.
CancelUpdates	*CancelUpdates*	Supprime toutes les mises à jour en attente du cache local, sans les appliquer.
CommitUpdates	*Reconcile*	Efface le cache des mises à jour après le succès de l'application des mises à jour.
FetchAll	*GetNextPacket* (et *PacketRecords*)	Copie les enregistrements de bases de données dans le cache local pour modification et mise à jour.
RevertRecord	*RevertRecord*	Annule les mises à jour de l'enregistrement en cours si elles ne sont pas encore appliquées.

Pour une vue d'ensemble du processus de mises à jour en mémoire cache, voir "Présentation de l'utilisation d'un cache pour les mises à jour" à la page 23-20.

Remarque Même si vous utilisez un ensemble de données client pour placer les mises à jour en mémoire cache, lisez la section sur les objets mise à jour, page 20-45. Vous pouvez utiliser les objets mise à jour dans le gestionnaire d'événement *BeforeUpdateRecord* de *TBDEClientDataSet* ou *TDataSetProvider* pour appliquer les mises à jour depuis des procédures stockées ou des requêtes multitables.

Activation des mises à jour BDE en mémoire cache

Pour utiliser le BDE pour les mises à jour en mémoire cache, l'ensemble de données BDE doit indiquer le placement des mises à jour en mémoire cache. Cela est spécifié en définissant la propriété *CachedUpdates* à *True*. Lorsque les mises à jour en mémoire cache sont activées, une copie de tous les enregistrements est placée en mémoire locale. Les utilisateurs consultent et éditent cette copie locale des données. Les modifications, insertions et suppressions ont aussi lieu en mémoire cache. Elles s'accumulent en mémoire jusqu'à ce que l'application applique ces modifications au serveur de base de données. Si les enregistrements modifiés sont appliqués avec succès à la base de données, l'enregistrement de ces modifications est libéré de la mémoire cache.

L'ensemble de données met toutes les mises à jour en mémoire cache jusqu'à ce que vous définissiez *CachedUpdates* à *False*. L'application des mises à jour présentes en mémoire cache ne désactive pas les futures mises à jour en mémoire cache ; cela écrit seulement l'ensemble en cours des modifications dans la base de données et les efface de la mémoire. L'annulation des mises à jour en appelant

CancelUpdates supprime toutes les modifications en cours dans la mémoire cache, mais n'empêche pas l'ensemble de données de continuer à placer en mémoire cache les modifications ultérieures.

Remarque Si vous désactivez les mises à jour en mémoire cache en définissant *CachedUpdates* à *False*, toutes les modifications en attente que vous n'avez pas encore appliquées sont perdues sans notification. Pour éviter la perte des modifications, testez la propriété *UpdatesPending* avant de désactiver les mises à jour en mémoire cache.

Application des mises à jour BDE en mémoire cache

L'application des mises à jour est un processus en deux phases qui doit se produire dans le contexte d'une transaction d'un composant base de données, afin que votre application puisse gérer facilement les erreurs. Pour plus d'informations sur la gestion des transactions avec les composants base de données, voir "Gestion des transactions" à la page 17-6.

Quand vous appliquez des mises à jour sous le contrôle d'une transaction de base de données, les événements suivants prennent place :

1. Une transaction de base de données démarre.
2. Les mises à jour en mémoire cache sont écrites dans la base de données (phase 1). Si vous le fournissez, un événement *OnUpdateRecord* est déclenché pour chaque enregistrement écrit dans la base de données. Si une erreur survient quand un enregistrement est appliqué à la base de données, l'événement *OnUpdateError* est déclenché si vous en fournissez un.
3. La transaction est validée si les opérations d'écriture sont réussies ; elle est annulée dans le cas contraire :

   Si l'écriture dans la base de données est réussie :

   - Les modifications de la base de données sont validées, ce qui termine la transaction de base de données.
   - Les mises à jour en mémoire cache sont validées, libérant le tampon interne du cache (phase 2).

   Si l'écriture dans la base de données échoue :

   - Les modifications de la base de données sont annulées, ce qui termine la transaction de base de données.
   - Les mises à jour en mémoire cache ne sont pas validées, et restent intactes dans le cache interne.

Pour plus d'informations sur la création et l'utilisation d'un gestionnaire d'événement *OnUpdateRecord*, voir "Création d'un gestionnaire d'événement OnUpdateRecord" à la page 20-42. Pour plus d'informations sur la gestion des erreurs de mise à jour lors de l'application des mises à jour présentes en mémoire cache, "Gestion des erreurs de mise à jour en mémoire cache" à la page 20-43.

Remarque L'application des mises à jour en mémoire cache est particulièrement difficile quand vous travaillez avec plusieurs ensembles de données en liaison maître/détail, car l'ordre dans lequel vous appliquez les mises à jour à chaque ensemble de données est significatif. Normalement, vous devez mettre à jour les tables maître avant les tables détail, sauf lors de la gestion des enregistrements supprimés, où cet ordre doit être inversé. En raison de cette difficulté, il est vivement recommandé d'utiliser des ensembles de données client lors de la mise en mémoire cache des mises à jour d'une fiche maître/détail. Les ensembles de données client gèrent automatiquement l'ordre des opérations dans les relations maître/détail.

Il y a deux façons d'appliquer les mises à jour BDE :

- Vous pouvez appliquer les mises à jour au moyen d'un composant base de données en appelant sa méthode *ApplyUpdates*. C'est l'approche la plus simple, car la base de données gère tous les détails de la gestion d'une transaction nécessaire au processus de mise à jour, et de l'effacement du cache de l'ensemble de données quand la mise à jour est terminée.
- Vous pouvez appliquer les mises à jour pour un ensemble de données unique en appelant les méthodes *ApplyUpdates* et *CommitUpdates* de l'ensemble de données. Quand vous appliquez les mises à jour au niveau de l'ensemble de données, vous devez coder explicitement la transaction pour le processus de mise à jour, et appeler explicitement *CommitUpdates* pour valider les mises à jour depuis le cache.

Important Pour appliquer les mises à jour depuis une procédure stockée ou une requête SQL qui ne renvoie pas un ensemble de résultats dynamique, vous devez utiliser *TUpdateSQL* pour spécifier comment effectuer les mises à jour. Pour les mises à jour de jointures (requêtes impliquant plusieurs tables), vous devez fournir un objet *TUpdateSQL* pour chaque table impliquée, et vous devez utiliser le gestionnaire d'événement *OnUpdateRecord* pour invoquer ces objets afin d'effectuer les mises à jour. Consultez "Utilisation d'objets mise à jour pour mettre à jour un ensemble de données" à la page 20-45 pour plus de détails.

Application des mises à jour en mémoire cache avec une base de données

Pour appliquer les mises à jour présentes en mémoire cache à un ou plusieurs ensembles de données dans le contexte d'une connexion de base de données, appelez la méthode *ApplyUpdates* du composant base de données. Le code suivant applique les mises à jour à l'ensemble de données *CustomersQuery* en réponse à un événement clic de bouton :

```
procedure TForm1.ApplyButtonClick(Sender: TObject);
begin
  // pour les bases de données locales Paradox, dBASE et FoxPro
  // définir TransIsolation à DirtyRead
  if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
    Database1.TransIsolation := tiDirtyRead;
  Database1.ApplyUpdates([CustomersQuery]);
end;
```

La séquence ci-dessus écrit les mises à jour en mémoire cache dans la base de données dans le contexte d'une transaction générée automatiquement. En cas de

succès, elle valide la transaction, puis les mises à jour présentes en mémoire cache. En cas d'échec, elle annule la transaction et laisse le cache de mise à jour inchangé. Dans ce dernier cas, vous devez gérer les erreurs de mises à jour en mémoire cache par le biais de l'événement *OnUpdateError* de l'ensemble de données. Pour plus d'informations sur la gestion des erreurs de mise à jour, voir "Gestion des erreurs de mise à jour en mémoire cache" à la page 20-43.

Le principal avantage à appeler la méthode *ApplyUpdates* d'un composant base de données est que vous pouvez mettre à jour tous les composants ensemble de données associés à la base de données. Le paramètre de la méthode *ApplyUpdates* pour une base de données est un tableau de *TDBDataSet*. Par exemple, le code suivant applique les mises à jour pour deux requêtes :

```
if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
  Database1.TransIsolation := tiDirtyRead;
Database1.ApplyUpdates([CustomerQuery, OrdersQuery]);
```

Application des mises à jour en mémoire cache avec les méthodes de composant base de données

Vous pouvez appliquer directement les mises à jour d'ensembles de données BDE individuels en utilisant les méthodes *ApplyUpdates* et *CommitUpdates* de l'ensemble de données. Chacune de ces méthodes encapsule une phase du processus de mise à jour :

1. *ApplyUpdates* écrit les mises à jour en mémoire cache dans une base de données (phase 1).
2. *CommitUpdates* efface le cache interne quand l'écriture dans la base de données est réussie (phase 2).

Le code suivant illustre la manière d'appliquer les mises à jour dans une transaction pour l'ensemble de données *CustomerQuery* :

```
procedure TForm1.ApplyButtonClick(Sender: TObject)
begin
  Database1.StartTransaction;
  try
    if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
      Database1.TransIsolation := tiDirtyRead;
    CustomerQuery.ApplyUpdates; {essai d'écriture des modifications dans la base de
données}
    Database1.Commit;           { si succès, valider les modifications }
  except
    Database1.Rollback;         { si échec, annuler les modifications }
    raise; { déclencher l'exception de nouveau pour éviter un appel à CommitUpdates }
  end;
  CustomerQuery.CommitUpdates;  { si succès, effacer le cache interne }
end;
```

Si une exception est déclenchée durant l'appel de *ApplyUpdates*, la transaction de base de données est annulée. Cette annulation garantit que la table de base de données sous-jacente n'est pas modifiée. L'instruction *raise* dans le bloc try...except déclenche l'exception, ce qui empêche l'appel de *CommitUpdates*.

Comme *CommitUpdates* n'est pas appelée, le cache interne des mises à jour n'est pas vidé, ce qui vous laisse la possibilité de gérer les conditions d'erreur et de relancer la mise à jour.

Création d'un gestionnaire d'événement OnUpdateRecord

Quand un ensemble de données BDE applique ses mises à jour en mémoire cache, il parcourt les modifications enregistrées dans son cache, essayant de les appliquer aux enregistrements correspondants dans la table de base de données. Au moment où chaque modification, insertion ou suppression est sur le point d'être appliquée, l'événement *OnUpdateRecord* du composant ensemble de données est déclenché.

Fournir un gestionnaire pour l'événement *OnUpdateRecord* permet d'effectuer des actions juste avant l'application effective de la mise à jour de l'enregistrement en cours. Ces actions peuvent comprendre une validation spéciale des données, la mise à jour d'autres tables, des substitutions spéciales de paramètres ou l'exécution de multiples objets mise à jour. Un gestionnaire d'événement *OnUpdateRecord* donne un meilleur contrôle sur le processus de mise à jour.

Voici le code squelette d'un gestionnaire d'événement *OnUpdateRecord* :

```
procedure TForm1.DataSetUpdateRecord(DataSet: TDataSet;
 UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { effectuer les mises à jour ici... }
end;
```

Le paramètre *DataSet* spécifie l'ensemble de données ayant des mises à jour en mémoire cache.

Le paramètre *UpdateKind* indique le type de mise à jour qui doit être effectué sur l'enregistrement en cours. *UpdateKind* peut prendre les valeurs *ukModify*, *ukInsert* et *ukDelete*. Si vous utilisez un objet mise à jour, vous devez lui passer ce paramètre lors de l'application de la mise à jour. Vous pouvez aussi avoir besoin d'inspecter ce paramètre si votre gestionnaire effectue un traitement spécial selon le type de mise à jour.

Le paramètre *UpdateAction* indique si vous avez appliqué la mise à jour. *UpdateAction* peut prendre les valeurs *uaFail* (valeur par défaut), *uaAbort*, *uaSkip*, *uaRetry*, *uaApplied*. Si votre gestionnaire d'événement applique la mise à jour avec succès, donnez à ce paramètre la valeur *uaApplied* avant de sortir. Si vous décidez de ne pas mettre à jour l'enregistrement en cours, donnez-lui la valeur *uaSkip* pour préserver les modifications en mémoire cache non appliquées. Si vous ne changez pas la valeur de *UpdateAction*, toute l'opération de mise à jour de l'ensemble de données est annulée et une exception est déclenchée. Vous pouvez supprimer le message d'erreur (et déclencher une exception silencieuse) en donnant à *UpdateAction* la valeur *uaAbort*.

En plus de ces paramètres, vous utiliserez les propriétés *OldValue* et *NewValue* pour le composant champ associé à l'enregistrement en cours. *OldValue* donne la valeur originale du champ telle que récupérée de la base de données. Elle peut servir à localiser l'enregistrement de base de données à mettre à jour. *NewValue* est la valeur modifiée dans la mise à jour que vous essayez d'appliquer.

Important Un gestionnaire d'événement *OnUpdateRecord*, comme un gestionnaire d'événement *OnUpdateError* ou *OnCalcFields*, ne doit jamais appeler les méthodes qui modifient l'enregistrement en cours d'un ensemble de données.

L'exemple suivant illustre comment utiliser ces paramètres et propriétés. Il emploie un composant *TTable* nommé *UpdateTable* pour appliquer les mises à jour. En pratique, il est plus facile d'utiliser un objet mise à jour, mais l'emploi d'une table illustre plus clairement les possibilités.

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  if UpdateKind = ukInsert then
    UpdateTable.AppendRecord([DataSet.Fields[0].NewValue, DataSet.Fields[1].NewValue])
  else
    if UpdateTable.Locate('KeyField', VarToStr(DataSet.Fields[1].OldValue), []) then
      case UpdateKind of
        ukModify:
          begin
            UpdateTable.Edit;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            UpdateTable.Post;
          end;
        ukInsert:
          begin
            UpdateTable.Insert;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            UpdateTable.Post;
          end;
        ukDelete: UpdateTable.Delete;
      end;
        UpdateAction := uaApplied;
end;
```

Gestion des erreurs de mise à jour en mémoire cache

Quand il essaie d'appliquer les mises à jour, le moteur de bases de données Borland (BDE) recherche les conflits de mise à jour utilisateur et autres conditions, et il signale les erreurs éventuelles. L'événement *OnUpdateError* du composant ensemble de données vous permet de capturer les erreurs et d'y répondre. Il est bon de créer un gestionnaire pour cet événement si vous utilisez les mises à jour en mémoire cache. Si vous ne le faites pas et qu'une erreur survient, toute l'opération de mise à jour échoue.

Voici le code squelette d'un gestionnaire d'événement *OnUpdateError* :

```
procedure TForm1.DataSetUpdateError(DataSet: TDataSet; E: EDatabaseError;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { ... effectuer ici la gestion d'erreur de mise à jour ... }
end;
```

DataSet référence l'ensemble de données auquel les mises à jour sont appliquées. Vous pouvez utiliser cet ensemble de données pour accéder aux nouvelles et anciennes valeurs durant la gestion d'erreur. Les valeurs originales des champs

de chaque enregistrement sont stockées dans une propriété *TField* en lecture seule appelée *OldValue*. Les valeurs modifiées sont stockées dans la propriété *TField* analogue *NewValue*. Ces valeurs constituent la seule possibilité d'inspecter et de modifier les valeurs de mise à jour dans le gestionnaire d'événement.

Attention N'appelez aucune méthode d'ensemble de données susceptible de changer l'enregistrement en cours (comme *Next* et *Prior*). Cela ferait entrer le gestionnaire d'événement dans une boucle sans fin.

Le paramètre *E* est habituellement du type *EDBEngineError*. Dans ce type d'exception, vous pouvez extraire un message d'erreur que vous pouvez afficher aux utilisateurs dans votre gestionnaire d'erreur. Par exemple, le code suivant affiche le message d'erreur dans l'intitulé d'une boîte de dialogue :

```
ErrorLabel.Caption := E.Message;
```

Ce paramètre est aussi utile pour déterminer la cause exacte de l'erreur de mise à jour. Vous pouvez récupérer les codes d'erreur spécifiques depuis *EDBEngineError*, et prendre les mesures appropriées.

Le paramètre *UpdateKind* décrit le type de mise à jour qui a généré l'erreur. Sauf si votre gestionnaire d'erreur doit entreprendre des actions spéciales selon le type d'erreur rapporté, vous ne l'utiliserez sans doute pas.

Le tableau suivant présente les valeurs possibles de *UpdateKind* :

Tableau 20.7 Valeurs UpdateKind

Valeur	Signification
ukModify	L'édition d'un enregistrement existant a généré une erreur.
ukInsert	L'insertion d'un enregistrement a généré une erreur.
ukDelete	La suppression d'un enregistrement existant a généré une erreur.

UpdateAction indique au BDE comment continuer le processus de mise à jour à la sortie de votre gestionnaire d'événement. Au premier appel de votre gestionnaire d'erreur de mise à jour, la valeur de ce paramètre est toujours définie à *uaFail*. Selon la condition d'erreur de l'enregistrement qui a généré l'erreur et ce que vous faites pour la corriger, vous définissez généralement *UpdateAction* à une valeur différente avant de quitter le gestionnaire :

- Si votre gestionnaire d'erreur peut corriger la condition d'erreur qui a provoqué son intervention, définissez *UpdateAction* à l'action appropriée à entreprendre en sortie. Pour les conditions d'erreur que vous corrigez, définissez *UpdateAction* à *uaRetry* pour appliquer de nouveau la mise à jour de l'enregistrement.
- Si vous utilisez la valeur *uaSkip*, la mise à jour de la ligne qui a provoqué l'erreur est ignorée et la mise à jour de l'enregistrement reste dans le cache quand toutes les autres mises à jour sont terminées.
- *uaFail* et *uaAbort* mettent toutes deux un terme à toute l'opération de mise à jour. *uaFail* déclenche une exception et affiche un message d'erreur. *uaAbort* déclenche une exception silencieuse (sans afficher de message d'erreur).

Le code suivant montre comment un gestionnaire d'événement *OnUpdateError* vérifie si l'erreur de mise à jour est liée à une violation de clé, et dans ce cas définit le paramètre *UpdateAction* à *uaSkip* :

```
{ Ajouter 'Bde' à votre clause uses pour cet exemple }
if (E is EDBEngineError) then
  with EDBEngineError(E) do begin
    if Errors[ErrorCount - 1].ErrorCode = DBIERR_KEYVIOL then
      UpdateAction := uaSkip         { violation de clé, ignorer cet enregistrement }
    else
      UpdateAction := uaAbort;       { ne sait pas ce qui est faux, annuler la mise à jour
}
  end;
```

Remarque Si une erreur survient durant l'application des mises à jour en mémoire cache, une exception est déclenchée et un message d'erreur est affiché. A moins que *ApplyUpdates* ne soit appelé dans une construction try...except, un message d'erreur affiché depuis votre gestionnaire d'événement *OnUpdateError* peut provoquer un double affichage par votre application du même message d'erreur. Pour éviter la duplication du message d'erreur, définissez *UpdateAction* à *uaAbort* pour désactiver l'affichage des messages d'erreur générés par le système.

Utilisation d'objets mise à jour pour mettre à jour un ensemble de données

Quand l'ensemble de données BDE représente une procédure stockée ou une requête non "dynamique", il n'est pas possible d'appliquer les mises à jour directement depuis l'ensemble de données. De tels ensembles de données peuvent aussi provoquer un problème quand vous utilisez un ensemble de données client pour placer les mises à jour en mémoire cache. Que vous utilisiez le BDE ou un ensemble de données client pour placer les mises à jour en mémoire cache, vous devez gérer ces problèmes en utilisant un objet mise à jour :

1. Si vous utilisez un ensemble de données client, utilisez un composant fournisseur externe avec *TClientDataSet* plutôt que *TBDEClientDataSet*. Vous pouvez ainsi définir la propriété *UpdateObject* de l'ensemble de données BDE source (étape 3).
2. Ajoutez un composant *TUpdateSQL* au même module de données que celui de l'ensemble de données BDE.
3. Affectez à la propriété *UpdateObject* du composant ensemble de données BDE le composant *TUpdateSQL* dans le module de données.
4. Spécifiez les instructions SQL nécessaires pour effectuer les mises à jour en utilisant les propriétés *ModifySQL*, *InsertSQL* et *DeleteSQL* de l'objet mise à jour. Vous pouvez utiliser l'éditeur de mise à jour SQL pour composer ces instructions.
5. Fermez l'ensemble de données.

6 Affectez à la propriété *CachedUpdates* du composant ensemble de données la valeur *True* ou liez l'ensemble de données à l'ensemble de données client à l'aide d'un fournisseur d'ensemble de données.

7 Rouvrez l'ensemble de données.

Remarque Parfois, vous devez utiliser plusieurs objets mise à jour . Par exemple, lors de la mise à jour d'une jointure multitable ou d'une procédure stockée qui représente des données venant de plusieurs ensembles de données, vous devez fournir un objet *TUpdateSQL* pour chaque table à mettre à jour. Quand vous utilisez plusieurs objets mise à jour, vous ne pouvez pas associer simplement l'objet mise à jour à l'ensemble de données en définissant la propriété *UpdateObject*. Vous devez plutôt appeler manuellement l'objet mise à jour depuis un gestionnaire d'événement *OnUpdateRecord* (si vous utilisez le BDE pour placer les mises à jour en mémoire cache) ou *BeforeUpdateRecord* (si vous utilisez un ensemble de données client).

L'objet mise à jour encapsule en réalité trois composants *TQuery*. Chacun de ces composants requête effectue une tâche de mise à jour unique. Le premier fournit une instruction SQL UPDATE pour modifier les enregistrements existants ; le second composant fournit une instruction INSERT pour ajouter de nouveaux enregistrements à une table ; le troisième fournit une instruction DELETE pour supprimer des enregistrements d'une table.

Quand vous placez un composant mise à jour dans un module de données, vous ne voyez pas les composants requête qu'il encapsule. Ils sont créés à l'exécution par le composant mise à jour à partir de trois propriétés de mise à jour pour lesquelles vous fournissez des instructions SQL :

- *ModifySQL* spécifie l'instruction UPDATE.
- *InsertSQL* spécifie l'instruction INSERT.
- *DeleteSQL* spécifie l'instruction DELETE.

A l'exécution, quand le composant de mise à jour est utilisé pour appliquer les mises à jour, il :

1 Sélectionne une instruction SQL à exécuter selon que l'enregistrement en cours est modifié, inséré ou supprimé.

2 Fournit les valeurs de paramètre aux instructions SQL.

3 Prépare et exécute l'instruction SQL pour effectuer la mise à jour spécifiée.

Création d'instructions SQL pour les composants mise à jour

Pour mettre à jour un enregistrement dans un ensemble de données associé, un objet mise à jour utilise l'une des trois instructions SQL. Chaque objet mise à jour permet de mettre à jour une seule table, et les instructions de mise à jour de chaque objet doivent référencer la même table de base de données.

Les trois instructions SQL suppriment, insèrent et modifient les enregistrements mis en mémoire cache en vue d'une mise à jour. Vous devez fournir ces instructions sous la forme de propriétés *DeleteSQL*, *InsertSQL*, et *ModifySQL* de l'objet mise à jour. Vous pouvez fournir ces valeurs à la conception ou à

l'exécution. Par exemple, le code suivant spécifie à l'exécution une valeur pour la propriété *DeleteSQL* :

```
with UpdateSQL1.DeleteSQL do begin
  Clear;
  Add('DELETE FROM Inventory I');
  Add('WHERE (I.ItemNo = :OLD_ItemNo)');
end;
```

A la conception, vous pouvez utiliser l'éditeur SQL de mise à jour pour vous aider à composer les instructions SQL qui appliquent les mises à jour.

Les objets mise à jour fournissent une liaison automatique des paramètres pour les paramètres qui référencent les valeurs originales et modifiées des champs de l'ensemble de données. Vous insérez donc normalement des paramètres avec des noms spécialement formatés quand vous composez les instructions SQL . Pour plus d'informations sur l'utilisation de ces paramètres, voir "Substitution de paramètres dans les instructions SQL de mise à jour" à la page 20-48.

Utilisation de l'éditeur SQL de mise à jour

Pour créer les instructions SQL d'un composant mise à jour :

1. Avec l'inspecteur d'objets, sélectionnez le nom de l'objet mise à jour dans la liste déroulante de la propriété *UpdateObject* de l'ensemble de données. Cette étape garantit que l'éditeur SQL de mise à jour invoqué dans l'étape suivante détermine les valeurs par défaut adéquates pour les options de génération SQL.
2. Faites un clic droit sur l'objet mise à jour et sélectionnez l'éditeur UpdateSQL dans le menu contextuel. L'éditeur SQL de mise à jour s'affiche. Il crée les instructions SQL pour les propriétés *ModifySQL*, *InsertSQL* et *DeleteSQL* de l'objet mise à jour, selon l'ensemble de données sous-jacent et les valeurs que vous lui fournissez.

L'éditeur SQL de mise à jour possède deux pages. La page Options est visible la première fois que vous invoquez l'éditeur. Utilisez la boîte à options Nom de table pour sélectionner la table à mettre à jour. Quand vous spécifiez un nom de table, les boîtes liste Champs clé et Mettre à jour les champs se remplissent avec les colonnes disponibles.

La boîte liste Mettre à jour les champs indique les colonnes à mettre à jour. Quand vous spécifiez une table pour la première fois, toutes les colonnes de cette boîte liste sont sélectionnées pour être incluses. Vous pouvez faire une sélection multiple des champs selon vos besoins.

La boîte liste Champs clé sert à spécifier les colonnes à utiliser comme clés durant la mise à jour. Pour Paradox, dBASE et FoxPro les colonnes que vous spécifiez doivent correspondre à un index existant, mais ce n'est pas obligatoire pour les bases de données SQL distantes. Au lieu de définir Champs clé, vous pouvez cliquer sur le bouton Clés primaires pour choisir les champs clé de la mise à jour en fonction de l'index primaire de la table. Cliquez sur Valeurs du Dataset pour ramener les listes de sélection à leur état original : tous les champs sélectionnés en tant que clés et tous ceux sélectionnés pour mise à jour.

Cochez la case Noms de champs entre guillemets si votre serveur requiert des guillemets autour des noms de champs.

Après avoir spécifié une table, sélectionnez les colonnes clés et les colonnes à mettre à jour, cliquez sur Générer le SQL pour générer les instructions SQL préliminaires à associer aux propriétés *ModifySQL*, *InsertSQL* et *DeleteSQL* du composant mise à jour. Dans la plupart des cas, vous devrez affiner les instructions SQL générées automatiquement.

Pour afficher et modifier les instructions SQL générées, sélectionnez la page SQL. Si vous avez généré des instructions SQL, l'instruction de la propriété *ModifySQL* y est déjà affichée dans la zone mémo Texte SQL. Vous pouvez éditer l'instruction selon vos désirs.

Important Gardez à l'esprit que les instructions SQL générées sont des points de départ pour la création d'instructions de mise à jour. Il peut être nécessaire de les modifier pour qu'elles s'exécutent correctement. Par exemple, si vous travaillez sur des données contenant des valeurs NULL, il faut modifier la clause WHERE comme ceci :

```
WHERE field IS NULL
```

plutôt que d'utiliser la variable champ générée. Testez chaque instruction directement avant de l'accepter.

Utilisez les boutons radio Type d'instruction pour basculer d'une instruction générée à une autre et les éditer.

Pour accepter les instructions et les associer avec les propriétés SQL du composant mise à jour, cliquez sur OK.

Substitution de paramètres dans les instructions SQL de mise à jour

Les instructions SQL de mise à jour utilisent une forme spéciale de substitution de paramètres qui vous permet de substituer les anciennes ou les nouvelles valeurs de champs dans les mises à jour d'enregistrements. Quand l'éditeur SQL de mise à jour génère ses instructions, il détermine quelles valeurs de champs utiliser. Quand vous écrivez le code SQL de mise à jour, vous spécifiez les valeurs de champs à utiliser.

Quand le nom d'un paramètre correspond à celui d'une colonne d'une table, la nouvelle valeur du champ dans la mise à jour en mémoire cache de l'enregistrement est utilisée automatiquement comme valeur du paramètre. Quand le nom d'un paramètre correspond à celui d'une colonne commençant par la chaîne "OLD_", l'ancienne valeur du champ est utilisée. Par exemple, dans l'instruction SQL de mise à jour ci-dessous, le paramètre :LastName est automatiquement rempli avec la nouvelle valeur du champ dans la mise à jour en mémoire cache pour l'enregistrement inséré.

```
INSERT INTO Names
(LastName, FirstName, Address, City, State, Zip)
VALUES (:LastName, :FirstName, :Address, :City, :State, :Zip)
```

Les nouvelles valeurs de champs sont habituellement utilisées dans les instructions *InsertSQL* et *ModifySQL*. Dans la mise à jour d'un enregistrement

modifié, la nouvelle valeur de champ du cache de mise à jour est utilisée par l'instruction UPDATE pour remplacer l'ancienne valeur de champ dans la table de base mise à jour.

Dans le cas d'un enregistrement supprimé, il n'y a pas de nouvelles valeurs, et la propriété *DeleteSQL* utilise donc la syntaxe ":OLD_FieldName". Les anciennes valeurs de champs sont aussi utilisées normalement dans la clause WHERE de l'instruction SQL pour une mise à jour par modification ou suppression pour déterminer quel enregistrement mettre à jour ou supprimer.

Dans la clause WHERE d'une instruction SQL de mise à jour UPDATE ou DELETE, fournissez au moins le nombre minimal de paramètres nécessaires pour identifier sans ambiguïté l'enregistrement de la table de base qui est mis à jour à partir des données en mémoire cache. Par exemple, dans une liste de clients, l'utilisation du nom de famille peut ne pas être suffisant pour identifier de manière unique l'enregistrement adéquat dans la table de base. Il peut y avoir plusieurs enregistrements sous le nom de "Martin". L'utilisation de paramètres pour le nom, le prénom et le numéro de téléphone peut constituer une combinaison assez précise, à défaut d'un code client unique.

Remarque Si vous créez des instructions SQL contenant des paramètres qui ne se réfèrent pas aux valeurs de champs originales ou modifiées, l'objet mise à jour ne sait pas comment relier leurs valeurs. Vous pouvez cependant le faire manuellement, en utilisant la propriété *Query* de l'objet mise à jour . Consultez "Utilisation de la propriété Query d'un composant mise à jour" à la page 20-54 pour plus de détails.

Elaboration des instructions SQL de mise à jour

Au moment de la conception, vous pouvez utiliser l'éditeur SQL de mise à jour pour écrire les instructions SQL pour les propriétés *DeleteSQL*, *InsertSQL* et *ModifySQL*. Si vous n'utilisez pas cet éditeur, ou si vous voulez modifier les instructions générées, respectez les points suivants en écrivant les instructions de suppression, d'insertion et de modification des enregistrements de la table de base.

La propriété *DeleteSQL* ne doit contenir qu'une instruction SQL avec la commande DELETE. La table de base à mettre à jour doit être nommée dans la clause FROM. Pour que l'instruction SQL ne supprime que l'enregistrement de la table de base correspondant à celui supprimé dans la mémoire cache de mise à jour, utilisez une clause WHERE. Dans la clause WHERE, utilisez un paramètre pour un ou plusieurs champs afin d'identifier de manière unique l'enregistrement dans la table de base correspondant à celui figurant dans la mémoire cache de mise à jour. Si les paramètres sont nommés comme les champs et précédés du préfixe "OLD_", ils reçoivent automatiquement les valeurs des champs correpondants de l'enregistrement figurant dans la mémoire cache de mise à jour. Si les paramètres sont nommés d'une autre façon, vous devez fournir les valeurs de paramètre.

```
DELETE FROM Inventory I
WHERE (I.ItemNo = :OLD_ItemNo)
```

Certains types de tables peuvent être incapables de trouver l'enregistrement dans la table de base quand les champs utilisés pour identifier l'enregistrement contiennent des valeurs NULL. Dans ces cas, la mise à jour par suppression échoue pour ces enregistrements. Pour prendre cela en compte, ajoutez une condition pour les champs pouvant contenir une valeur NULL, à l'aide du prédicat IS NULL (en plus d'une condition pour une valeur non NULL). Par exemple, si le champ FirstName peut contenir une valeur NULL :

```
DELETE FROM Names
WHERE (LastName = :OLD_LastName) AND
  ((FirstName = :OLD_FirstName) OR (FirstName IS NULL))
```

L'instruction *InsertSQL* ne doit contenir qu'une instruction SQL avec la commande INSERT. La table de base à mettre à jour doit être nommée dans la clause INTO. Dans la clause VALUES, fournissez une liste de paramètres séparés par des vigules. Si les paramètres sont nommés comme les champs, ils reçoivent automatiquement les valeurs des champs correspondants de l'enregistrement figurant dans la mémoire cache de mise à jour. Si les paramètres sont nommés d'une autre façon, vous devez fournir les valeurs de paramètres. La liste de paramètres fournit les valeurs des champs du nouvel enregistrement inséré. Il doit y avoir autant de paramètres de valeur que de champs listés dans l'instruction.

```
INSERT INTO Inventory
(ItemNo, Amount)
VALUES (:ItemNo, 0)
```

L'instruction *ModifySQL* ne doit contenir qu'une instruction SQL avec la commande UPDATE. La table de base à mettre à jour doit être nommée dans la clause FROM. Incluez une ou plusieurs affectations de valeur dans la clause SET. Si les valeurs des affectations de la clause SET sont des paramètres nommés comme les champs, les paramètres reçoivent automatiquement les valeurs des champs correspondants de l'enregistrement mis à jour figurant dans la mémoire cache. Vous pouvez affecter d'autres valeurs de champ en utilisant d'autres paramètres, tant que les paramètres ne portent pas le nom de champs existants et que vous fournissiez les valeurs. Comme avec l'instruction *DeleteSQL*, fournissez une clause WHERE pour identifier de manière unique l'enregistrement de la table de base à mettre à jour, en utilisant des paramètres nommés comme les champs et précédés du préfixe "OLD_". Dans l'instruction de mise à jour ci-dessous, le paramètre :ItemNo reçoit automatiquement une valeur, contrairement à :Price.

```
UPDATE Inventory I
SET I.ItemNo = :ItemNo, Amount = :Price
WHERE (I.ItemNo = :OLD_ItemNo)
```

Considérant le code SQL de mise à jour ci-dessus, prenons un exemple où l'utilisateur final de l'application modifie un enregistrement existant. La valeur d'origine du champ ItemNo est 999. Dans une grille connectée à l'ensemble de données en mémoire cache, l'utilisateur final modifie la valeur du champ ItemNo en 123, et celle du champ Amount en 20. Quand la méthode ApplyUpdates est invoquée, cette instruction SQL affecte tous les enregistrements de la table de base dont le champ ItemNo a pour valeur 999, en utilisant l'ancienne valeur

contenue dans le paramètre :OLD_ItemNo. Dans ces enregistrements, l'instruction modifie la valeur du champ ItemNo field en 123 (en utilisant le paramètre :ItemNo, la valeur provenant de la grille) et celle du champ Amount en 20.

Utilisation de plusieurs objets mise à jour

Quand plusieurs tables de base référencées dans l'ensemble de données de mise à jour doivent être mises à jour, vous devez utiliser plusieurs objets mise à jour, à raison d'un par table de base mise à jour. Comme l'objet *UpdateObject* du composant ensemble de données ne permet d'associer qu'un seul objet mise à jour à l'ensemble de données, vous devez associer chaque objet mise à jour à un ensemble de données en affectant à sa propriété *DataSet* le nom de l'ensemble de données.

Conseil Quand vous utilisez plusieurs objets mise à jour, vous pouvez utiliser *TBDEClientDataSet* au lieu de *TClientDataSet* avec un fournisseur externe. La raison est qu'il n'est pas nécessaire de définir la propriété *UpdateObject* de l'ensemble de données source.

La propriété *DataSet* des objets mise à jour n'est pas disponible dans l'inspecteur d'objets à la conception. Vous ne pouvez définir cette propriété qu'à l'exécution.

```
UpdateSQL1.DataSet := Query1;
```

L'objet mise à jour utilise cet ensemble de données pour obtenir les valeurs de champs originales et modifiées pour la substitution de paramètres et, s'il s'agit d'un ensemble de données BDE, pour identifier la session et la base de données à utiliser lors de l'application des mises à jour. Pour que la substitution de paramètres fonctionne correctement, la propriété *DataSet* de l'objet mise à jour doit être l'ensemble de données qui contient les valeurs de champs mises à jour. Lors de l'utilisation d'un ensemble de données BDE pour mettre en mémoire cache les mises à jour, il s'agit de l'ensemble de données BDE lui-même. Lors de l'utilisation d'un ensemble de données client, c'est un ensemble de données client qui est fourni en paramètre au gestionnaire d'événement *BeforeUpdateRecord*.

Quand l'objet mise à jour n'a pas été affecté à la propriété *UpdateObject* de l'ensemble de données, ses instructions SQL ne sont pas automatiquement exécutées quand vous appelez *ApplyUpdates*. Pour mettre à jour les enregistrements, vous devez appeler manuellement l'objet mise à jour depuis un gestionnaire d'événement *OnUpdateRecord* (si vous utilisez le BDE pour placer les mises à jour en mémoire cache) ou *BeforeUpdateRecord* (si vous utilisez un ensemble de données client). Dans ce gestionnaire d'événement, vous devez au moins entreprendre les actions suivantes :

- Si vous utilisez un ensemble de données client pour placer les mises à jour en mémoire cache, vous devez vous assurer que les propriétés *DatabaseName* et *SessionName* des objets mise à jour sont définies comme les propriétés *DatabaseName* et *SessionName* de l'ensemble de données source.
- Le gestionnaire d'événement doit appeler la méthode *ExecSQL* ou *Apply* de l'objet mise à jour. Cela invoque l'objet mise à jour pour chaque enregistrement nécessitant une mise à jour. Pour plus d'informations sur

l'exécution des instructions de mise à jour, voir "Exécution des instructions SQL" ci-dessous.

- Affectez au paramètre *UpdateAction* du gestionnaire d'événement la valeur *uaApplied* (*OnUpdateRecord*) ou au paramètre *Applied* la valeur *True* (*BeforeUpdateRecord*).

Si vous le désirez, vous pouvez effectuer une validation ou une modification des données, ou d'autres opérations qui dépendent de la mise à jour de chaque enregistrement.

Attention Si vous appelez la méthode *ExecSQL* ou *Apply* d'un objet mise à jour dans un gestionnaire d'événement *OnUpdateRecord*, assurez-vous de ne pas définir la propriété *UpdateObject* de l'ensemble de données à cet objet mise à jour. Sinon, le résultat sera une seconde tentative d'application de la mise à jour de chaque enregistrement.

Exécution des instructions SQL

Quand vous utilisez plusieurs objets mise à jour, vous ne les associez pas à un ensemble de données par sa propriété *UpdateObject*. En conséquence, les instructions appropriées ne sont pas automatiquement exécutées quand vous appliquez les mises à jour. A la place, vous devez invoquer explicitement l'objet mise à jour dans le code.

Il y a deux façons d'invoquer l'objet mise à jour. Celle que vous choisissez varie selon que l'instruction SQL utilise ou non des paramètres pour représenter les valeurs des champs:

- Si l'instruction SQL à exécuter utilise des paramètres, appelez la méthode *Apply*.
- Si l'instruction SQL à exécuter n'utilise pas de paramètre, il vaut mieuxappeler la méthode *ExecSQL*.

Remarque Si l'instruction SQL utilise des paramètres autres que des types intégrés (pour les valeurs de champs originales et mises à jour), vous devez fournir manuellement les valeurs de paramètres au lieu de vous appuyer sur la substitution de paramètre fournie par la méthode *Apply*. Consultez "Utilisation de la propriété Query d'un composant mise à jour" à la page 20-54 pour plus d'informations sur la fourniture manuelle des valeurs de paramètres.

Pour plus d'informations sur la substitution de paramètre par défaut dans les instructions SQL de mise à jour, voir "Substitution de paramètres dans les instructions SQL de mise à jour" à la page 20-48.

Appel de la méthode Apply

La méthode *Apply* d'un composant mise à jour applique manuellement les mises à jour pour l'enregistrement en cours. Le processus comprend deux étapes :

1. Les valeurs de champs initiale et modifiée de l'enregistrement sont liées aux paramètres de l'instruction SQL approprié.
2. L'instruction SQL est exécutée.

Appelez la méthode *Apply* pour appliquer la mise à jour de l'enregistrement en cours dans la mémoire cache de mise à jour. *Apply* est souvent appelée depuis un gestionnaire d'événement *OnUpdateRecord* de l'ensemble de données ou depuis un gestionnaire d'événement *BeforeUpdateRecord* d'un fournisseur.

Attention Si vous utilisez la propriété *UpdateObject* de l'ensemble de données pour associer ensemble de données et objet mise à jour, *Apply* est appelée automatiquement. Dans ce cas, n'appelez pas *Apply* dans un gestionnaire d'événement *OnUpdateRecord* pour éviter une seconde tentative d'application de la mise à jour de l'enregistrement en cours.

Les gestionnaires d'événements *OnUpdateRecord* indiquent le type de mise à jour qui doit être appliqué avec un paramètre *UpdateKind* de type *TUpdateKind*. Vous devez passer ce paramètre à la méthode *Apply* pour indiquer l'instruction SQL à utiliser. Le code suivant illustre l'utilisation d'un gestionnaire d'événement *BeforeUpdateRecord* :

```
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
          DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  with UpdateSQL1 do
  begin
    DataSet := DeltaDS;
    DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
    SessionName := (SourceDS as TDBDataSet).SessionName;
    Apply(UpdateKind);
    Applied := True;
  end;
end;
```

Appel de la méthode ExecSQL

La méthode *ExecSQL* d'un composant mise à jour applique manuellement les mises à jour relatives à l'enregistrement en cours. A la différence de la méthode *Apply*, *ExecSQL* ne lie pas les paramètres dans l'instruction SQL avant de l'exécuter. La méthode *ExecSQL* est appelée le plus souvent depuis un gestionnaire d'événement *OnUpdateRecord* (utilisation du BDE) ou *BeforeUpdateRecord* (utilisation d'un ensemble de données client).

Comme *ExecSQL* ne lie pas les valeurs de paramètres, elle est utilisée principalement quand les instructions SQL de l'objet mise à jour ne comprennent pas de paramètres. Vous pouvez utiliser à la place *Apply*, même s'il n'y a pas de paramètres, mais *ExecSQL* est plus efficace car elle ne vérifie pas les paramètres.

Si les instructions SQL comprennent des paramètres, vous pouvez tout de même appeler *ExecSQL*, mais seulement après avoir lié explicitement les paramètres. Si vous utilisez le BDE pour mettre les mises à jour en mémoire cache, vous pouvez lier explicitement les paramètres en définissant la propriété *DataSet* de l'objet mise à jour, puis en appelant sa méthode *SetParams*. Si vous utilisez un ensemble de données client, vous devez fournir les paramètres à l'objet requête sous-jacent maintenu par *TUpdateSQL*. Pour plus d'informations sur la façon de procéder, voir "Utilisation de la propriété Query d'un composant mise à jour" à la page 20-54.

Attention Si vous utilisez la propriété *UpdateObject* de l'ensemble de données pour associer ensemble de données et objet mise à jour, *ExecSQL* est appelée automatiquement. Dans ce cas, n'appelez pas *ExecSQL* dans un gestionnaire d'événement *OnUpdateRecord* ou *BeforeUpdateRecord* pour éviter une seconde tentative d'application de la mise à jour de l'enregistrement en cours.

Les gestionnaires d'événements *OnUpdateRecord* et *BeforeUpdateRecord* indiquent le type de mise à jour qui doit être appliqué avec un paramètre *UpdateKind* de type *TUpdateKind*. Vous devez passer ce paramètre à la méthode *ExecSQL* pour indiquer l'instruction SQL à utiliser. Le code suivant illustre ceci avec un gestionnaire d'événement *BeforeUpdateRecord* :

```
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
          DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  with UpdateSQL1 do
  begin
    DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
    SessionName := (SourceDS as TDBDataSet).SessionName;
    ExecSQL(UpdateKind);
    Applied := True;
  end;
end;
```

Si une exception est déclenchée durant l'exécution du programme de mise à jour, l'exécution continue dans l'événement *OnUpdateError*, s'il est défini.

Utilisation de la propriété Query d'un composant mise à jour

La propriété *Query* d'un composant mise à jour fournit l'accès aux composants requête qui implémentent ses instructions *DeleteSQL*, *InsertSQL* et *ModifySQL*. Dans la plupart des applications, il n'est pas nécessaire d'accéder directement à ces composants requêtes : vous pouvez utiliser les propriétés *DeleteSQL*, *InsertSQL* et *ModifySQL* pour spécifier les instructions que ces requêtes exécutent, et les exécuter en appelant la méthode *Apply* ou *ExecSQL* de l'objet mise à jour. Il y a des cas, cependant, où vous pouvez avoir besoin de manipuler directement le composant requête. En particulier, la propriété *Query* est utile si vous voulez fournir vos propres valeurs aux paramètres des instructions SQL, plutôt que de vous appuyer sur la liaison automatique de paramètres de l'objet mise à jour aux anciennes et nouvelles valeurs de champs.

Remarque La propriété *Query* n'est accessible qu'à l'exécution.

La propriété *Query* est indexée sur une valeur *TUpdateKind* :

- Avec l'index *ukModify*, elle accède à la requête qui met à jour des enregistrements existants.
- Avec l'index *ukInsert,* elle accède à la requête qui insère de nouveaux enregistrements.
- Avec l'index *ukDelete,* elle accède à la requête qui supprime des enregistrements.

L'exemple suivant montre comment utiliser la propriété *Query* pour fournir des valeurs de paramètres qui ne peuvent être liées automatiquement :

```
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
          DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  UpdateSQL1.DataSet := DeltaDS; { requis pour la substitution de paramètres automatique }
  with UpdateSQL1.Query[UpdateKind] do
  begin
    { veillez que DatabaseName et SessionName sont corrects pour la requête }
    DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
    SessionName := (SourceDS as TDBDataSet).SessionName;
    ParamByName('TimeOfUpdate').Value = Now;
  end;
  UpdateSQL1.Apply(UpdateKind); { effectue maintenant les substitutions automatiques }
  Applied := True;
end;
```

Utilisation de TBatchMove

TBatchMove encapsule les fonctionnalités du moteur de bases de données (BDE) qui permettent de dupliquer un ensemble de données, d'ajouter à un ensemble de données des enregistrements d'un autre, de mettre à jour des enregistrements d'un ensemble de données avec ceux d'un autre, et de supprimer dans un ensemble de données les enregistrements qui correspondent à ceux d'un autre ensemble de données. *TBatchMove* sert le plus souvent à :

- Charger les données d'un serveur dans une source de données locale afin de les analyser ou d'effectuer d'autres opérations.
- Transférer, dans le cadre d'une opération d'upsizing, une base de données de bureau dans des tables stockées sur un serveur distant.

Un composant action groupée peut créer sur la destination des tables correspondant aux tables source, en établissant une correspondance automatique entre les noms de colonnes et les types de données.

Création d'un composant action groupée

Pour créer un composant action groupée :

1. Placez sur une fiche ou dans un module de données un composant table ou requête d'un ensemble de données depuis lequel vous voulez importer des enregistrements (appelé ensemble de données *Source*).
2. Placez sur une fiche ou dans un module de données l'ensemble de données dans lequel vous voulez déplacer les enregistrements (appelé ensemble de données *Destination*).
3. Placez un composant *TBatchMove* depuis la page BDE de la palette des composants dans le module de données ou sur la fiche, et définissez sa propriété *Name*à une valeur unique appropriée à votre application.

4. Affectez à la propriété *Source* du composant action groupée le nom de la table depuis laquelle copier, ajouter ou mettre à jour des enregistrements. Vous pouvez sélectionner les tables dans les listes déroulantes des composants ensemble de données disponibles.
5. Affectez à la propriété *Destination* le nom de l'ensemble de données à créer, à compléter ou à mettre à jour. Vous pouvez sélectionner une table de destination dans les listes déroulantes des composants ensemble de données disponibles.
   - Si vous ajoutez, mettez à jour ou supprimez des enregistrements, *Destination* doit représenter une table de base de données existante.
   - Si vous copiez une table et si *Destination* représente une table existante, l'exécution de l'action groupée écrase toutes les données en cours de la table de destination.
   - Si vous créez une table entièrement nouvelle en copiant une table existante, la table résultante possède le nom spécifié dans la propriété *Name* du composant table vers lequel vous copiez les informations. Le type de la table obtenu présente une structure appropriée au serveur spécifié par la propriété *DatabaseName*.
6. Définissez la propriété *Mode* pour indiquer le type d'opération à effectuer. Les opérations possibles sont *batAppend* (valeur par défaut), *batUpdate*, *batAppendUpdate*, *batCopy* et *batDelete*. Pour plus d'informations sur ces modes, voir "Spécification d'un mode d'action groupée" à la page 20-57.
7. Facultativement, définissez la propriété *Transliterate*. Si la valeur de *Transliterate* est *True* (valeur par défaut), les données caractère sont transcrites du jeu de caractères de l'ensemble de données *Source* vers celui de l'ensemble de données *Destination*, si nécessaire.
8. Facultativement, définissez une association entre les colonnes à l'aide de la propriété *Mappings*. Il n'est pas nécessaire de le faire si vous voulez que l'action groupée associe chaque colonne en fonction de sa position dans les tables source et destination. Pour plus d'informations sur les correspondances de colonnes, voir "Mappage des types de données" à la page 20-58.
9. Facultativement, définissez les propriétés *ChangedTableName*, *KeyViolTableName* et *ProblemTableName*. L'action groupée stocke les enregistrements posant problème dans la table spécifiée par *ProblemTableName*. Si vous mettez à jour une table Paradox par une opération action groupée, les violations de clés seront signalées dans la table spécifiée par *KeyViolTableName*. *ChangedTableName* liste tous les enregistrements modifiés dans la table destination lors de l'action groupée. Si vous ne spécifiez pas ces propriétés, les tables d'erreurs ne sont ni créées, ni utilisées. Pour plus d'informations sur la gestion des erreurs relatives aux actions groupées, voir "Gestion des erreurs relatives aux actions groupées" à la page 20-60.

Spécification d'un mode d'action groupée

La propriété *Mode* spécifie l'opération réalisée par un composant action groupée :

Tableau 20.8 Modes relatifs aux actions groupées

Propriété	Signification
batAppend	Ajoute les enregistrements à la table destination.
batUpdate	Met à jour les enregistrements de la table destination avec les enregistrements correspondants de la table source. La mise à jour est basée sur l'index en cours dans la table destination.
batAppendUpdate	Si un enregistrement correspondant existe dans la table destination, une mise à jour est effectuée. Sinon, les enregistrements sont ajoutés à la table destination.
batCopy	Crée la table destination à partir de la structure de la table source. Si la table destination existe déjà, elle est supprimée et recréée.
batDelete	Supprime les enregistrements de la table destination ayant une correspondance dans la table source.

Ajout d'enregistrements

Pour ajouter des données, l'ensemble de données destination doit représenter une table existante. Durant l'opération d'ajout, le BDE convertit si nécessaire les données à des tailles et des types de données appropriés à l'ensemble de données destination. Si la conversion n'est pas possible, une exception est déclenchée et les données ne sont pas ajoutées.

Mise à jour d'enregistrements

Pour mettre à jour les données, l'ensemble de données destination doit représenter une table existante et avoir un index qui permet d'établir la correspondance entre les enregistrements. Si les champs de l'index primaire sont utilisés pour établir les correspondances, les enregistrements pour lesquels des champs indexés dans l'ensemble de données destination correspondent à des champs indexés dans l'ensemble de données source sont écrasés avec les données source. Si nécessaire, pendant l'opération de mise à jour, le BDE convertit les données à des tailles et des types de données appropriés à l'ensemble de données destination.

Ajout et mise à jour d'enregistrements

Pour ajouter et mettre à jour des données, l'ensemble de données destination doit représenter une table existante et doit avoir un index qui permet d'établir la correspondance entre les enregistrements. Si les champs de l'index primaire sont utilisés pour établir les correspondances, les enregistrements pour lesquels des champs indexés dans l'ensemble de données destination correspondent à des champs indexés dans l'ensemble de données source sont écrasés avec les données source. Sinon, les données de l'ensemble de données source sont ajoutées à l'ensemble de données destination. Durant les opérations d'ajout et de mise à

jour, le BDE convertit si nécessaire les données à des tailles et des types de données appropriés à l'ensemble de données destination.

Copie d'ensembles de données

Pour copier un ensemble de données source, l'ensemble de données destination ne doit pas représenter une table existante. Si c'est le cas, l'opération action groupée écrase la table existante avec une copie de l'ensemble de données source.

Si les ensembles de données source et destination sont gérés par des moteurs de bases de données de types différents, par exemple Paradox et InterBase, le BDE crée un ensemble de données destination avec une structure aussi proche que possible de celle de l'ensemble de données source et effectue automatiquement les conversions de taille et de type de données en cas de besoin.

Remarque *TBatchMove* ne copie pas les structures de métadonnées, comme les index, les contraintes et les procédures stockées. Vous devez recréer ces objets métadonnées sur votre serveur de bases de données ou utiliser l'explorateur SQL.

Suppression d'enregistrements

Pour supprimer des données dans l'ensemble de données destination, celui-ci doit représenter une table existante et doit avoir un index qui permet d'établir la correspondance entre les enregistrements. Si les champs de l'index primaire sont utilisés pour établir les correspondances, les enregistrements pour lesquels des champs indexés dans l'ensemble de données destination correspondent à des champs indexés dans l'ensemble de données source sont supprimés dans la table destination.

Mappage des types de données

En mode *batAppend*, un composant action groupée crée la table destination à partir des types de données des colonnes de la table source. Les colonnes et les types de données sont mappés à partir de leur position dans les tables source et destination. Ainsi, la première colonne de la source est mappée avec la première colonne de la destination et ainsi de suite.

Pour outrepasser le mappage par défaut des colonnes, utilisez la propriété *Mappings*. *Mappings* est une liste de mappages de colonnes (un par ligne). Cette liste peut prendre deux formes. Pour mapper une colonne de la table source sur une colonne de même nom dans la table destination, vous pouvez utiliser une liste simple qui spécifie le nom de colonne à faire correspondre. Par exemple, le mappage suivant spécifie qu'une colonne nommée *ColName* de la table source doit être mappée sur une colonne de même nom dans la table destination :

```
ColName
```

Pour mapper une colonne nommée *SourceColName* de la table source sur une colonne nommée *DestColName* dans la table destination, la syntaxe est la suivante :

```
DestColName = SourceColName
```

Si les types de données des colonnes source et destination ne sont pas les mêmes, l'opération action groupée tente de les traduire au mieux. Elle tronque les types de données caractère, si nécessaire, et essaie d'effectuer une conversion limitée, si possible. Par exemple, mapper une colonne CHAR(10) sur une colonne CHAR(5) perd les cinq derniers caractères de la colonne source.

A titre d'exemple de conversion, si une colonne source de type caractère est mappée sur une colonne de type entier, l'opération action groupée convertit une valeur caractère '5' à la valeur entière 5 correspondante. Les valeurs qui ne peuvent pas être converties génèrent des erreurs. Pour plus d'informations sur les erreurs, voir "Gestion des erreurs relatives aux actions groupées" à la page 20-60.

Lorsque vous déplacez des données entre des tables de types différents, un composant action groupée traduit les types de données conformément aux types de serveurs de l'ensemble de données. Consultez l'aide en ligne du BDE pour obtenir les dernières tables de mappage entre types de serveurs.

Remarque Pour exécuter une action groupée sur des données en direction d'une base de données sur un serveur SQL, vous devez avoir installé ce serveur de bases de données et une version de Delphi proposant le pilote SQL Link approprié, ou utiliser ODBC si les pilotes ODBC adéquats des tiers sont installés.

Exécution d'une action groupée

Utilisez la méthode *Execute* pour exécuter une opération groupée préalablement préparée à l'exécution. Par exemple, si *BatchMoveAdd* est le nom d'un composant action groupée, l'instruction suivante l'exécute :

```
BatchMoveAdd.Execute;
```

Vous pouvez aussi exécuter une action groupée à la conception en effectuant un clic droit sur un composant action groupée et en choisissant Exécuter dans le menu contextuel.

La propriété *MovedCount* garde en mémoire le nombre d'enregistrements déplacés lors de l'exécution d'une action groupée.

La propriété *RecordCount* spécifie le nombre maximal d'enregistrements à déplacer. Si *RecordCount* est à zéro, tous les enregistrements sont déplacés, en commençant par le premier enregistrement de l'ensemble de données source. Si *RecordCount* est un nombre positif, un maximum de *RecordCount* enregistrements est déplacé, en commençant par l'enregistrement en cours de l'ensemble de données source. Si *RecordCount* est supérieur au nombre d'enregistrements situés entre l'enregistrement en cours et le dernier enregistrement, l'opération se termine lorsque la fin de l'ensemble de données source est atteinte. Vous pouvez examiner *MoveCount* pour déterminer combien d'enregistrements ont été effectivement transférés.

Gestion des erreurs relatives aux actions groupées

Deux types d'erreurs peuvent survenir dans une opération action groupée : les erreurs de conversion de type de données et les violations d'intégrité. *TBatchMove* possède plusieurs propriétés indiquant comment ces erreurs doivent être traitées.

La propriété *AbortOnProblem* spécifie s'il faut arrêter l'opération quand une erreur de conversion de type de données survient. Si *AbortOnProblem* est à *True*, l'opération est annulée si une erreur survient. Si elle est à *False*, l'opération continue. Vous pouvez examiner la table spécifiée dans la propriété *ProblemTableName* pour déterminer les enregistrements ayant posé problème.

La propriété *AbortOnKeyViol* indique s'il faut arrêter l'opération quand une violation de clé Paradox survient.

La propriété *ProblemCount* indique le nombre d'enregistrements qui n'ont pas pu être gérés dans la table destination sans perte de données. Si *AbortOnProblem* est à *True*, ce nombre vaut 1, puisque l'opération est arrêtée quand une erreur survient.

Les propriétés suivantes permettent à un composant action groupée de créer des tables supplémentaires documentant l'opération effectuée :

- *ChangedTableName*, si spécifiée, crée une table Paradox locale contenant tous les enregistrements de la table destination qui ont été modifiés suite à des opérations impliquant une suppression ou des mises à jour.
- *KeyViolTableName*, si spécifiée, crée une table Paradox locale contenant tous les enregistrements de la table source ayant généré une violation de clé lors de la manipulation d'une table Paradox. Si *AbortOnKeyViol* est à *True*, cette table contiendra au plus une entrée, puisque l'opération est arrêtée au premier problème rencontré.
- *ProblemTableName*, si spécifiée, crée une table Paradox locale contenant tous les enregistrements n'ayant pu être validés dans la table destination à cause d'erreurs de conversion de type de données. Par exemple, la table pourrait contenir les enregistrements de la table source dont les données ont du être tronquées pour tenir dans la table destination. Si *AbortOnKeyViol* est à *True*, cette table contiendra au plus un enregistrement puisque l'opération est arrêtée au premier problème rencontré.

Remarque Si *ProblemTableName* n'est pas spécifiée, les données de l'enregistrement sont tronquées et placées dans la table destination.

Dictionnaire de données

Quand vous utilisez le BDE pour accéder aux données, votre application doit accéder au dictionnaire de données. Celui-ci fournit une zone de stockage personnalisable, indépendante de vos applications, où vous pouvez créer des

ensembles d'attributs de champs étendus qui décrivent le contenu et l'apparence des données.

Par exemple, si vous développez fréquemment des applications financières, vous pouvez créer des ensembles d'attributs de champs spécialisés décrivant différents formats d'affichage monétaire. Quand vous créez des ensembles de données pour votre application à la conception, plutôt que d'utiliser l'inspecteur d'objets pour définir manuellement les champs monétaires de chaque ensemble de données, vous pouvez associer ces champs à un ensemble d'attributs de champs étendus dans le dictionnaire de données. L'usage de ce dernier assure une apparence homogène des données dans et entre les applications que vous créez.

En environnement client/serveur, le dictionnaire de données peut résider sur un serveur distant pour un partage supplémentaire des informations.

Pour apprendre comment créer des ensembles d'attributs de champs étendus dans l'éditeur de champs à la conception et comment les associer aux champs des ensembles de données de votre application, voir "Création des ensembles d'attributs pour les composants champ" à la page 19-15. Pour plus d'informations sur la création d'un dictionnaire de données et sur les attributs de champs étendus avec les explorateurs SQL et de base de données, consultez leurs aides en ligne respectives.

Une interface de programmation pour le dictionnaire de données est disponible dans l'unitédrintf (situé dans le répertoire lib). Cette interface fournit les méthodes suivantes :

Tableau 20.9 Interface du dictionnaire de données

Routine	Utilisation
DictionaryActive	Indique si le dictionnaire de données est actif.
DictionaryDeactivate	Désactive le dictionnaire de données.
IsNullID	Indique si un identificateur donné est un identificateur null.
FindDatabaseID	Renvoie l'identificateur d'une base de données d'après son alias.
FindTableID	Renvoie l'identificateur d'une table de la base de données spécifiée.
FindFieldID	Renvoie l'identificateur d'un champ de la table spécifiée.
FindAttrID	Renvoie l'identificateur d'un ensemble d'attributs nommé.
GetAttrName	Renvoie le nom d'un ensemble d'attributs d'après son identificateur.
GetAttrNames	Exécute un callback pour chaque ensemble d'attributs du dictionnaire.
GetAttrID	Renvoie l'identificateur d'un ensemble d'attributs pour le champ spécifié.
NewAttr	Crée un nouvel ensemble d'attributs à partir d'un composant champ.
UpdateAttr	Met à jour un ensemble d'attributs pour correspondre aux propriétés d'un champ.
CreateField	Crée un composant champ d'après des attributs stockés.

Tableau 20.9 Interface du dictionnaire de données (suite)

Routine	Utilisation
UpdateField	Change les propriétés d'un champ pour qu'elles correspondent à l'ensemble d'attributs spécifié.
AssociateAttr	Associe un ensemble d'attributs à un identificateur de champ donné.
UnassociateAttr	Supprime une association d'ensemble d'attributs pour un identificateur de champ.
GetControlClass	Renvoie la classe de contrôle pour un identificateur d'attribut spécifié.
QualifyTableName	Renvoie un nom de table qualifié (par le nom de l'utilisateur).
QualifyTableNameByName	Renvoie un nom de table qualifié (par le nom de l'utilisateur).
HasConstraints	Indique si l'ensemble de données a des contraintes dans le dictionnaire.
UpdateConstraints	Met à jour les contraintes importées d'un ensemble de données.
UpdateDataset	Applique à un ensemble de données les contraintes et les paramètres en cours du dictionnaire.

Outils de manipulation du BDE

L'un des avantages de l'utilisation du BDE comme mécanisme d'accès aux données est la force du support fourni avec Delphi. Ces utilitaires comprennent :

- **L'explorateur SQL et l'explorateur de base de données** : Delphi est livré avec une de ces deux applications, selon la version que vous avez achetée. Les deux explorateurs vous permettent :
  - D'examiner les structures et les tables des bases de données existantes. L'explorateur SQL vous permet d'examiner et d'interroger les bases de données SQL distantes.
  - De remplir les tables de données.
  - De créer des ensembles d'attributs de champs étendus dans le dictionnaire de données ou de les associer aux champs de votre application.
  - De créer et de gérer des alias BDE.

  L'explorateur SQL vous permet aussi :
  - De créer des objets SQL tels que des procédures stockées sur les serveurs de bases de données distants.
  - D'afficher le texte reconstruit des objets SQL sur les serveurs de bases de données distants.
  - De lancer des scripts SQL.
- **Le moniteur SQL** : Il vous permet de surveiller toutes les communications qui passent entre le serveur de base de données distant et le BDE. Vous pouvez filtrer les messages que vous surveillez, en vous limitant aux catégories qui

vous intéressent. Le moniteur SQL est très utile pour déboguer votre application.

- **L'utilitaire d'administration BDE :** Il vous permet d'ajouter de nouveaux pilotes de bases de données, de configurer les valeurs par défaut des pilotes existants et de créer de nouveaux alias BDE.
- **Le module base de données** : Si vous utilisez des tables Paradox ou dBASE, ce module vous permet d'afficher et d'éditer leurs données, de créer de nouvelles tables et de restructurer les tables existantes. Son emploi vous donne plus de contrôle que les méthodes d'un composant *TTable* (par exemple, il vous permet de spécifier des contrôles de validité et les pilotes de langage). Il fournit le seul mécanisme de restructuration des tables Paradox et dBASE en dehors des appels directs à l'API du BDE.

Chapitre

21

Utilisation des composants ADO

Les composants *ADOExpress* permettent d'accéder aux données par le biais du modèle ADO. ADO (Microsoft ActiveX Data Objects) est un ensemble d'objets COM qui accèdent aux données par le biais d'un fournisseur OLE DB. Les composants *ADOExpress* Delphi encapsulent ces objets ADO dans l'architecture de base de données Delphi.

La couche ADO d'une application ADO comprend Microsoft ADO 2.1, un fournisseur OLE DB ou un pilote ODBC pour l'accès au stockage de données, le logiciel client propre au système de base de données utilisé (dans le cas des bases de données SQL), un système de base de données dorsal accessible à l'application (dans le cas des systèmes de bases de données SQL) et une base de données. Tous ces éléments doivent être accessibles à l'application ADO pour que celle-ci soit totalement opérationnelle.

Les objets ADO essentiels sont les objets Connection, Command et Recordset. Ces objets ADO sont encapsulés par les composants *TADOConnection*, *TADOCommand* et ensemble de données ADO. Le modèle ADO comprend d'autres objets utilitaires, comme les objets Field ou Properties, mais ils ne sont généralement pas utilisés directement dans les applications Delphi et ne sont pas encapsulés par des composants dédiés.

Ce chapitre présente les composants *ADOExpress* et les fonctionnalités uniques qu'ils apportent à l'architecture de base de données Delphi commune. Avant de découvrir les fonctionnalités propres aux composants *ADOExpress*, vous devez connaître les fonctionnalités communes des composants connexion de base de données et des ensembles de données décrites dans les chapitres 17, "Connexion aux bases de données" et 18, "Présentation des ensembles de données".

Présentation générale des composants ADO

La page ADO de la palette de composants héberge les composants *ADOExpress*. Ces composants permettent de se connecter à un stockage de données ADO, d'exécuter des commandes et d'extraire des données de tables de bases de données utilisant le modèle ADO. Ils requièrent ADO version 2.1 (ou supérieure) sur l'ordinateur hôte. De plus, il faut que le logiciel client pour le système de bases de données cible (par exemple, Microsoft SQL Server) soit installé ainsi qu'un pilote OLE DB ou ODBC spécifique à ce système de bases de données.

La plupart des composants *ADOExpress* ont des homologues directs dans les composants disponibles pour d'autres mécanismes d'accès aux données : un composant connexion de base de données (*TADOConnection*) et différents types d'ensembles de données. En outre, *ADOExpress* comprend *TADOCommand*, simple composant qui n'est pas un ensemble de données mais qui représente une commande SQL à exécuter sur le stockage de données ADO.

Le tableau suivant présente les composants ADO.

Tableau 21.1 Composants ADO

Composant	Utilisation
TADOConnection	Composant connexion de base de données qui établit une connexion avec un stockage de données ADO ; des composants ensemble de données ou commande ADO peuvent partager cette connexion pour exécuter des commandes, obtenir des données ou agir sur les métadonnées.
TADODataSet	Ensemble de données de base utilisé pour obtenir et transformer les données ; *TADODataSet* peut obtenir les données d'une ou de plusieurs tables ; il peut se connecter directement à un stockage de données ou utiliser un composant *TADOConnection*.
TADOTable	Ensemble de données de type table utilisé pour l'obtention et la manipulation d'un ensemble d'enregistrements produit par une seule table de base de données ; *TADOTable* peut se connecter directement à un stockage de données ou utiliser un composant *TADOConnection*.
TADOQuery	Ensemble de données de type requête utilisé pour l'obtention et la manipulation d'un ensemble d'enregistrements produit par une instruction SQL valide ; *TADOQuery* peut également exécuter des instructions SQL DDL (langage de définition de données). Il peut se connecter directement à un stockage de données ou utiliser un composant *TADOConnection*.
TADOStoredProc	Ensemble de données de type procédure stockée utilisé pour l'exécution de procédures stockées ; *TADOStoredProc* exécute des procédures stockées qui peuvent ou non extraire des données. Il peut se connecter directement à un stockage de données ou utiliser un composant *TADOConnection*.
TADOCommand	Simple composant permettant d'exécuter des commandes (des instructions SQL ne produisant pas d'ensemble de résultats) ; *TADOCommand* peut être utilisé avec un composant ensemble de données associé ou obtenir un ensemble de données d'une table. Il peut se connecter directement à un stockage de données ou utiliser un composant *TADOConnection*.

Connexion à des stockages de données ADO

Delphi utilisent Microsoft ActiveX Data Objects (ADO) 2.1 pour interagir avec un fournisseur OLE DB qui se connecte à un stockage de données et accède à ses données. L'un des éléments qu'un stockage de données peut représenter est une base de données. Une application ADO n'est opérationnelle que si ADO 2.1 est installé sur l'ordinateur client. ADO et OLE DB sont fournis par Microsoft et installés avec Windows.

Un fournisseur ADO représente l'un des nombreux types d'accès, allant des pilotes OLE DB natifs aux pilotes ODBC. Ces pilotes doivent être installés sur l'ordinateur client. Les pilotes OLE DB des différents systèmes de bases de données sont fournis par l'éditeur de bases de données ou par une tierce partie. Si l'application utilise une base de données SQL, telle que Microsoft SQL Server ou Oracle, le logiciel client de ce système de base de données doit également être installé sur l'ordinateur client. Le logiciel client est fourni par l'éditeur de bases de données et installé à partir du CD-ROM (ou de la disquette) des systèmes de bases de données.

Pour connecter votre application au stockage de données, utilisez un composant connexion ADO (*TADOConnection*). Configurez le composant connexion ADO afin qu'il utilise l'un des fournisseurs ADO disponibles. Bien que *TADOConnection* ne soit pas obligatoirement requis, du fait que les composants ensemble de données et commande ADO peuvent directement établir des connections par le biais de leur propriété *ConnectionString*, vous pouvez utiliser *TADOConnection* pour partager une connexion parmi plusieurs composants ADO. Cela permet de réduire la consommation des ressources et de créer des transactions qui englobent plusieurs ensembles de données.

Comme les autres composants connexion de base de données, *TADOConnection* permet de réaliser les tâches suivantes :

- Contrôles des connexions
- Contrôle de la connexion au serveur
- Gestion des transactions
- Utilisation d'ensembles de données associés
- Envoi de commandes au serveur
- Obtention de métadonnées

Outre ces fonctionnalités communes à tous les composants connexion de base de données, *TADOConnection* offre :

- Un large éventail d'options permettant d'optimiser la connexion.
- La possibilité d'afficher la liste des objets commande utilisant la connexion.
- Des événements supplémentaires lors de la réalisation des tâches courantes.

Connexion à un stockage de données avec TADOConnection

Un ou plusieurs composants commande et ensemble de données ADO peuvent utiliser une même connexion à un stockage de données grâce à *TADOConnection*. Pour ce faire, associez les composants ensemble de données et commande au

composant connexion via leur propriété *Connection*. A la conception, sélectionnez le composant connexion souhaité dans la liste déroulante affichée par la propriété *Connection* dans l'inspecteur d'objet. A l'exécution, affectez la référence à la propriété *Connection*. Par exemple, la ligne suivante associe un composant *TADODataSet* à un composant *TADOConnection*.

```
ADODataSet1.Connection := ADOConnection1;
```

Le composant connexion représente un objet connexion ADO. Avant d'utiliser l'objet connexion pour établir une connexion, vous devez identifier le stockage de données auquel vous souhaitez vous connecter. Généralement, vous fournissez ces informations par le biais de la propriété *ConnectionString*. *ConnectionString* est une chaîne contenant un ou plusieurs paramètres de connexion nommés séparés par des points-virgules. Ces paramètres identifient le stockage de données en spécifiant le nom d'un fichier qui contient les informations de connection ou le nom d'un fournisseur ADO et une référence qui désigne le stockage de données. Utilisez les noms de paramètres prédéfinis suivants pour fournir ces informations :

Tableau 21.2 Modes de connexion ADO

Paramètre	Description
Provider	Le nom d'un fournisseur ADO local à utiliser pour la connexion.
Data Source	Le nom du stockage de données.
File name	Le nom d'un fichier contenant les informations de connexion.
Remote Provider	Le nom d'un fournisseur ADO qui réside sur une machine distante.
Remote Server	Le nom du serveur distant lors de l'utilisation d'un fournisseur distant.

Ainsi, une valeur classique de *ConnectionString* a la forme

```
Provider=MSDASQL.1;Data Source=MQIS
```

Remarque Il n'est pas nécessaire que les paramètres de connexion de *ConnectionString* incluent le paramètre *Provider* ou *Remote Provider* si vous spécifiez un fournisseur ADO à l'aide de la propriété *Provider*. De même, vous n'avez pas besoin de spécifier le paramètre *Data Source* si vous utilisez la propriété *DefaultDatabase*.

Outre les paramètres ci-dessus, *ConnectionString* peut comprendre tous les paramètres de connexion propres au fournisseur ADO spécifique que vous utilisez. Ces paramètres de connexion supplémentaires peuvent comprendre l'ID et le mot de passe utilisateur si vous souhaitez coder en dur les informations de connexion.

A la conception, vous pouvez utiliser l'éditeur de chaîne de connexion pour construire la chaîne de connexion en sélectionnant dans des listes les éléments de connexion (comme le fournisseur ou le serveur). Cliquez sur le bouton points de suspension de la propriété *ConnectionString* dans l'inspecteur d'objets pour lancer l'éditeur de chaîne de connexion, qui est un éditeur de propriété ActiveX fourni par ADO.

Après avoir spécifié la propriété *ConnectionString* (et, éventuellement, la propriété *Provider*), vous pouvez utiliser le composant connexion ADO pour établir la connexion ou rompre la connexion au stockage de données ADO, à moins que

vous ne souhaitiez au préalable utiliser les autres propriétés pour optimiser la connexion. Lorsque vous établissez ou rompez la connexion au stockage de données, *TADOConnection* vous permet de répondre à quelques événements supplémentaires, outre ceux communs à tous les composants connexion de base de données. Ces événements supplémentaires sont décrits dans "Evénements se produisant pendant l'établissement d'une connexion" à la page 21-8 et "Evénements se produisant pendant la déconnexion" à la page 21-9.

Remarque Si vous n'activez pas explicitement la connexion en affectant la valeur *True* à la propriété *Connected* du composant connexion, la connexion est automatiquement établie quand le premier composant ensemble de données est ouvert ou la première fois que vous utilisez un composant commande ADO pour exécuter une commande.

Accès à l'objet connexion

Utilisez la propriété *ConnectionObject* de *TADOConnection* pour accéder à l'objet connexion ADO sous-jacent. En utilisant cette référence, vous pouvez accéder aux propriétés et appeler les méthodes de l'objet ADO Connection sous-jacent.

L'utilisation de l'objet Connection ADO sous-jacent nécessite une bonne connaissance pratique des objets ADO en général et de l'objet Connection ADO en particulier. Il n'est pas conseillé d'utiliser l'objet Connection à moins d'être familier des opérations sur les objets connexion. Reportez-vous à l'aide du SDK Microsoft Data Access pour des informations spécifiques sur l'utilisation des objets Connection ADO.

Optimisation d'une connexion

L'utilisation de *TADOConnection* pour établir la connexion à un stockage de données, au lieu de fournir simplement une chaîne de connexion pour les composants commande et ensemble de données ADO, présente l'avantage de procurer un contrôle accru des conditions et attributs de la connexion.

Connexions asynchrones

Utilisez la propriété *ConnectOptions* pour que la connexion soit asynchrone. L'utilisation de connexions asynchrones permet à votre application de poursuivre son traitement sans attendre leur ouverture définitive.

Par défaut, *ConnectionOptions* a la valeur *coConnectUnspecified* qui laisse le serveur décider du meilleur type de connexion. Pour imposer explicitement une connexion asynchrone, initialisez *ConnectOptions* à *coAsyncConnect*.

Les deux exemples de code suivants active et désactive des connexions asynchrones dans le composant connexion spécifié :

```
procedure TForm1.AsyncConnectButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    ConnectOptions := coAsyncConnect;
    Open;
```

```
  end;
end;

procedure TForm1.ServerChoiceConnectButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    ConnectOptions := coConnectUnspecified;
    Open;
  end;
end;
```

Contrôle des dépassements de délais

Vous pouvez contrôler la durée s'écoulant avant que les commandes ou les connexions tentées soient considérées comme ayant échoué et arrêtées avec les propriétés *ConnectionTimeout* et *CommandTimeout*.

ConnectionTimeout spécifie la durée, en seconde, s'écoulant avant le dépassement de délai pour établir la connexion avec un stockage de données. Si la connexion n'est pas correctement établie avant l'expiration du délai spécifié dans *ConnectionTimeout*, la tentative de connexion est abandonnée :

```
with ADOConnection1 do begin
  ConnectionTimeout := 10 {seconds};
  Open;
end;
```

CommandTimeout spécifie la durée, en seconde, s'écoulant avant le dépassement de délai pour une commande. Si la commande démarrée par un appel de la méthode *Execute* n'est pas achevée avant l'expiration du délai spécifié par *CommandTimeout*, la commande est annulée et ADO génère une exception.

```
with ADOConnection1 do begin
  CommandTimeout = 10 {secondes};
  Execute('DROP TABLE Employee1997', cmdText, []);
end;
```

Indication des types d'opérations pris en charge par la connexion

Les connexions ADO sont établies à l'aide d'un mode spécifique, similaire au mode que vous utilisez lorsque vous ouvrez un fichier. Le mode de connexion détermine les permissions disponibles pour la connexion et, par là-même, les types d'opérations (telles que la lecture et l'écriture) réalisables avec cette connexion.

Utilisez la propriété *Mode* pour indiquer le mode de connexion. Les valeurs possibles sont indiquées dans le tableau suivant :

Tableau 21.3 Modes de connexion ADO

Mode de connexion	Signification
cmUnknown	Les permissions ne sont pas encore définies pour la connexion ou ne peuvent être déterminées.
cmRead	Des permissions en lecture seule sont accessibles pour la connexion.

Tableau 21.3 Modes de connexion ADO (suite)

Mode de connexion	Signification
cmWrite	Des permissions en écriture seule sont accessibles pour la connexion.
cmReadWrite	Des permissions en lecture/écriture sont accessibles pour la connexion.
cmShareDenyRead	Empêche les autres utilisateurs d'ouvrir des connexions avec des droits en lecture.
cmShareDenyWrite	Empêche les autres utilisateurs d'ouvrir des connexions avec des droits en écriture.
cmShareExclusive	Empêche les autres utilisateurs d'ouvrir des connexions.
cmShareDenyNone	Empêche les autres utilisateurs d'ouvrir des connexions avec n'importe quelle permission.

Les valeurs possibles pour *Mode* correspondent aux valeurs *ConnectModeEnum* de la propriété *Mode* sur l'objet connexion ADO sous-jacent. Consultez l'aide SDK d'accès aux données de Microsoft pour plus d'informations sur ces valeurs.

Spécification de l'exécution automatique des transactions par la connexion

Utilisez la propriété *Attributes* pour contrôler l'utilisation par le composant de la conservation de la validation ou de l'annulation. Lorsque le composant connexion utilise la conservation des validations, chaque fois que votre application valide une transaction, une nouvelle transaction est automatiquement démarrée. Lorsque le composant connexion utilise la conservation des annulations, chaque fois que votre application annule une transaction, une nouvelle transaction est automatiquement démarrée.

Attributes est un ensemble qui peut contenir aucune, l'une et/ou l'autre des constantes *xaCommitRetaining* et *xaAbortRetaining*. Lorsque *Attributes* contient *xaCommitRetaining*, la connexion utilise la conservation des validations. Lorsque *Attributes* contient *xaAbortRetaining*, elle utilise la conservation des annulations.

Pour tester si l'une ou l'autre de ces caractéristiques est déjà définie, utilisez *l'opérateur in*. Activez la conservation des validations ou des annulations en ajoutant la valeur appropriée à la propriété Attributes ou retirez-les en soustrayant la valeur. Les exemples de code suivant active et désactive, respectivement, la conservation des validations dans un composant connexion ADO.

```
procedure TForm1.RetainingCommitsOnButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    if not (xaCommitRetaining in Attributes) then
      Attributes := (Attributes + [xaCommitRetaining])
    Open;
  end;
end;

procedure TForm1.RetainingCommitsOffButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
```

```
    if (xaCommitRetaining in Attributes) then
      Attributes := (Attributes - [xaCommitRetaining]);
    Open;
  end;
end;
```

Accès aux commandes d'une connexion

Comme avec les autres composants connexion de base de données, vous pouvez accéder aux ensembles de données associés à la connexion, à l'aide des propriétés *DataSets* et *DataSetCount*. Toutefois, *ADOExpress* comprend également des objets *TADOCommand*, qui ne sont pas des ensembles de données, mais qui gèrent une relation similaire avec le composant connexion.

Vous pouvez utiliser les propriétés *Commands* et *CommandCount* de *TADOConnection* pour accéder aux objets commande ADO associés de la même manière que les propriétés *DataSets* et *DataSetCount* pour accéder aux ensembles de données associés. A la différence de *DataSets* et *DataSetCount*, qui ne présentent que des ensembles de données actifs, *Commands* et *CommandCount* fournissent des références à tous les composants *TADOCommand* associés au composant connexion.

Commands est un tableau d'indice de base zéro de références à des composants commande ADO. *CommandCount* fournit le décompte de toutes les commandes répertoriées dans *Commands*. Vous pouvez utiliser ces propriétés conjointement pour parcourir toutes les commandes qui utilisent un composant connexion, comme l'illustre le code suivant *:*

```
var
  i: Integer
begin
  for i := 0 to (ADOConnection1.CommandCount - 1) do
    ADOConnection1.Commands[i].Execute;
end;
```

Evénements connexion ADO

Outre les événements habituels qui se produisent pour tous les composants connexion de base de données, *TADOConnection* génère une série d'événements supplémentaires qui interviennent pendant le déroulement normal des opérations.

Evénements se produisant pendant l'établissement d'une connexion

Outre les événements *BeforeConnect* et *AfterConnect*, communs à tous les composants connexion de base de données, *TADOConnection* génère les événements *OnWillConnect* et *OnConnectComplete* lors de l'établissement d'une connexion. Ces événements se produisent après l'événement *BeforeConnect*.

- *OnWillConnect* se produit avant que le fournisseur ADO n'établisse une connexion. Il vous permet d'apporter des modifications de dernière minute à

la chaîne de connexion, de fournir un nom et un mot de passe utilisateur si vous gérez votre propre support de connexion (login), d'imposer une connexion asynchrone ou même d'annuler la connexion avant son ouverture.

- *OnConnectComplete* se produit après l'ouverture de la connexion. Etant donné que *TADOConnection* peut représenter des connexions asynchrones, vous devez utiliser *OnConnectComplete*, qui se produit après l'ouverture de la connexion ou son échec en raison d'une condition d'erreur, au lieu de l'événement *AfterConnect*, qui se produit lorsque le composant connexion indique au fournisseur ADO d'ouvrir une connexion mais pas nécessairement après l'ouverture de la connexion.

Evénements se produisant pendant la déconnexion

Outre les événements *BeforeDisconnect* et *AfterDisconnect*, communs à tous les composants connexion de base de données, *TADOConnection* génère un événement *OnDisconnect* après la fermeture d'une connexion. *OnDisconnect* se produit après la fermeture de la connexion mais avant celle de tout ensemble de données associé et avant l'événement *AfterDisconnect*.

Evénements se produisant pendant la gestion des transactions

Les composants de connexion ADO proposent plusieurs événements pour gérer le cours d'un processus de transaction. Ces événements indiquent à quel moment un processus de transaction démarré par une méthode *BeginTrans*, *CommitTrans* et *RollbackTrans* a été correctement exécuté sur le stockage de données.

- L'événement *OnBeginTransComplete* se produit lorsque le stockage de données a correctement démarré une transaction après un appel de la méthode *BeginTrans*.
- L'événement *OnCommitTransComplete* se produit après la validation réussie d'une transaction à la suite de l'appel de *CommitTrans*.
- L'événement *OnRollbackTransComplete* se produit après l'annulation réussie d'une transaction à la suite de l'appel de *RollbackTrans*.

Autres événements

Les composants connexion ADO introduisent deux événements supplémentaires qui permettent de répondre aux notifications provenant de l'objet connexion ADO sous-jacent :

- L'événement *OnExecuteComplete* se produit après que le composant connexion exécute une commande sur le stockage de données (par exemple, après l'appel de la méthode *Execute*). *OnExecuteComplete* indique si l'exécution a réussi.
- L'événement *OnInfoMessage* se produit lorsque l'objet connexion sous-jacent fournit des informations détaillées après l'exécution d'une opération. Le gestionnaire d'événement *OnInfoMessage* reçoit l'interface d'un objet Error ADO qui contient les informations détaillées et un code d'état indiquant si l'opération a réussi.

Utilisation des ensembles de données ADO

Les composants ensemble de données ADO encapsulent l'objet Recordset ADO. Ils héritent des fonctionnalités d'ensemble de données communes décrites dans le chapitre 18, "Présentation des ensembles de données", et utilisent ADO pour fournir l'implémentation. L'utilisation d'un ensemble de données ADO suppose la connaissance de ces fonctionnalités communes.

Outre les fonctionnalités d'ensemble de données communes, tous les ensembles de données ADO ajoutent des propriétés, des événements et des méthodes pour effectuer les opérations suivantes :

- Connexion à un stockage de données ADO.
- Accès à l'objet Recordset sous-jacent.
- Filtrage d'enregistrements à partir de signets.
- Lecture d'enregistrements de façon asynchrone.
- Réalisation de mises à jour groupées (placement des mises à jour en mémoire cache).
- Utilisation de fichiers sur disque pour stocker les données.

Il existe quatre ensembles de données ADO :

- *TADOTable*, ensemble de données de type table qui représente toutes les lignes et colonnes d'une seule table de base de données. Voir "Utilisation d'ensembles de données de type table" à la page 18-29, pour plus d'informations sur l'utilisation de *TADOTable* et des autres ensembles de données de type table.
- *TADOQuery*, ensemble de données de type requête qui encapsule une instruction SQL et permet aux applications d'accéder aux enregistrements éventuellement obtenus. Voir "Utilisation d'ensembles de données de type requête" à la page 18-49, pour plus d'informations sur l'utilisation de *TADOQuery* et des autres ensembles de données de type requête.
- *TADOStoredProc*, ensemble de données de type procédure stockée qui exécute une procédure stockée définie sur un serveur de bases de données. Voir "Utilisation d'ensembles de données de type procédure stockée" à la page 18-58, pour plus d'informations sur l'utilisation de *TADOStoredProc* et des autres ensembles de données de type procédure stockée.
- *TADODataSet*, ensemble de données polyvalent qui comprend les fonctionnalités des trois autres types. Voir "Utilisation de TADODataSet" à la page 21-17, pour une description des fonctionnalités propres à *TADODataSet*.

Remarque Lorsque vous utilisez ADO pour accéder aux informations de bases de données, vous n'avez pas besoin de recourir à un ensemble de données tel que *TADOQuery* pour représenter les commandes SQL qui ne renvoient pas de curseur. Par contre, vous pouvez utiliser *TADOCommand*, simple composant qui n'est pas un ensemble de données. Pour plus d'informations sur *TADOCommand*, voir "Utilisation d'objets commande" à la page 21-19.

Connexion d'un ensemble de données ADO à un stockage de données

Il est possible de connecter individuellement ou collectivement des ensembles de données ADO à un stockage de données ADO.

Lorsque vous connectez des ensembles de données collectivement, attribuez à la propriété *Connection* de chaque ensemble de données un composant *TADOConnection*. Chaque ensemble de données utilise alors la connexion du composant connexion ADO.

```
ADODataSet1.Connection := ADOConnection1;
ADODataSet2.Connection := ADOConnection1;
...
```

Cette manière de procéder présente, entre autres, les avantages suivants :

- Les ensembles de données partagent les attributs de l'objet connexion.
- Il n'y a qu'une seule connexion à configurer : celle de l'objet *TADOConnection*.
- Les ensembles de données peuvent participer à des transactions.

Pour plus d'informations sur l'utilisation de *TADOConnection*, voir "Connexion à des stockages de données ADO" à la page 21-3.

Lorsque vous connectez des ensembles de données individuellement, définissez la propriété *ConnectionString* de chaque ensemble de données. Chaque ensemble de données qui utilise *ConnectionString* établit sa propre connexion avec le stockage de données, de manière indépendante des autres connexions d'ensembles de données de l'application.

La propriété *ConnectionString* des ensembles de données ADO fonctionne de la même façon que la propriété *ConnectionString* de *TADOConnection* ; il s'agit d'un ensemble de paramètres de connexion séparés par des points-virgules, tels que celui-ci :

```
ADODataSet1.ConnectionString := 'Provider=VotreFournisseur;Password=MotDePasse;' +
  'User ID=JaneDoe;SERVER=PURGATORY;UID=JaneDoe;PWD=MotDePasse;' +
  'Initial Catalog=Employee';
```

A la conception, vous pouvez utiliser l'éditeur de chaîne de connexion pour concevoir la chaîne de connexion. Pour plus d'informations sur les chaînes de connexion, voir "Connexion à un stockage de données avec TADOConnection" à la page 21-3.

Utilisation des ensembles d'enregistrements

La propriété *Recordset* donne un accès direct à l'objet ensemble d'enregistrements ADO sous-jacent au composant ensemble de données. En utilisant cet objet, il est possible dans une application d'accéder aux propriétés et d'appeler les méthodes de l'objet ensemble d'enregistrements. L'utilisation de *Recordset* pour accéder directement à l'objet ensemble d'enregistrements ADO sous-jacent suppose une bonne connaissance des objets ADO en général et de l'objet ensemble d'enregistrements ADO en particulier. L'utilisation directe de l'objet ensemble d'enregistrements n'est pas conseillée à moins d'être familiarisé avec sa manipulation. Consultez l'aide du SDK Microsoft Data Access pour des informations spécifiques sur les objets ensemble d'enregistrements.

La propriété *RecordsetState* indique l'état en cours de l'objet ensemble d'enregistrements sous-jacent. *RecordsetState* correspond à la propriété *State* de l'objet ensemble d'enregistrements ADO. *RecordsetState* a pour valeur *stOpen, stExecuting* ou *stFetching*. (*TObjectState*, qui est le type de la propriété *RecordsetState*, définit d'autres valeurs, mais seuls *stOpen*, *stExecuting* et *stFetching* appartiennent aux ensembles d'enregistrements). La valeur *stOpen* indique que l'ensemble d'enregistrements est actuellement inactif. La valeur *stExecuting* indique qu'il est en train d'exécuter une commande. La valeur *stFetching* indique qu'il est en train de lire des lignes dans les tables associées.

Utilisez les valeurs *RecordsetState* quand vous effectuez des actions qui dépendent de l'état en cours de l'ensemble de données. Ainsi, une routine qui actualise les données peut tester la valeur de la propriété *RecordsetState* pour vérifier si l'ensemble de données est actif et n'est pas en train de traiter d'autres opérations ou de lire des données.

Filtrage d'enregistrements à partir de signets

Les ensembles de données ADO prennent en charge la fonctionnalité d'ensemble de données commune consistant àutiliser des signets pour marquer et retrouver des enregistrements spécifiques. En outre, à l'image des autres ensembles de données, un ensemble de données ADO vous permet d'utiliser des filtres pour limiter les enregistrements disponibles dans l'ensemble de données. Les ensembles de données ADO offrent une fonctionnalité supplémentaire qui combine ces deux fonctionnalités d'ensemble de données communes : la possibilité de filtrer un ensemble d'enregistrements identifié par des signets.

Pour filtrer un ensemble de signets,

1. Utilisez la méthode *Bookmark* pour marquer les enregistrements à inclure dans l'ensemble de données filtré.
2. Appelez la méthode *FilterOnBookmarks* pour filtrer l'ensemble de données afin que seuls les enregistrements marqués d'un signet apparaissent.

Ce processus est illustré ci-dessous :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  BM1, BM2: TBookmarkStr;
begin
  with ADODataSet1 do begin
    BM1 := Bookmark;
    BMList.Add(Pointer(BM1));
    MoveBy(3);
    BM2 := Bookmark;
    BMList.Add(Pointer(BM2));
    FilterOnBookmarks([BM1, BM2]);
  end;
end;
```

Notez que l'exemple ci-dessus ajoute les signets à un objet liste nommé BMList. Cela permettra à l'application de libérer les signets lorsqu'ils ne seront plus nécessaires.

Pour plus d'informations sur l'utilisation des signets, voir "Marquage d'enregistrements" à la page 18-11. Pour des détails sur les autres types de filtres, voir "Affichage et édition d'ensembles de données en utilisant des filtres" à la page 18-14.

Lecture d'enregistrements de façon asynchrone

A la différence des autres ensembles de données, les ensembles de données ADO peuvent lire leurs données de façon asynchrone. Cela permet à votre application de poursuivre l'exécution d'autres tâches pendant le remplissage de l'ensemble de données avec des données du stockage de données.

Pour déterminer si l'ensemble de données lit les données de façon asynchrone, utilisez la propriété *ExecuteOptions*. *ExecuteOptions* régit la lecture des enregistrements par l'ensemble de données lorsque vous appelez *Open* ou attribuez à *Active* la valeur *True*. Si l'ensemble de données représente une requête ou procédure stockée qui ne renvoie pas d'enregistrements, ExecuteOptions détermine la façon dont elle est exécutée lorsque vous appelez *ExecSQL* ou *ExecProc*.

ExecuteOptions est un ensemble contenant aucune ou plusieurs des valeurs suivantes :

Tableau 21.4 Options d'exécution des ensembles de données ADO

Option d'exécution	Signification
eoAsyncExecute	La commande ou l'opération de lecture de données est exécutée de façon asynchrone.
eoAsyncFetch	L'ensemble de données lit d'abord le nombre d'enregistrements spécifié par la propriété *CacheSize* de façon synchrone, puis toutes les lignes restantes de façon asynchrone.
eoAsyncFetchNonBlocking	L'exécution de la commande ou les lectures de données asynchrones ne bloquent pas le thread d'exécution en cours.
eoExecuteNoRecords	Une commande ou une procédure stockée qui ne renvoie pas de données. Si des lignes sont récupérées, elles ne sont pas prises en compte et rien n'est renvoyé.

Utilisation des mises à jour groupées

Une approche pour placer les mises à jour en mémoire cache consiste à connecter l'ensemble de données ADO à un ensemble de données client à l'aide d'un fournisseur d'ensemble de données. Cette approche est présentée dans "Utilisation d'un ensemble de données client pour mettre en cache les mises à jour" à la page 23-18.

Toutefois, les composants ensemble de données ADO prennent en charge les mises à jour placées en mémoire cache, appelées alors mises à jour groupées. Le tableau suivant établit une correspondance entre le placement des mises à jour

en mémoire cache à l'aide d'un ensemble de données client et leur placement à l'aide des fonctionnalités de mises à jour groupées :

Tableau 21.5 Comparaison des mises à jour en mémoire cache d'un ensemble de données client et ADO

Ensemble de données ADO	TClientDataSet	Description
LockType	Inutilisé : les ensembles de données client placent toujours les mises à jour en mémoire cache.	Spécifie si l'ensemble de données est ouvert en mode mise à jour groupée.
CursorType	Inutilisé : les ensembles de données client manipulent toujours une photographie mémorisée des données.	Spécifie le degré d'isolement de l'ensemble de données ADO par rapport aux modifications présentes sur le serveur.
RecordStatus	UpdateStatus	Indique la mise à jour qui a éventuellement affecté la ligne en cours. *RecordStatus* fournit davantage d'informations que *UpdateStatus*.
FilterGroup	StatusFilter	Spécifie les types d'enregistrements disponibles. *FilterGroup* fournit davantage d'informations.
UpdateBatch	ApplyUpdates	Applique les mises à jour en mémoire cache au serveur de base de données. A la différence de *ApplyUpdates*, *UpdateBatch* permet de limiter les types de mises à jour à appliquer.
CancelBatch	CancelUpdates	Ne prend pas en compte les mises à jour en attente et rétablit les valeurs initiales. A la différence de *CancelUpdates*, *CancelBatch* permet de limiter les types de mises à jour à annuler.

L'utilisation des caractéristiques de mises à jour groupées des composants ensemble de données ADO fait intervenir :

- Ouverture de l'ensemble de données en mode mises à jour groupées
- Examen du statut de mise à jour ligne par ligne
- Filtrage de lignes en fonction du statut de mise à jour
- Application des mises à jour groupées dans les tables des bases
- Annulation des mises à jour groupées

Ouverture de l'ensemble de données en mode mises à jour groupées

Pour ouvrir un ensemble de données ADO en mode mises à jour groupées, il doit respecter les critères suivants :

1. La propriété *CursorType* du composant doit avoir la valeur *ctKeySet* (valeur par défaut de la propriété) ou *ctStatic*.
2. La propriété *LockType* doit avoir la valeur *ltBatchOptimistic*.
3. La commande doit être une requête SELECT.

Avant d'activer le composant ensemble de données, initialisez les propriétés *CursorType* et *LockType* comme indiqué ci-dessus. Affectez une instruction SELECT à la propriété *CommandText* du composant (pour *TADODataSet*) ou à la propriété *SQL* (pour *TADOQuery*). Pour les composants *TADOStoredProc*, affectez

à la propriété *ProcedureName* le nom d'une procédure stockée qui renvoie un ensemble de résultats. Ces propriétés peuvent être définies à la conception en utilisant l'inspecteur d'objet ou par code à l'exécution. L'exemple suivant illustre la préparation d'un composant *TADODataSet* en mode mises à jour groupées.

```
with ADODataSet1 do begin
  CursorLocation := clUseClient;
  CursorType := ctStatic;
  LockType := ltBatchOptimistic;
  CommandType := cmdText;
  CommandText := 'SELECT * FROM Employee';
  Open;
end;
```

Une fois l'ensemble de données ouvert en mode mises à jour groupées, toutes les modifications des données sont placées dans le cache au lieu d'être appliquées directement aux tables de la base.

Examen du statut de mise à jour ligne par ligne

Pour déterminer le statut de la mise à jour pour une ligne donnée, faites-en la ligne en cours puis inspectez la propriété *RecordStatus* du composant de données ADO. *RecordStatus* indique l'état de mise à jour pour la ligne en cours et uniquement pour elle.

```
case ADOQuery1.RecordStatus of
  rsUnmodified: StatusBar1.Panels[0].Text := 'Enregistrement non modifié';
  rsModified:   StatusBar1.Panels[0].Text := 'Enregistrement modifié';
  rsDeleted:    StatusBar1.Panels[0].Text := 'Enregistrement supprimé';
  rsNew:        StatusBar1.Panels[0].Text := 'Enregistrement ajouté';
end;
```

Filtrage de lignes en fonction du statut de mise à jour

Le filtrage d'un ensemble d'enregistrements ne montre que les lignes qui appartiennent au groupe des lignes ayant le statut de mise à jour avec la valeur spécifiée par la propriété *FilterGroup*. Affectez à *FilterGroup* les constantes de type *TFilterGroup* qui représentent le statut de mise à jour des lignes à afficher. La valeur *fgNone* (valeur par défaut de cette propriété) spécifie une absence de filtrage et l'affichage de toutes les lignes sans tenir compte du statut de mise à jour (sauf les lignes marquées pour effacement). L'exemple suivant n'affiche que les lignes mises à jour en attente.

```
FilterGroup := fgPendingRecords;
Filtered := True;
```

Remarque Pour que la propriété *FilterGroup* prenne effet, la propriété *Filtered* du composant ensemble de données ADO doit avoir la valeur *True*.

Application des mises à jour groupées dans les tables des bases

Pour appliquer les mises à jour en attente qui n'ont pas déjà été appliquées ou annulées, appelez la méthode *UpdateBatch*. Pour les lignes dont le contenu a changé et qui sont appliquées, les modifications sont placées dans les tables de base sur lesquelles l'ensemble d'enregistrement est basé. Une ligne du cache

marqué pour la suppression entraîne la suppression de la ligne correspondante de la table de la base. Une insertion d'enregistrement (l'enregistrement existe dans le cache mais pas dans la table de la base) est ajoutée à la table de la base. Pour les lignes modifiées, les colonnes des lignes correspondantes des tables de la base sont modifiées pour adopter les nouvelles valeurs de colonnes présentes dans le cache.

Utilisée sans paramètre, la méthode *UpdateBatch* applique toutes les mises à jour en attente. Il est possible de transmettre une valeur de type *TAffectRecords* comme paramètre de *UpdateBatch*. Si une valeur autre que *arAll* est transmise, seul un sous-ensemble des modifications en attente est appliqué. Le paramètre *arAll* a le même effet que l'absence de paramètre et provoque l'application de toutes les mises à jour en attente. L'exemple suivant n'applique que la ligne en cours :

```
ADODataSet1.UpdateBatch(arCurrent);
```

Annulation des mises à jour groupées

Pour annuler les mises à jour en attente, appelez la méthode *CancelBatch*. Lorsque vous annulez des mises à jour groupées en attente, les valeurs de champ des lignes modifiées reviennent à celles existant avant le dernier appel de *CancelBatch* ou de *UpdateBatch*, si l'une de ces méthodes a déjà été appelée, ou avant le groupe de modifications en attente.

Utilisée sans paramètre, la méthode *CancelBatch* annule toutes les mises à jour en attente. Il est possible de transmettre une valeur de type *TAffectRecords* comme paramètre de *CancelBatch*. Si une valeur autre que *arAll* est transmise, seul un sous-ensemble des modifications en attente est annulé. Le paramètre *arAll* a le même effet que l'absence de paramètre et provoque l'annulation de toutes les mises à jour en attente. L'exemple suivant annule toutes les modifications en attente :

```
ADODataSet1.CancelBatch;
```

Lecture et enregistrement des données dans des fichiers

Les données obtenues dans un composant ensemble de données ADO peuvent être enregistrées dans un fichier pour une utilisation ultérieure dans le même ordinateur ou dans un autre. Les données sont enregistrées en utilisant un des deux formats propriétaire proposés : ADTG ou XML. Ces deux formats de fichiers sont les seuls formats gérés par ADO. Toutefois, ils ne sont pas nécessairement gérés par toutes les versions de ADO. Consultez la documentation ADO pour connaître quels formats de fichiers gèrent la version en cours.

Enregistrez les données dans un fichier en utilisant la méthode *SaveToFile*. *SaveToFile* accepte deux paramètres, le nom du fichier dans lequel les données sont enregistrées, et, éventuellement, le format (ADTG ou XML) de sauvegarde des données. Indiquez le format du fichier enregistré en attribuant au paramètre *Format* la valeur *pfADTG* ou *pfXML*. Si le fichier spécifié par le paramètre *FileName* existe déjà, *SaveToFile* déclenche une exception *EOleException*.

Récupérez les données enregistrées en utilisant la méthode *LoadFromFile*. *LoadFromFile* accepte un paramètre : le nom du fichier à charger. Si le fichier spécifié n'existe pas, *LoadFromFile* déclenche une exception *EOleException*. Lors de l'appel de la méthode *LoadFromFile*, le composant ensemble de données est automatiquement activé.

Dans l'exemple suivant, la première procédure enregistre dans un fichier l'ensemble de données obtenu par le composant *TADODataSet ADODataSet1*. Le fichier destination est un fichier ADTG nommé SaveFile qui est créé sur un disque local. La seconde procédure charge ce fichier dans le composant *TADODataSet ADODataSet2*.

```
procedure TForm1.SaveBtnClick(Sender: TObject);
begin
  if (FileExists('c:\SaveFile')) then
  begin
    DeleteFile('c:\SaveFile');
    StatusBar1.Panels[0].Text := 'Fichier sauvegarde supprimé!';
  end;
  ADODataSet1.SaveToFile('c:\SaveFile', pfADTG);
end;

procedure TForm1.LoadBtnClick(Sender: TObject);
begin
  if (FileExists('c:\SaveFile')) then
    ADODataSet2.LoadFromFile('c:\SaveFile')
  else
    StatusBar1.Panels[0].Text := 'Fichier sauvegarde inexistant!';
end;
```

Il n'est pas nécessaire que les ensembles de données qui effectuent la sauvegarde et la lecture des données figurent sur la même fiche comme dans l'exemple précédent, ni dans la même application ou sur la même machine. Cela permet des transferts de données de style porte-documents d'un ordinateur à un autre.

Utilisation de TADODataSet

TADODataSet est un ensemble de données polyvalent qui permet de manipuler les données d'un stockage de données ADO. A la différence des autres composants ensemble de données ADO, *TADODataSet* n'est pas un ensemble de données de type table, de type requête ou de type procédure stockée. Par contre, il peut fonctionner comme l'un des types suivants :

- Comme un ensemble de données de type table, *TADODataSet* permet de représenter toutes les lignes et colonnes d'une seule table de base de données. Pour l'utiliser à cet effet, attribuez à la propriété *CommandType* la valeur *cmdTable* et à la propriété *CommandText* le nom de la table. *TADODataSet* gère les tâches de type table suivantes :
  - Affectation d'index pour trier des enregistrements ou constituer la base de recherches à partir d'enregistrements. Outre les propriétés et méthodes d'index standard décrites dans "Tri des enregistrements avec des index" à la page 18-30, vous pouvez utiliser la propriété *Sort* de *TADODataSet* pour

effectuer des tris à partir d'index temporaires. Les recherches basées sur les index effectuées à l'aide de la méthode *Seek* utilisent l'index en cours.

- vidage de l'ensemble de données. La méthode *DeleteRecords* offre davantage de contrôle que les méthodes connexes des autres ensembles de données de type table car elle permet de spécifier les enregistrements à supprimer.

Les tâches de type table gérées par *TADODataSet* sont disponibles même si *CommandType* n'a pas pour valeur *cmdTable*.

- Comme un ensemble de données de type requête, *TADODataSet* permet de spécifier une commande SQL unique à exécuter à l'ouverture de l'ensemble de données. Pour l'utiliser à cet effet, attribuez à la propriété *CommandType* la valeur *cmdText* et à la propriété *CommandText* la commande SQL à exécuter. A la conception, vous pouvez double-cliquer sur la propriété *CommandText* dans l'inspecteur d'objets pour utiliser l'éditeur de texte de commande afin d'élaborer la commande SQL. *TADODataSet* gère les tâches de type requête suivantes :
  - Utilisation de paramètres dans le texte des requêtes. Voir "Utilisation de paramètres dans les requêtes" à la page 18-52, pour plus d'informations sur les paramètres de requête.
  - Définition de relations maître/détail à l'aide de paramètres. Voir "Etablissement de relations maître/détail en utilisant des paramètres" à la page 18-55, pour plus d'informations sur ce sujet.
  - Préparation de la requête afin d'améliorer les performances en attribuant à la propriété *Prepared* la valeur *True*.
- Comme un ensemble de données de type procédure stockée, *TADODataSet* permet de spécifier une procédure stockée à exécuter à l'ouverture de l'ensemble de données. Pour l'utiliser à cet effet, attribuez à la propriété *CommandType* la valeur *cmdStoredProc* et à la propriété *CommandText* le nom de la procédure stockée. *TADODataSet* gère les tâches de type procédure stockée suivantes :
  - Utilisation de paramètres de procédure stockée. Voir "Utilisation de paramètres avec les procédures stockées" à la page 18-59, pour plus d'informations sur les paramètres de procédure stockée.
  - Lecture de plusieurs ensembles de résultats. Voir "Lecture de plusieurs ensembles de résultats" à la page 18-64, pour plus d'informations sur ce sujet.
  - Préparation de la procédure stockée afin d'améliorer les performances en attribuant à la propriété *Prepared* la valeur *True*.

En outre, *TADODataSet* vous permet d'utiliser les données stockées dans un fichier en attribuant à la propriété *CommandType* la valeur *cmdFile* et à la propriété *CommandText* le nom du fichier.

Avant de définir les propriétés *CommandText* et *CommandType*, vous devez lier *TADODataSet* à un stockage de données en définissant la propriété *Connection* ou *ConnectionString*. Ce processus est décrit dans "Connexion d'un ensemble de

données ADO à un stockage de données" à la page 21-11. Vous pouvez aussi utiliser un objet DataSpace RDS pour connecter *TADODataSet* à un serveur d'applications ADO. Pour ce faire, attribuez à la propriété *RDSConnection* un objet *TRDSConnection*.

Utilisation d'objets commande

Dans l'environnement ADO, les commandes sont une représentation sous forme de texte de demandes d'actions spécifiques à un fournisseur. Ce sont généralement des instructions SQL utilisant le langage de définition de données (DDL) ou le langage de manipulation de données (DML). Le langage utilisé dans ces commandes est spécifique au fournisseur, mais, généralement, il respecte le standard SQL-92 du langage SQL.

Bien que vous puissiez toujours exécuter une commande à l'aide de *TADOQuery*, il peut vous sembler opportun de ne pas utiliser un composant ensemble de données, notamment si la commande ne renvoie pas un ensemble de résultats. Vous pouvez aussi utiliser le composant *TADOCommand*, objet léger permettant d'exécuter une série de commandes, à raison d'une commande à la fois. *TADOCommand* est avant tout conçu pour l'exécution de commandes ne renvoyant pas d'ensemble de résultats, comme les instructions SQL du langage de définition de données (DDL). Cependant, il est capable via une version redéfinie de sa méthode *Execute* de renvoyer un ensemble de résultats qui peut être affecté à la propriété *RecordSet* d'un composant ensemble de données ADO.

En général, l'utilisation de *TADOCommand* est très similaire à celle de *TADODataSet*, à la différence que vous ne pouvez pas utiliser les méthodes d'ensemble de données standard pour lire les données, parcourir les enregistrements, modifier les données, etc. Les objets *TADOCommand* se connectent à un stockage de données de la même façon que les ensembles de données ADO. Voir "Connexion d'un ensemble de données ADO à un stockage de données" à la page 21-11 pour plus de détails.

Les rubriques suivantes expliquent comment spécifier et exécuter des commandes à l'aide de *TADOCommand*.

Spécification de la commande

Spécifiez les commands d'un composant *TADOCommand* à l'aide de la propriété *CommandText*. Comme *TADODataSet*, *TADOCommand* vous permet de spécifier la commande de différentes manières, en fonction de la propriété *CommandType*. *CommandType* peut prendre les valeurs suivantes : *cmdText* (si la commande est une instruction SQL), *cmdTable* (si c'est un nom de table) et *cmdStoredProc* (si la commande spécifie le nom d'une procédure stockée). A la conception, sélectionnez la valeur appropriée dans la liste proposée par l'inspecteur d'objet. A l'exécution, affectez une valeur de type *TCommandType* à la propriété *CommandType*.

```
with ADOCommand1 do begin
```

```
  CommandText := 'AddEmployee';
  CommandType := cmdStoredProc;
...
end;
```

Si le type spécifique n'est pas indiqué, c'est le serveur qui le détermine en se basant sur la commande spécifiée par *CommandText*.

CommandText peut contenir le texte d'une requête SQL qui comprend des paramètres ou le nom d'une procédure stockée qui utilise des paramètres. Vous devez alors fournir les valeurs des paramètres, qui sont liées à ceux-ci avant l'exécution de la commande. Voir "Gestion des paramètres de commande" à la page 21-21, pour plus de détails.

Utilisation de la méthode Execute

Pour que *TADOCommand* puisse exécuter sa commande, il doit être correctement connecté à un stockage de données. Cette connexion est établie de la même façon qu'avec un ensemble de données ADO. Voir "Connexion d'un ensemble de données ADO à un stockage de données" à la page 21-11, pour plus de détails.

Pour exécuter la commande, appelez la méthode *Execute*. *Execute* est une méthode redéfinie qui permet de choisir la façon la plus appropriée pour exécuter la commande.

Dans le cas des commandes qui ne requièrent aucun paramètre et dont le nombre d'enregistrements qu'elles affectent n'a aucune importance, appelez *Execute* sans définir de paramètres :

```
with ADOCommand1 do begin
  CommandText := 'UpdateInventory';
  CommandType := cmdStoredProc;
  Execute;
end;
```

Les autres versions de *Execute* vous permettent de fournir des valeurs de paramètres à l'aide d'un tableau Variant et d'obtenir le nombre d'enregistrements affectés par la commande.

Pour plus d'informations sur l'exécution de commandes renvoyant un ensemble de résultats, voir "Récupération d'ensembles de résultats à l'aide de commandes" à la page 21-21.

Annulation des commandes

Si vous exécutez la commande de façon asynchrone, après avoir appelé *Execute*, vous pouvez abandonner l'exécution en appelant la méthode *Cancel* :

```
procedure TDataForm.ExecuteButtonClick(Sender: TObject);
begin
  ADOCommand1.Execute;
end;
```

```
procedure TDataForm.CancelButtonClick(Sender: TObject);
begin
  ADOCommand1.Cancel;
end;
```

La méthode *Cancel* n'a d'effet que si une commande en attente a été exécutée de manière asynchrone (*eoAsynchExecute* dans le paramètre *ExecuteOptions* de la méthode *Execute*). Une commande est dite en attente si la méthode *Execute* a été appelée mais n'est pas encore achevée ou n'a pas dépassé le délai limite.

Une commande dépasse le délai limite si elle n'est pas terminée ou annulée dans l'intervalle, en secondes, imparti par la propriété *CommandTimeout*. Par défaut, les commandes disposent d'un délai de 30 secondes.

Récupération d'ensembles de résultats à l'aide de commandes

A la différence des composants *TADOQuery*, qui exécutent différentes méthodes selon qu'elles renvoient ou non un ensemble de résultats, *TADOCommand* utilise toujours *Execute* pour exécuter la commande, qu'elle renvoie ou non un ensemble de résultats. Lorsque la commande renvoie un ensemble de résultats, *Execute* renvoie une interface de l'interface _RecordSet ADO.

La façon la plus pratique d'utiliser cette interface consiste à l'affecter à la propriété *RecordSet* d'un ensemble de données ADO.

Par exemple, le code suivant utilise *TADOCommand* (*ADOCommand1*) pour exécuter une requête SELECT, qui renvoie un ensemble de résultats. Cet ensemble de résultats est ensuite affecté à la propriété *RecordSet* d'un composant *TADODataSet* (*ADODataSet1*).

```
with ADOCommand1 do begin
  CommandText := 'SELECT Company, State ' +
    'FROM customer ' +
    'WHERE State = :StateParam';
  CommandType := cmdText;
  Parameters.ParamByName('StateParam').Value := 'HI';
  ADODataSet1.Recordset := Execute;
end;
```

Dès que l'ensemble de résultats est affecté à la propriété *Recordset* d'un ensemble de données ADO, l'ensemble de données est automatiquement activé et les données sont accessibles.

Gestion des paramètres de commande

Un objet *TADOCommand* peut utiliser des paramètres de deux façons :

- La propriété *CommandText* peut spécifier une requête qui comprend des paramètres. L'utilisation de requêtes paramétrées dans *TADOCommand* s'apparente à celle d'une requête paramétrée dans un objet ADO. Voir "Utilisation de paramètres dans les requêtes" à la page 18-52, pour plus d'informations sur les requêtes paramétrées.

- La propriété *CommandText* peut spécifier une procédure stockée qui utilise des paramètres. Les paramètres de procédure stockée s'utilisent avec *TADOCommand* de la même façon qu'avec un ensemble de données ADO. Voir "Utilisation de paramètres avec les procédures stockées" à la page 18-59, pour plus d'informations sur les paramètres de procédures stockées.

Lorsque vous utilisez *TADOCommand*, vous pouvez indiquer les valeurs de paramètres de deux façons : en les fournissant au moment d'appeler la méthode *Execute* ou, à l'avance, en les définissant par le biais de la propriété *Parameters*.

La méthode *Execute* est redéfinie dans des versions qui acceptent un ensemble de valeurs de paramètres sous la forme d'un tableau Variant. Cela permet de fournir des valeurs de paramètres rapidement sans définir la propriété *Parameters* :

```
ADOCommand1.Execute(VarArrayOf([Edit1.Text, Date]));
```

Lorsque vous utilisez des procédures stockées qui renvoient des paramètres de sortie, vous devez recourir à la propriété *Parameters*. Même si vous n'avez pas besoin de lire les paramètres de sortie, la propriété *Parameters* peut s'avérer utile car elle vous permet de fournir les paramètres à la conception et d'utiliser les propriétés de *TADOCommand* de la même façon que les paramètres des ensembles de données.

Lorsque vous définissez la propriété *CommandText*, la propriété *Parameters* est automatiquement mise à jour pour refléter les paramètres de la requête ou ceux utilisés par la procédure stockée. A la conception, vous pouvez utilise l'éditeur de paramètres pour accéder aux paramètres en cliquant sur le bouton points de suspension de la propriété *Parameters* dans l'inspecteur d'objets. A l'exécution, utilisez les propriétés et méthodes de *TParameter* pour spécifier ou connaître la valeur de chaque paramètre.

```
with ADOCommand1 do begin
  CommandText := 'INSERT INTO Talley ' +
    '(Counter) ' +
    'VALUES (:NewValueParam)';
  CommandType := cmdText;
  Parameters.ParamByName('NewValueParam').Value := 57;
  Execute
end;
```

Chapitre

22

Utilisation d'ensembles de données unidirectionnels

dbExpress est un ensemble de pilotes de bases de données légers qui permettent d'accéder rapidement aux serveurs de bases de données SQL. Pour chaque base de données prise en charge, *dbExpress* fournit un pilote qui adapte les logiciels serveur à un ensemble uniforme d'interfaces *dbExpress*. Lorsque vous déployez une application de base de données qui utilise *dbExpress*, il suffit d'inclure une bibliothèque de liaison dynamique (le pilote propre au serveur) avec les fichiers d'application que vous créez.

dbExpress vous permet d'accéder aux bases de données à l'aide d'ensembles de données unidirectionnels. Ceux-ci ont été conçus pour un accès léger et rapide aux informations de la base, avec des temps système réduits. Comme les autres ensembles de données, ils peuvent envoyer une commande SQL au serveur de la base de données et, si la commande retourne un ensemble d'enregistrements, obtenir un curseur pour accéder à ces enregistrements. Toutefois, les ensembles de données unidirectionnels peuvent uniquement récupérer un curseur unidirectionnel. Ils ne placent pas les données dans un tampon de mémoire, ce qui les rend plus rapides et moins consommateurs de mémoire que les autres types d'ensembles de données. Mais, parce qu'ils ne mettent pas les enregistrements dans un tampon, les ensembles de données unidirectionnels sont moins flexibles que les autres ensembles de données. De nombreuses capacités de *TDataSet* ne sont pas implémentées ou provoquent des exceptions dans les ensembles de données unidirectionnels. Par exemple :

- Les seules méthodes de navigation supportées sont *First* and *Next.* La plupart des autres déclenchent des exceptions. Certaines, comme celles qui participent au support des signets, ne font rien du tout.
- Il n'y a pas de support d'édition intégré, car cela nécessiterait un tampon pour les modifications. La propriété *CanModify* est toujours à *False*, et donc les tentatives d'édition de l'ensemble de données échouent toujours. Vous pouvez cependant utiliser les ensembles de données unidirectionnels pour mettre à

jour les données en faisant appel à une commande SQL UPDATE, ou leur conférer la prise en charge conventionnelle de l'édition en recourant à un ensemble de données client dbExpress ou en connectant l'ensemble de données à un ensemble de données client (voir "Connexion à un autre ensemble de données" à la page 14-12).

- Il n'y a pas de support des filtres, car ceux-ci doivent fonctionner avec plusieurs enregistrements, ce qui nécessite la mise en tampon. Si vous essayez de filtrer un ensemble de données unidirectionnel, une exception est déclenchée. Pour imposer des limites aux données à afficher, il vous faut utiliser la commande SQL qui définit les données pour l'ensemble de données.
- Il n'y a pas de support des champs de référence, qui nécessitent un tampon pour les enregistrements multiples contenant des valeurs de référence. Si vous définissez un champ de référence sur un ensemble de données unidirectionnel, il ne fonctionnera pas correctement.

Malgré ces limites, les ensembles de données unidirectionnels offrent un moyen puissant d'accéder aux données. Ils représentent le mécanisme d'accès aux données le plus rapide et sont très simples à utiliser et à déployer.

Types d'ensembles de données unidirectionnels

La page *dbExpress* de la palette des composants contient quatre types d'ensembles de données unidirectionnels : *TSQLDataSet, TSQLQuery, TSQLTable* et *TSQLStoredProc.*

TSQLDataSet est le plus général des quatre. Vous pouvez utiliser un ensemble de données SQL pour représenter n'importe quelles données disponibles via *dbExpress*, ou pour envoyer des commandes à une base de données accédée via *dbExpress*. C'est le composant recommandé pour travailler avec des tables dans de nouvelles applications de bases de données.

TSQLQuery est un ensemble de données de type requête qui encapsule une instruction SQL et permet aux applications d'accéder aux enregistrements éventuellement obtenus. Voir "Utilisation d'ensembles de données de type requête" à la page 18-49, pour plus d'informations sur l'utilisation des ensembles de données de type requête.

TSQLTable est un ensemble de données de type table qui représente toutes les lignes et colonnes d'une seule table de base de données. Voir"Utilisation d'ensembles de données de type table" à la page 18-29, pour plus d'informations sur l'utilisation des ensembles de données de type table.

TSQLStoredProc est un ensemble de données de type procédure stockée qui exécute une procédure stockée définie sur un serveur de bases de données. Voir"Utilisation d'ensembles de données de type procédure stockée" à la page 18-58, pour plus d'informations sur l'utilisation des ensembles de données de type procédure stockée.

Remarque La page *dbExpress* comprend également *TSQLClientDataSet*, qui n'est pas un ensemble de données unidirectionnel. Il s'agit d'un ensemble de données client

qui utilise un ensemble de données unidirectionnel de manière interne pour accéder à ses données.

Connexion au serveur de bases de données

La première étape du travail avec un ensemble de données unidirectionnel est de se connecter à un serveur de base de données. A la conception, une fois qu'un ensemble de données dispose d'une connexion active à un serveur de base de données, l'inspecteur d'objets peut fournir des listes déroulantes de valeurs pour les autres propriétés. Par exemple, s'il s'agit d'une procédure stockée, il faut qu'une connexion soit active pour que l'inspecteur d'objets puisse indiquer les procédures stockées disponibles sur le serveur.

La connexion à un serveur de base de données est représentée par un composant *TSQLConnection* distinct. Vous utilisez *TSQLConnection* comme tout autre composant connexion de base de données. Pour plus d'informations sur les composants connexion de base de données, voir chapitre 17, "Connexion aux bases de données".

Pour utiliser *TSQLConnection* afin de connecter un ensemble de données unidirectionnel à un serveur de bases de données, définissez la propriété *SQLConnection.* Pendant la conception, vous pouvez choisir le composant connexion SQL dans une liste déroulante de l'inspecteur d'objets. Si vous faites cette affectation pendant l'exécution, soyez sûr que la connexion est active :

```
SQLDataSet1.SQLConnection := SQLConnection1;
SQLConnection1.Connected := True;
```

Généralement, tous les ensembles de données unidirectionnels d'une application partagent le même composant connexion, sauf si vous travaillez avec des données issues de plusieurs serveurs de bases de données. Toutefois, vous pouvez utiliser une connexion différente pour chaque ensemble de données si le serveur ne prend pas en charge plusieurs instructions par connexion. Vérifiez si le serveur de base de données nécessite une connexion différente pour chaque ensemble de données en lisant la propriété *MaxStmtsPerConn*. Par défaut, *TSQLConnection* génère les connexions selon les besoins lorsque le serveur limite le nombre d'instructions pouvant être exécutées via une connexion. Si vous souhaitez assurer un suivi plus approfondi des connexions utilisées, attribuez à la propriété *AutoClone* la valeur *False*.

Avant de définir la propriété *SQLConnection*, vous devez configurer le composant *TSQLConnection* afin qu'il identifie le serveur de bases de données et tous les paramètres de connexion requis (notamment la base de données à utiliser sur le serveur, le nom de la machine hôte exécutant le serveur, le nom d'utilisateur, le mot de passe, etc.).

Configuration de TSQLConnection

Afin de décrire une connexion de base de données de façon suffisamment détaillée pour que *TSQLConnection* puisse l'ouvrir, vous devez identifier le pilote à utiliser et un ensemble de paramètres de connexion transmis au pilote.

Identification du pilote

Le pilote est identifié par la propriété *DriverName*, qui correspond au nom d'un pilote *dbExpress* installé, tel que INTERBASE, ORACLE, MYSQL ou DB2. Le nom de pilote est associé à deux fichiers :

- Le pilote *dbExpress*. Celui-ci peut être une bibliothèque de liaison dynamique portant un nom tel que dbexpint.dll, dbexpora.dll, dbexpmys.dll ou dbexpdb2.dll, ou une unité compilée que vous pouvez lier de façon statique à votre application (dbexptint.dcu, dbexpora.dcu, dbexpmys.dcu ou dbexpdb2.dcu).
- La bibliothèque de liaison dynamique fournie par l'éditeur de bases de données pour la gestion de la partie client.

La relation entre ces deux fichiers et le nom de la base de données est stockée dans un fichier appelé dbxdrivers.ini, qui est mis à jour lorsque vous installez un pilote *dbExpress*. Généralement, vous n'avez pas à vous soucier de ces fichiers car le composant connexion SQL les recherche dans dbxdrivers.ini lorsqu'il reçoit la valeur de *DriverName*. Lorsque vous définissez la propriété *DriverName*, *TSQLConnection* attribue automatiquement aux propriétés *LibraryName* et *VendorLib* les noms des bibliothèques de liaison dynamique associées. Une fois *LibraryName* et *VendorLib* définies, votre application n'est plus dépendante de dbxdrivers.ini (en d'autres termes, vous n'avez pas besoin de déployer le fichier dbxdrivers.ini avec votre application sauf si vous définissez la propriété *DriverName* à l'exécution.)

Spécification des paramètres de connexion

La propriété *Params* est une liste de chaînes associant des noms à des valeurs. Chaque association présente la forme *Nom=Valeur*, où *Nom* représente le nom du paramètre, et *Valeur* la valeur à attribuer.

Les paramètres particuliers requis dépendent du serveur de base de données utilisé. Toutefois, un paramètre particulier, *Database*, est requis pour tous les serveurs. Sa valeur dépend du serveur que vous utilisez. Par exemple, *Database* représente le nom du fichier .gdb (InterBase), l'entrée dans TNSNames.ora (ORACLE) ou le nœud côté client (DB2).

Parmi les paramètres classiques figurent *User_Name* (nom à utiliser lors de l'ouverture de session), *Password* (mot de passe associé à *User_Name*), *HostName* (nom de la machine ou adresse IP du serveur) et *TransIsolation* (degré de reconnaissance par vos transactions des modifications apportées par les autres transactions). Lorsque vous spécifiez un nom de pilote, la propriété *Params* est préchargée avec tous les paramètres requis pour ce type de pilote, initialisés à leurs valeurs par défaut.

Params étant une liste de chaînes, lors de la conception, vous pouvez double-cliquer sur la propriété *Params* dans l'inspecteur d'objets pour modifier les paramètres à l'aide de l'éditeur de liste de chaînes. Lors de l'exécution, utilisez la propriété *Params.Values* pour attribuer des valeurs aux différents paramètres.

Dénomination d'une description de connexion

Bien que vous puissiez toujours spécifier une connexion à l'aide uniquement des propriétés *DatabaseName* et *Params*, il peut s'avérer plus pratique de nommer une combinaison spécifique puis simplement d'identifier la connexion par son nom. Vous pouvez nommer les combinaisons de paramètres et de bases de données *dbExpress*, qui sont ensuite enregistrées dans un fichier appelé dbxconnections.ini. Le nom de chaque combinaison est un nom de connexion.

Une fois que vous avez défini le nom de connexion, vous pouvez identifier une connexion de base de données en attribuant simplement à la propriété *ConnectionName* un nom de connexion valide. Le fait de définir *ConnectionName* définit automatiquement les propriétés *DriverName* et *Params*. Une fois *ConnectionName* définie, vous pouvez modifier la propriété *Params* afin de créer des variantes temporaires à partir de l'ensemble de valeurs de paramètres enregistré, mais la modification de la propriété *DriverName* efface à la fois *Params* et *ConnectionName*.

L'un des avantages de l'utilisation de noms de connexion apparaît lorsque vous développez votre application en utilisant une base de données (par exemple Local InterBase), mais que vous la déployez en vue d'une utilisation par une autre base de données (telle qu'ORACLE). Dans ce cas, sur le système où le déploiement est effectué, le profil de *DriverName* et *Params* est susceptible de différer par rapport aux valeurs utilisées pendant le développement. Vous pouvez facilement passer d'une description de connexion à l'autre en utilisant deux versions du fichier dbxconnections.ini. A la conception, votre application charge *DriverName* et *Params* à partir de la version de conception de dbxconnections.ini. Puis, lorsque vous déployez votre application, elle charge ces valeurs à partir d'une version distincte de dbxconnections.ini qui utilise la base de données "réelle". Toutefois, pour que cela fonctionne, vous devez indiquer à votre composant connexion de recharger les propriétés *DriverName* et *Params* lors de l'exécution. Il existe deux moyens de le faire :

- Attribuer à la propriété *LoadParamsOnConnect* la valeur *True*. Ainsi, à l'ouverture de la connexion, *TSQLConnection* attribue automatiquement à *DriverName* et *Params* les valeurs associées à *ConnectionName* dans dbxconnections.ini.
- Appeler la méthode *LoadParamsFromIniFile*. Elle attribue à *DriverName* et *Params* les valeurs associées à *ConnectionName* dans dbxconnections.ini (ou dans un autre fichier que vous spécifiez). Vous pouvez opter pour cette méthode si vous souhaitez remplacer certaines valeurs de paramètres avant d'ouvrir la connexion.

Utilisation de l'éditeur de connexion

La relation entre les noms de connexion et leurs paramètres de pilote et de connexion associés est stockée dans le fichier dbxconnections.ini. Vous pouvez créer ou modifier ces associations à l'aide de l'éditeur de connexion.

Pour afficher l'éditeur de connexion, double-cliquez sur le composant *TSQLConnection*. L'éditeur de connexion apparaît, avec une liste déroulante contenant tous les pilotes disponibles, une liste de noms de connexion pour le

pilote sélectionné et un tableau présentant les paramètres de connexion associés au nom de connexion sélectionné.

Cette boîte de dialogue vous permet d'indiquer la connexion à utiliser en sélectionnant un pilote et un nom de connexion. Après avoir déterminé la configuration souhaitée, cliquez sur le bouton Tester la connexion afin de vérifier que vous avez choisi une configuration valide.

En outre, cette boîte de dialogue vous permet de modifier les connexions nommées dans dbxconnections.ini :

- Modifiez les valeurs de paramètres dans le tableau des paramètres afin de modifier la connexion nommée sélectionnée. Lorsque vous quittez la boîte de dialogue en cliquant sur OK, les nouvelles valeurs de paramètres sont enregistrées dans le fichier dbxconnections.ini.
- Cliquez sur le bouton Ajouter une connexion afin de définir une nouvelle connexion nommée. Une boîte de dialogue apparaît dans laquelle vous pouvez spécifier le pilote à utiliser et le nom de la nouvelle connexion. Une fois la connexion nommée, modifiez les paramètres afin de spécifier la connexion souhaitée puis cliquez sur OK pour enregistrer la nouvelle connexion dans dbxconnections.ini.
- Cliquez sur le bouton Supprimer la connexion pour supprimer la connexion nommée sélectionnée de dbxconnections.ini.
- Cliquez sur le bouton Renommer la connexion pour modifier le nom de la connexion nommée sélectionnée. Toutes les modifications que vous avez apportées aux paramètres sont enregistrées sous le nouveau nom lorsque vous cliquez sur OK.

Spécification des données à afficher

Il existe de nombreux moyens de spécifier les données que représente un ensemble de données unidirectionnel. Votre choix dépendra du type d'ensemble de données unidirectionnel que vous utilisez et de l'origine des informations : une simple table, les résultats d'une requête ou les résultats d'une procédure stockée.

Quand vous travaillez avec un composant *TSQLDataSet*, utilisez la propriété *CommandType* pour indiquer d'où l'ensemble de données obtient ses données. *CommandType* peut prendre l'une quelconque des valeurs suivantes :

- *ctQuery* : Si *CommandType* est *ctQuery*, *TSQLDataSet* exécute la requête que vous spécifiez. Si la requête est une commande SELECT, l'ensemble de données contient l'ensemble d'enregistrements résultant.
- *ctTable* : Si *CommandType* est *ctTable*, *TSQLDataSet* extrait tous les enregistrements de la table spécifiée.
- *ctStoredProc* : Si *CommandType* est *ctStoredProc*, *TSQLDataSet* exécute une procédure stockée. Si cette dernière renvoie un curseur, l'ensemble de données contient les enregistrements renvoyés.

Remarque Vous pouvez aussi remplir l'ensemble de données unidirectionnel par des métadonnées qui vous renseignent sur ce qui est disponible sur le serveur. Pour connaître la façon de le faire, voir "Récupération de métadonnées dans un ensemble de données unidirectionnel" à la page 22-14.

Représentation des résultats d'une requête

L'utilisation d'une requête est le moyen le plus général de spécifier un ensemble d'enregistrements. Les requêtes sont de simples commandes écrites en SQL. Vous pouvez utiliser soit *TSQLDataSet* soit *TSQLQuery* pour représenter le résultat d'une requête.

Lorsque vous utilisez *TSQLDataSet*, attribuez à la propriété *CommandType* la valeur *ctQuery* et le texte de l'instruction de requête à la propriété *CommandText*. Si vous utilisez *TSQLQuery*, affectez alors la requête à la propriété *SQL*. Ces propriétés fonctionnent de la même manière pour tous les ensembles de données polyvalents ou de type requête. Voir "Spécification de la requête" à la page 18-50, pour plus de détails à leur sujet.

Lorsque vous spécifiez la requête, elle peut inclure des paramètres, ou variables, dont les valeurs peuvent être modifiées pendant la conception ou pendant l'exécution. Les paramètres peuvent remplacer les valeurs des données qui apparaissent dans l'instruction SQL. L'utilisation de paramètres dans les requêtes et la définition de valeurs pour ces paramètres sont décrites dans "Utilisation de paramètres dans les requêtes" à la page 18-52.

SQL définit des requêtes comme UPDATE qui exécutent des actions sur le serveur mais ne renvoient pas un ensemble d'enregistrements. Ces requêtes sont décrites dans "Exécution des commandes ne renvoyant pas d'enregistrement" à la page 22-10.

Représentation des enregistrements d'une table

Quand vous voulez représenter tous les champs et tous les enregistrements d'une seule table de la base de données sous-jacente, vous pouvez utiliser soit *TSQLDataSet*, soit *TSQLTable* pour générer la requête SQL à votre place.

Remarque Si les performances du serveur entrent en compte, vous préférerez composer explicitement la requête et non vous reposer sur la génération automatique. Les requêtes générées automatiquement utilisent des caractères génériques au lieu d'énumérer explicitement tous les champs de la table. Cela peut entraîner une légère baisse de performance sur le serveur. Le caractère générique (*) des requêtes générées automatiquement est plus robuste lors des modifications de champs sur le serveur.

Représentation d'une table en utilisant TSQLDataSet

Pour que *TSQLDataSet* génère une requête qui lise tous les champs et tous les enregistrements d'une seule table de base de données, définissez la propriété *CommandType* par *ctTable*.

Quand *CommandType* vaut *ctTable*, *TSQLDataSet* génère une requête en se basant sur les valeurs de deux propriétés :

- *CommandText* spécifie le nom de la table de base de données que doit représenter l'objet *TSQLDataSet*.
- *SortFieldNames* énumère les noms des champs à utiliser pour trier les données, dans l'ordre de leur prépondérance.

Par exemple, si vous spécifiez ceci :

```
SQLDataSet1.CommandType := ctTable;
SQLDataSet1.CommandText := 'Employee';
SQLDataSet1.SortFieldNames := 'HireDate,Salary'
```

TSQLDataSet génère la requête suivante, qui présente tous les enregistrements de la table Employee, triés par date d'embauche (HireDate) et, pour une même date d'embauche, par salaire (Salary) :

```
select * from Employee order by HireDate, Salary
```

Représentation d'une table en utilisant TSQLTable

Si vous utilisez *TSQLTable*, spécifiez la table que vous voulez par le biais de la propriété *TableName*.

Pour spécifier l'ordre des champs dans l'ensemble de données, vous devez spécifier un index. Il existe deux moyens de le faire :

- Définissez la propriété *IndexName* par le nom d'un index défini sur le serveur qui impose l'ordre que vous souhaitez.
- Définissez la propriété *IndexFieldNames* par la liste, délimitée par des points virgules, des noms des champs sur lesquels trier. *IndexFieldNames* fonctionne comme la propriété *SortFieldNames* de *TSQLDataSet*, sauf qu'elle utilise comme délimiteur le point virgule au lieu de la virgule.

Représentation des résultats d'une procédure stockée

Les procédures stockées sont des jeux d'instructions SQL nommés et enregistrés sur un serveur SQL. La façon d'indiquer la procédure stockée à exécuter dépend du type d'ensemble de données unidirectionnel que vous utilisez.

Lorsque vous utilisez *TSQLDataSet*, pour spécifier une procédure stockée :

- Définissez la propriété *CommandType* par *ctStoredProc*.
- Spécifiez le nom de la procédure stockée comme valeur de la propriété *CommandText* :

```
SQLDataSet1.CommandType := ctStoredProc;
SQLDataSet1.CommandText := 'MyStoredProcName';
```

Lorsque vous utilisez *TSQLStoredProc*, il vous suffit de spécifier le nom de la procédure stockée comme valeur de la propriété *StoredProcName*.

```
SQLStoredProc1.StoredProcName := 'MyStoredProcName';
```

Une fois que vous avez identifié une procédure stockée, il se peut que votre application ait besoin de fournir les valeurs des paramètres d'entrée de la procédure stockée ou d'extraire les valeurs des paramètres de sortie à la fin de l'exécution de la procédure stockée. Voir "Utilisation de paramètres avec les procédures stockées" à la page 18-59, pour plus d'informations sur l'utilisation des paramètres de procédure stockée.

Récupération des données

Lorsque vous avez spécifié la source des données, vous devez récupérer les données avant que votre application ne puisse y accéder. Lorsque l'ensemble de données a récupéré les données, les contrôles orientés données, liés à l'ensemble de données au moyen d'une source de données, affichent automatiquement les valeurs des données, tandis que les ensembles de données client liés à l'ensemble de données au moyen d'un fournisseur peuvent être remplis par les enregistrements.

Comme pour n'importe quel ensemble de données, il y a deux façons de récupérer les données pour un ensemble de données unidirectionnel :

- Définir la propriété *Active* par *True*, soit dans l'inspecteur d'objets lors de la conception, soit dans le code lors de l'exécution :

```
CustQuery.Active := True;
```

- Appeler la méthode *Open* lors de l'exécution :

```
CustQuery.Open;
```

Vous pouvez utiliser la propriété *Active* ou la méthode *Open* avec n'importe quel ensemble de données unidirectionnel obtenant ses enregistrements du serveur. Que ces enregistrements proviennent d'une requête SELECT (y compris d'une requête générée automatiquement lorsque *CommandType* vaut *ctTable*) ou d'une procédure stockée n'a pas d'importance.

Préparation de l'ensemble de données

Avant qu'une requête ou une procédure stockée ne s'exécute sur le serveur, elle doit d'abord être "préparée". Préparer l'ensemble de données signifie que *dbExpress* et le serveur allouent des ressources à l'instruction et à ses paramètres. Si *CommandType* vaut *ctTable*, cela se passe au moment où l'ensemble de données génère sa requête SELECT. Tous les paramètres qui ne sont pas liés par le serveur sont incorporés dans une requête à ce moment-là.

Les ensembles de données unidirectionnels sont automatiquement préparés lorsque vous définissez *Active* par *True* ou appelez la méthode *Open*. Lorsque vous fermez l'ensemble de données, les ressources allouées à l'exécution de l'instruction sont libérées. Si vous prévoyez d'exécuter plusieurs fois la requête ou la procédure stockée, vous pouvez améliorer les performances en préparant l'ensemble de données de manière explicite avant de l'ouvrir pour la première

fois. Pour préparer un ensemble de données de manière explicite, définissez sa propriété *Prepared* par *True*.

```
CustQuery.Prepared := True;
```

Lorsque vous préparez l'ensemble de données de manière explicite, les ressources allouées à l'exécution de l'instruction ne sont pas libérées tant que vous ne définissiez pas *Prepared* par *False*.

Définissez la propriété *Prepared* par *False* si vous voulez que l'ensemble de données soit re-préparé avant son exécution (par exemple, si vous modifiez la valeur d'un paramètre ou la propriété *SortFieldNames*).

Récupération de plusieurs ensembles de données

Certaines procédures stockées renvoient plusieurs ensembles d'enregistrements. Lorsque vous l'ouvrez, l'ensemble de données ne récupère que le premier ensemble d'enregistrements. Pour accéder aux autres ensembles d'enregistrements, appelez la méthode *NextRecordSet* :

```
var
  DataSet2: TSQLDataSet;
  nRows: Integer;
begin
  DataSet2 := SQLDataSet1.NextRecordSet(nRows);
  ...
```

NextRecordSet renvoie un composant *TSQLDataSet* nouvellement créé qui donne accès au prochain ensemble d'enregistrements. La première fois que vous appelez *NextRecordSet*, elle renvoie un ensemble de données pour le second ensemble d'enregistrements. Appeler *NextRecordSet* renvoie un troisième ensemble de données, et ainsi de suite jusqu'à ce qu'il n'y ait plus d'ensemble d'enregistrements. Lorsque qu'il n'y a pas d'ensemble de données supplémentaire, *NextRecordSet* renvoie **nil**.

Exécution des commandes ne renvoyant pas d'enregistrement

Vous pouvez utiliser un ensemble de données unidirectionnel même si la requête ou la procédure stockée qu'il représente ne renvoie pas d'enregistrement. De telles commandes comprennent les instructions DDL (Data Definition Language) ou DML (Data Manipulation Language) autres que les instructions SELECT (par exemple, les commandes INSERT, DELETE, UPDATE, CREATE INDEX et ALTER TABLE ne renvoie pas d'enregistrement). Le langage utilisé dans les commandes est spécifique au serveur mais généralement conforme au standard SQL-92 du langage SQL.

La commande SQL que vous exécutez doit être acceptable pour le serveur que vous utilisez. Les ensembles de données unidirectionnels n'évaluent pas la commande SQL et ne l'exécutent pas. Ils transmettent simplement la commande au serveur pour son exécution.

Remarque Si la commande ne renvoie pas d'enregistrement, vous n'avez pas besoin d'utiliser un ensemble de données unidirectionnel, car vous n'avez pas besoin des méthodes de l'ensemble de données donnant accès à l'ensemble d'enregistrements. Le composant de connexion SQL, qui se connecte au serveur de base de données, peut être utilisé directement pour exécuter les commandes sur le serveur. Voir "Envoi de commandes au serveur" à la page 17-11, pour plus de détails.

Spécification de la commande à exécuter

Avec les ensembles de données unidirectionnels, la façon dont vous spécifiez la commande à exécuter est la même, que la commande produise un ensemble de données ou non. Ainsi :

Lorsque vous employez *TSQLDataSet*, utilisez les propriétés *CommandType* et *CommandText* pour spécifier la commande :

- Si *CommandType* vaut *ctQuery*, *CommandText* est l'instruction SQL à transmettre au serveur.
- Si *CommandType* vaut *ctStoredProc*, *CommandText* est le nom de la procédure stockée à exécuter.

Lorsque vous employez *TSQLQuery*, utilisez la propriété *SQL* pour spécifier l'instruction SQL à transmettre au serveur.

Lorsque vous employez *TSQLStoredProc*, utilisez la propriété *StoredProcName* pour spécifier le nom de la procédure stockée à exécuter.

Tout comme vous avez spécifié la commande de la même façon que pour récupérer des enregistrements, vous travaillerez avec les paramètres de requête ou les paramètres de procédure stockée comme avec les requêtes et les procédures stockées renvoyant des enregistrements. Voir "Utilisation de paramètres dans les requêtes" à la page 18-52 et "Utilisation de paramètres avec les procédures stockées" à la page 18-59, pour plus de détails.

Exécution de la commande

Pour exécuter une requête ou une procédure stockée ne renvoyant pas d'enregistrement, n'utilisez pas la propriété *Active* ni la méthode *Open*. En revanche, utilisez :

- La méthode *ExecSQL* si l'ensemble de données est une instance de *TSQLDataSet* or *TSQLQuery*.

```
FixTicket.CommandText := 'DELETE FROM TrafficViolations WHERE (TicketID = 1099)';
FixTicket.ExecSQL;
```

- La méthode *ExecProc* si l'ensemble de données est une instance de*TSQLStoredProc*.

```
SQLStoredProc1.StoredProcName := 'MyCommandWithNoResults';
SQLStoredProc1.ExecProc;
```

Conseil Si vous exécutez la requête ou la procédure stockée plusieurs fois, il convient de définir la propriété *Prepared* par *True*.

Création et modification des métadonnées du serveur

La majorité des commandes ne renvoyant pas de données se répartissent en deux catégories : celles servant à modifier les données (comme les commandes INSERT, DELETE et UPDATE) et celles servant à créer ou modifier sur le serveur des entités telles que les tables, les index et les procédures stockées.

Si vous ne voulez pas utiliser de commandes SQL explicites pour l'édition, vous pouvez lier votre ensemble de données unidirectionnel à un ensemble de données client et laisser celui-ci gérer toute la génération des commandes SQL s'appliquant à l'édition (voir "Connexion d'un ensemble de données client à un autre ensemble de données dans la même application" à la page 14-14). En fait, c'est l'approche recommandée car les contrôles orientés données sont conçus pour opérer les modifications via un ensemble de données client tel que *TClientDataSet*.

Cependant, le seul moyen dont dispose votre application pour créer ou modifier les métadonnées sur le serveur, est d'envoyer une commande. Tous les pilotes de bases de données ne supportent pas la même syntaxe SQL. Il n'est pas du ressort de ce document de décrire la syntaxe SQL supportée par chaque type de base de données ni les différences existant entre ces types. Pour avoir des informations complètes et récentes sur l'implémentation SQL d'un système de base de données particulier, reportez-vous à la documentation livrée avec ce système.

En général, utilisez l'instruction CREATE TABLE pour créer des tables dans une base de données et CREATE INDEX pour créer de nouveaux index pour ces tables. Lorsqu'elles sont supportées, utilisez les instructions CREATE qui ajoutent les divers objets de métadonnées, comme CREATE DOMAIN, CREATE VIEW, CREATE SCHEMA et CREATE PROCEDURE.

Pour chaque instruction CREATE, il existe une instruction DROP correspondante qui efface l'objet de métadonnées. Ces instructions comprennent DROP TABLE, DROP VIEW, DROP DOMAIN, DROP SCHEMA et DROP PROCEDURE.

Pour modifier la structure d'une table, utilisez une instruction ALTER TABLE. ALTER TABLE a des clauses ADD et DROP permettant de créer de nouveaux éléments dans la table et de les supprimer. Par exemple, utilisez la clause ADD COLUMN pour ajouter à la table une nouvelle colonne, et DROP CONSTRAINT pour supprimer une contrainte préalablement établie pour la table.

Par exemple, l'instruction suivante crée une procédure stockée appelée GET_EMP_PROJ sur une base de données InterBase :

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT PROJ_ID
```

```
  FROM EMPLOYEE_PROJECT
  WHERE EMP_NO = :EMP_NO
  INTO :PROJ_ID
  DO
    SUSPEND;
END
```

Le code suivant utilise un *TSQLDataSet* pour créer cette procédure stockée. Remarquez l'utilisation de la propriété *ParamCheck* pour empêcher l'ensemble de données de confondre les paramètres de la définition de la procédure stockée (:EMP_NO et :PROJ_ID) avec un paramètre de la requête créant la procédure stockée.

```
with SQLDataSet1 do
begin
  ParamCheck := False;
  CommandType := ctQuery;
  CommandText := 'CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT) ' +
      'RETURNS (PROJ_ID CHAR(5)) AS ' +
      'BEGIN ' +
        'FOR SELECT PROJ_ID FROM EMPLOYEE_PROJECT ' +
        'WHERE EMP_NO = :EMP_NO ' +
        'INTO :PROJ_ID ' +
          'DO SUSPEND; ' +
       'END';
  ExecSQL;
end;
```

Définition de curseurs liés maître/détail

Il existe deux façons d'utiliser des curseurs liés pour définir une relation maître/détail dans laquelle un ensemble de données unidirectionnel fait office d'ensemble détail. Le choix dépend du type d'ensemble de données unidirectionnel utilisé. Lorsque vous avez défini cette relation, l'ensemble de données unidirectionnel (le "plusieurs" de la relation un-à-plusieurs) donne accès aux seuls enregistrements qui correspondent à l'enregistrement en cours dans l'ensemble maître (le "un" de la relation un-à-plusieurs).

TSQLDataSet et *TSQLQuery* nécessitent l'utilisation d'une requête paramétrée pour établir une relation maître/détail. Cette technique permet de créer ces relations sur tous les ensembles de données de type requête. Pour plus d'informations sur la création de relations maître/détail impliquant des ensembles de données de type requête, voir "Etablissement de relations maître/détail en utilisant des paramètres" à la page 18-55.

Pour définir une relation maître/détail où l'ensemble détail est une instance de *TSQLTable*, utilisez les propriétés *MasterSource* et *MasterFields*, comme vous le feriez avec tout autre ensemble de données de type table. Pour plus d'informations sur la création de relations maître/détail impliquant des ensembles de données de type table, voir "Etablissement de relations maître/détail en utilisant des paramètres" à la page 18-55.

Accès aux informations de schéma

Deux procédés permettent d'obtenir des informations sur les éléments disponibles sur le serveur. Ces informations, appelées métadonnées ou informations de schéma, indiquent quelles tables et quelles procédures stockées sont accessibles sur le serveur et donnent des précisions sur ces tables et ces procédures stockées (comme les champs contenus dans une table, les index ayant été définis, les paramètres utilisés par une procédure stockée).

Le procédé le plus simple pour obtenir ces métadonnées consiste à utiliser les méthodes de *TSQLConnection*. Ces méthodes remplissent une liste de chaînes ou un objet liste existant avec les noms de tables, de procédures stockées, de champs ou d'index, ou avec des descripteurs de paramètre. Cette technique s'apparente à celle que vous utilisez pour remplir des listes avec des métadonnées pour tout autre composant connexion de base de données. Ces méthodes sont décrites dans "Obtention de métadonnées" à la page 17-14.

Si vous avez besoin d'informations de schéma plus détaillées, vous pouvez remplir un ensemble de données unidirectionnel avec des métadonnées. Au lieu d'une simple liste, l'ensemble de données unidirectionnel est rempli avec des informations de schéma, dans lesquelles chaque enregistrement représente une table, une procédure stockée, un index, un champ ou un paramètre unique.

Récupération de métadonnées dans un ensemble de données unidirectionnel

Pour remplir un ensemble de données unidirectionnel avec des métadonnées du serveur de base de données, vous devez d'abord indiquer quelles sont les données que vous voulez, en utilisant la méthode *SetSchemaInfo*. *SetSchemaInfo* accepte trois paramètres :

- Le type d'information de schéma (métadonnées) que vous voulez lire. Cela peut être une liste de tables (*stTables*), une liste de tables système (*stSysTables*), une liste de procédures stockées (*stProcedures*), une liste de champs dans une table (*stColumns*), une liste d'index (*stIndexes*) ou la liste des paramètres utilisés par une procédure stockée (*stProcedureParams*). Chaque type d'information utilise un ensemble de champs spécifique pour décrire les éléments de la liste. Pour plus de détails sur les structures de ces ensembles de données, voir "Structure des ensembles de métadonnées" à la page 22-15.
- Si vous recherchez des informations sur des champs des index ou les paramètres d'une procédure stockée, le nom de la table ou de la procédure stockée auxquels elles s'appliquent. Si vous recherchez n'importe quel autre type d'informations de schéma, ce paramètre est nil.
- Un modèle à respecter pour chaque nom renvoyé. Ce modèle est un modèle SQL, comme 'Cust%', qui utilise les jokers '%' (correspondant à une chaîne de n'importe quels caractères et de n'importe quelle longueur) et '_' (correspondant à un seul caractère quelconque). Pour utiliser un pourcentage ou un souligné dans un modèle pour sa valeur normale, doublez le caractère

(%% ou __). Si vous ne voulez pas utiliser de modèle, ce paramètre peut être nil.

Remarque Si vous lisez des informations de schéma concernant les tables (*stTables*), les informations de schéma résultantes peuvent décrire les tables ordinaires, les tables système, les vues et/ou les synonymes, selon la valeur de la propriété *TableScope* de la connexion SQL.

L'appel suivant demande une table listant toutes les tables système (les tables du serveur qui contiennent des métadonnées) :

```
SQLDataSet1.SetSchemaInfo(stSysTable, '', '');
```

Lorsque vous ouvrez l'ensemble de données après cet appel de *SetSchemaInfo*, il possède un enregistrement pour chaque table, dont les colonnes fournissent le nom de table, le type, le nom de schéma, etc. Si le serveur n'utilise pas de tables système pour stocker les métadonnées (par exemple MySQL), l'ensemble de données ne contient aucun enregistrement lorsque vous l'ouvrez.

L'exemple précédent n'utilisait que le premier paramètre. Supposons que vous vouliez obtenir la liste des paramètres utilisés en entrée de la procédure stockée nommée 'MyProc'. Supposons également que la personne ayant écrit cette procédure se soit servie d'un préfixe dans le nom de tous les paramètres pour indiquer s'il s'agit d'un paramètre en entrée ou en sortie ('inName', 'outValue', etc.). Vous pourriez appeler *SetSchemaInfo* comme suit :

```
SQLDataSet1.SetSchemaInfo(stProcedureParams, 'MyProc', 'in%');
```

L'ensemble de données résultant est une table de paramètres d'entrée dont les colonnes décrivent les propriétés de chaque paramètre.

Lecture des données après l'utilisation de l'ensemble de données pour des métadonnées

Après un appel à *SetSchemaInfo*, il y a deux façons de revenir à l'exécution des requêtes ou des procédures stockées avec l'ensemble de données :

- Changez la propriété *CommandText*, en spécifiant la requête, la table ou la procédure stockée à partir de laquelle vous voulez lire des données.
- Appelez *SetSchemaInfo*, en définissant le premier paramètre par *stNoSchema*. Dans ce cas, l'ensemble de données se remet à lire les données spécifiées par la valeur en cours de *CommandText*.

Structure des ensembles de métadonnées

Pour chaque type de métadonnées auquel vous pouvez accéder en utilisant *TSQLDataSet*, il existe un ensemble de colonnes (champs) qui sont remplies par les informations sur les éléments du type concerné.

Informations sur les tables

Lorsque vous demandez des informations sur les tables (*stTables* ou *stSysTables*), l'ensemble de données résultant comprend un enregistrement pour chaque table. Ces enregistrements possèdent les colonnes suivantes :

Tableau 22.1 Colonnes des tables de métadonnées concernant les tables

Nom de colonne	Type de champ	Contenu
RECNO	ftInteger	Un numéro d'enregistrement identifiant chaque enregistrement de manière unique.
CATALOG_NAME	ftString	Le nom du catalogue (la base de données) contenant la table. C'est le même que le paramètre *Database* pour un composant de connexion SQL.
SCHEMA_NAME	ftString	Le nom du schéma identifiant le propriétaire de la table.
TABLE_NAME	ftString	Le nom de la table. Ce champ détermine l'ordre de tri de l'ensemble de données.
TABLE_TYPE	ftInteger	Identifie le type de table. C'est la somme d'un nombre quelconque des valeurs suivantes : 1 : table 2 : vue 4 : table système 8 : synonyme 16 : table temporaire 32 : table locale.

Informations sur les procédures stockées

Lorsque vous demandez des informations sur les procédures stockées (*stProcedures*), l'ensemble de données résultant comprend un enregistrement pour chaque procédure stockée. Ces enregistrements possèdent les colonnes suivantes :

Tableau 22.2 Colonnes des tables de métadonnées concernant les procédures stockées

Nom de colonne	Type de champ	Contenu
RECNO	ftInteger	Un numéro d'enregistrement identifiant chaque enregistrement de manière unique.
CATALOG_NAME	ftString	Le nom du catalogue (la base de données) contenant la procédure stockée. C'est le même que le paramètre *Database* pour un composant de connexion SQL.
SCHEMA_NAME	ftString	Le nom du schéma identifiant le propriétaire de la procédure stockée.
PROC_NAME	ftString	Le nom de la procédure stockée. Ce champ détermine l'ordre de tri de l'ensemble de données.
PROC_TYPE	ftInteger	Identifie le type de procédure stockée. C'est la somme d'un nombre quelconque des valeurs suivantes : 1 : procédure 2 : fonction 4 : paquetage 8 : procédure système

Tableau 22.2 Colonnes des tables de métadonnées concernant les procédures stockées

Nom de colonne	Type de champ	Contenu
IN_PARAMS	ftSmallint	Le nombre de paramètres d'entrée.
OUT_PARAMS	ftSmallint	Le nombre de paramètres de sortie.

Informations sur les champs

Lorsque vous demandez des informations sur les champs d'une table particulière (*stColumns*), l'ensemble de données résultant comprend un enregistrement pour chaque champ. Ces enregistrements possèdent les colonnes suivantes :

Tableau 22.3 Colonnes des tables de métadonnées concernant les champs

Nom de colonne	Type de champ	Contenu
RECNO	ftInteger	Un numéro d'enregistrement identifiant chaque enregistrement de manière unique.
CATALOG_NAME	ftString	Le nom du catalogue (la base de données) contenant la table dont vous voulez connaître les champs. C'est le même que le paramètre *Database* pour un composant de connexion SQL.
SCHEMA_NAME	ftString	Le nom du schéma identifiant le propriétaire du champ.
TABLE_NAME	ftString	Le nom de la table contenant les champs.
COLUMN_NAME	ftString	Le nom du champ. Cette valeur détermine l'ordre de tri de l'ensemble de données.
COLUMN_POSITION	ftSmallint	La position de la colonne dans sa table.
COLUMN_TYPE	ftInteger	Identifie le type de valeur contenue dans le champ. C'est la somme d'un nombre quelconque des valeurs suivantes : 1 : identificateur de ligne 2 : version de ligne 4 : champ à incrémentation automatique 8 : Champ possédant une valeur par défaut
COLUMN_DATATYPE	ftSmallint	Le type de données de la colonne. C'est une des constantes de types de champs logiques définies dans sqllinks.pas.
COLUMN_TYPENAME	ftString	Une chaîne décrivant le type de données. C'est la même que celle contenue dans COLUMN_DATATYPE et COLUMN_SUBTYPE, mais sous une forme utilisée par certaines instructions DDL.
COLUMN_SUBTYPE	ftSmallint	Un sous-type du type de données de la colonne. C'est une des constantes de sous-types logiques définies dans sqllinks.pas.
COLUMN_PRECISION	ftInteger	La taille du type de champ (nombre de caractères dans une chaîne, octets dans un champ d'octets, chiffres significatifs dans une valeur BCD, membres d'un champ ADT, etc.).

Tableau 22.3 Colonnes des tables de métadonnées concernant les champs (suite)

Nom de colonne	Type de champ	Contenu
COLUMN_SCALE	ftSmallint	Le nombre de chiffres à droite de la décimale dans les valeurs BCD, ou de descendants dans les champs ADT et tableau.
COLUMN_LENGTH	ftInteger	Le nombre d'octets nécessaires pour stocker les valeurs des champs.
COLUMN_NULLABLE	ftSmallint	Un booléen indiquant si le champ peut être laissé vierge (0 signifie que le champ exige une valeur).

Informations sur les index

Lorsque vous demandez des informations sur les index d'une table particulière (stIndexes), l'ensemble de données résultant comprend un enregistrement pour chaque champ de chaque enregistrement. (Les index multi-enregistrements sont décrit par plusieurs enregistrements.) L'ensemble de données possède les colonnes suivantes :

Tableau 22.4 Colonnes des tables de métadonnées concernant les index

Nom de colonne	Type de champ	Contenu
RECNO	ftInteger	Un numéro d'enregistrement identifiant chaque enregistrement de manière unique.
CATALOG_NAME	ftString	Le nom du catalogue (la base de données) contenant l'index. C'est le même que le paramètre *Database* pour un composant de connexion SQL.
SCHEMA_NAME	ftString	Le nom du schéma identifiant le propriétaire de l'index.
TABLE_NAME	ftString	Le nom de la table pour laquelle est défini l'index.
INDEX_NAME	ftString	Le nom de l'index. Ce champ détermine l'ordre de tri de l'ensemble de données.
PKEY_NAME	ftString	Indique le nom de la clé primaire.
COLUMN_NAME	ftString	Le nom du champ (colonne) dans l'index.
COLUMN_POSITION	ftSmallint	La position de ce champ dans l'index.
INDEX_TYPE	ftSmallint	Identifie le type d'index. C'est la somme d'un nombre quelconque des valeurs suivantes : 1 : non unique 2 : unique 4 : clé primaire
SORT_ORDER	ftString	Indique que l'index est ascendant (a) ou descendant (d).
FILTER	ftString	Décrit une condition de filtre limitant les enregistrements indexés.

Informations sur les paramètres des procédures stockées

Lorsque vous demandez des informations sur les paramètres d'une procédure stockée (*stProcedureParams*), l'ensemble de données résultant comprend un enregistrement pour chaque paramètre. Ces enregistrements possèdent les colonnes suivantes :

Tableau 22.5 Colonnes des tables de métadonnées concernant les paramètres

Nom de colonne	Type de champ	Contenu
RECNO	ftInteger	Un numéro d'enregistrement identifiant chaque enregistrement de manière unique.
CATALOG_NAME	ftString	Le nom du catalogue (la base de données) contenant la procédure stockée. C'est le même que le paramètre *Database* pour un composant de connexion SQL.
SCHEMA_NAME	ftString	Le nom du schéma identifiant le propriétaire de la procédure stockée.
PROC_NAME	ftString	Le nom de la procédure stockée contenant le paramètre.
PARAM_NAME	ftString	Le nom du paramètre. Ce champ détermine l'ordre de tri de l'ensemble de données.
PARAM_TYPE	ftSmallint	Identifie le type de paramètre. C'est le même que la propriété *ParamType* d'un objet *TParam*.
PARAM_DATATYPE	ftSmallint	Le type de données du paramètre. C'est une des constantes de types de champs logiques définies dans sqllinks.pas.
PARAM_SUBTYPE	ftSmallint	Un sous-type du type de données du paramètre. C'est une des constantes de sous-types logiques définies dans sqllinks.pas.
PARAM_TYPENAME	ftString	Une chaîne décrivant le type de données. C'est la même que celle contenue dans PARAM_DATATYPE et PARAM_SUBTYPE, mais sous une forme utilisée par certaines instructions DDL.
PARAM_PRECISION	ftInteger	Le nombre maximal de chiffres dans les valeurs en virgule flottante ou dans les octets (pour les chaînes et les champs d'octets).
PARAM_SCALE	ftSmallint	Le nombre de chiffres à droite de la décimale dans les valeurs en virgule flottante.
PARAM_LENGTH	ftInteger	Le nombre d'octets nécessaires pour stocker les valeurs des paramètres.
PARAM_NULLABLE	ftSmallint	Un booléen indiquant si le paramètre peut être laissé vierge (0 signifie que le paramètre exige une valeur).

Débogage d'applications dbExpress

Lors du débogage de votre application de base de données, il peut s'avérer utile de contrôler les messages SQL envoyés à la base de données et reçus de celle-ci par le biais de votre composant connexion, y compris ceux automatiquement générés (par exemple, par un composant fournisseur ou par le pilote *dbExpress*).

Utilisation de TSQLMonitor pour contrôler les commandes SQL

TSQLConnection utilise un composant compagnon, *TSQLMonitor*, pour intercepter ces messages et les enregistrer dans une liste de chaînes. *TSQLMonitor* fonctionne comme l'utilitaire moniteur SQL que vous pouvez utiliser avec le BDE, à la différence qu'il contrôle uniquement les commandes impliquant un seul composant *TSQLConnection* plutôt que toutes les commandes gérées par *dbExpress*.

Pour utiliser *TSQLMonitor* :

1. Ajoutez un composant *TSQLMonitor* à la fiche ou au module de données contenant le composant *TSQLConnection* dont vous souhaitez contrôler les commandes SQL.
2. Attribuez à sa propriété *SQLConnection* le composant *TSQLConnection*.
3. Attribuez la valeur *True* à la propriété *Active* du moniteur SQL.

A mesure que les commandes SQL sont envoyées au serveur, la propriété *TraceList* du moniteur SQL est automatiquement mise à jour afin de présenter toutes les commandes SQL interceptées.

Vous pouvez enregistrer cette liste dans un fichier en spécifiant une valeur pour la propriété *FileName* puis en attribuant à la propriété *AutoSave* la valeur *True*. *AutoSave* oblige le moniteur SQL à enregistrer le contenu de la propriété *TraceList* dans un fichier à chaque consignation d'un nouveau message.

Si vous voulez épargner à votre système la charge liée à l'enregistrement d'un fichier à chaque consignation, vous pouvez utiliser le gestionnaire d'événement *OnLogTrace* afin de n'enregistrer des fichiers qu'à la suite d'un certain nombre de consignations de messages. Par exemple, le gestionnaire d'événement suivant enregistre le contenu de *TraceList* tous les 10 messages et efface le journal après l'avoir enregistré afin de limiter la longueur de la liste :

```
procedure TForm1.SQLMonitor1LogTrace(Sender: TObject; CBInfo: Pointer);
var
  LogFileName: string;
begin
  with Sender as TSQLMonitor do
  begin
    if TraceCount = 10 then
    begin
      LogFileName := 'c:\log' + IntToStr(Tag) + '.txt';
      Tag := Tag + 1; {permet de nommer différemment le fichier journal suivant }
      SaveToFile(LogFileName);
      TraceList.Clear; { efface la liste }
    end;
  end;
end;
```

Remarque Le gestionnaire d'événement précédent vous permet également d'enregistrer une liste qui s'avère partielle (contenant moins de 10 entrées) à l'arrêt de l'application.

Utilisation d'un callback pour contrôler les commandes SQL

Au lieu d'utiliser *TSQLMonitor*, vous pouvez personnaliser la façon dont votre application assure le suivi des commandes SQL en recourant à la méthode *SetTraceCallbackEvent* du composant connexion SQL. *SetTraceCallbackEvent* accepte deux paramètres : un callback de type *TSQLCallbackEvent* et une valeur définie par l'utilisateur transmise à la fonction de callback.

La fonction de callback accepte deux paramètres, *CallType* et *CBInfo* :

- *CallType* est réservé à un usage ultérieur.
- *CBInfo* est un pointeur vers un enregistrement qui comprend la catégorie (la même que *CallType*), le texte de la commande SQL et la valeur définie par l'utilisateur transmise à la méthode *SetTraceCallbackEvent*.

Le callback renvoie une valeur de type *CBRType*, généralement *cbrUSEDEF*.

Le pilote *dbExpress* appelle votre callback chaque fois que le composant connexion SQL transmet une commande au serveur ou que celui-ci renvoie un message d'erreur.

Attention N'appelez pas *SetTraceCallbackEvent* si l'objet *TSQLConnection* est associé à un composant *TSQLMonitor*. *TSQLMonitor* utilise le mécanisme de callback pour fonctionner et *TSQLConnection* ne peut prendre en charge qu'un callback à la fois.

Chapitre

23

Utilisation d'ensembles de données client

Les ensembles de données client sont des ensembles de données spécialisés qui stockent toutes leurs données en mémoire. La manipulation des données stockées en mémoire est possible grâce à midaslib.dcu ou midas.dll. Le format utilisé par les ensembles de données client pour le stockage des données est auto-contenu et facilement transportable ; c'est ce qui permet aux ensembles de données client de :

- Lire et écrire dans des fichiers dédiés sur le disque, en agissant comme un ensemble de données basé sur des fichiers. Les propriétés et méthodes prenant en charge ce mécanisme sont décrites dans "Utilisation d'un ensemble de données client avec des données basées sur des fichiers" à la page 23-39.
- Mettre en cache les mises à jour des données d'un serveur de base de données. Les fonctionnalités des ensembles de données client prenant en charge les mises à jour en cache sont décrites dans "Utilisation d'un ensemble de données client pour mettre en cache les mises à jour" à la page 23-18.
- Représenter les données dans la partie client d'une application multiniveau. Pour fonctionner ainsi, l'ensemble de données client doit fonctionner avec un fournisseur externe, comme décrit dans "Utilisation d'un ensemble de données client avec un fournisseur" à la page 23-29. Pour plus d'informations sur ces méthodes, voir chapitre 25, "Création d'applications multiniveaux".
- Représenter les données issue d'une source autre qu'un ensemble de données. Comme un ensemble de données client peut utiliser les données provenant d'un fournisseur externe, des fournisseurs spécialisés peuvent adapter une grande variété de sources d'informations pour les faire fonctionner avec des ensembles de données client. Par exemple, vous pouvez utiliser un fournisseur XML pour permettre à un ensemble de données client de représenter les informations contenues dans un document XML.

Que vous utilisiez les ensembles de données client avec des données basées sur des fichiers, pour mettre en cache les mises à jour, avec des données issues d'un fournisseur externe (comme avec un document XML ou dans une application multiniveau), ou en combinant toutes ces approches (comme dans une application de modèle "briefcase"), vous bénéficierez de la vaste gamme des fonctionnalités pour ensemble de données client lors de la manipulation des données.

Manipulation des données avec un ensemble de données client

Comme tout ensemble de données, les ensembles de données client vous permettent de fournir les données aux contrôles orientés données à l'aide d'un composant source de données. Voir chapitre 15, "Utilisation de contrôles de données", pour plus d'informations sur l'affichage des informations de bases de données dans les contrôles orientés données.

Les ensembles de données client implémentent toutes les propriétés et les méthodes héritées de *TDataSet*. Pour une présentation complète du comportement générique des ensembles de données, voir chapitre 18, "Présentation des ensembles de données".

De plus, les ensembles de données client implémentent nombre des fonctionnalités courantes des ensembles de données de type table comme :

- Tri des enregistrements avec des index.
- Utilisation d'index pour chercher des enregistrements.
- Limitation des enregistrements avec des portées.
- Création de relations maître/détail.
- Contrôle des accès en lecture/écriture aux tables
- Création de l'ensemble de données sous-jacent
- Vidage de l'ensemble de données
- Synchronisation des ensembles de données client

Pour plus de détails sur ces fonctionnalités, voir "Utilisation d'ensembles de données de type table" à la page 18-29.

Les ensembles de données client diffèrent des autres ensembles de données en ce sens qu'ils stockent toutes leurs données en mémoire. C'est pourquoi la prise en charge de certaines fonctions de bases de données peut élargir leur champ d'action et impliquer de nouvelles contraintes. Ce chapitre décrit certaines de ces fonctions communes et les différences introduites par les ensembles de données client.

Navigation parmi les données des ensembles de données client

Si une application utilise des contrôles orientés données standard, un utilisateur peut se déplacer parmi les enregistrements d'un ensemble de données client en utilisant le comportement intégré de ces contrôles. Il est aussi possible d'avoir recours au code pour se déplacer parmi les enregistrements ; utilisez pour cela

les méthodes standard comme *First*, *Last*, *Next* et *Prior*. Pour plus d'informations sur ces méthodes, voir "Navigation dans les ensembles de données" à la page 18-6.

Au contraire de la majorité des ensembles de données, les ensembles de données client peuvent positionner le curseur sur un enregistrement spécifique dans l'ensemble de données en utilisant la propriété *RecNo*. Habituellement, une application utilise *RecNo* pour déterminer le numéro de l'enregistrement en cours. Mais, les ensembles de données client peuvent définir *RecNo* par un numéro d'enregistrement afin que cet enregistrement devienne l'enregistrement en cours.

Limitation des enregistrements affichés

Pour restreindre l'accès aux données de manière temporaire, les applications peuvent définir des portées et des filtres. Lorsqu'une portée ou un filtre est appliqué, l'ensemble de données client n'affiche pas toutes les données placées dans sa mémoire cache. A la place, il affiche uniquement celles correspondant aux conditions de la portée ou du filtre. Pour plus d'informations sur l'utilisation de filtres, voir "Affichage et édition d'ensembles de données en utilisant des filtres" à la page 18-14. Pour plus d'informations sur les portées, voir "Limitation des enregistrements avec des portées" à la page 18-35.

Avec la majorité des ensembles de données, les chaînes des filtres sont analysées et traduites en commandes SQL qui sont ensuite implémentées sur le serveur de base de données. De ce fait, le dialecte SQL du serveur limite les opérations utilisées dans les chaînes de filtre. Les ensembles de données client implémentent leur propre support des filtres, qui comprend davantage d'opérations que celui des autres ensembles de données. Par exemple, avec un ensemble de données client, les expressions de filtre peuvent comprendre des opérateurs de chaînes renvoyant des sous-chaînes, des opérateurs qui décomposent des valeurs de date et d'heure, etc. Les ensembles de données client autorisent l'utilisation des filtres sur les champs BLOB et les types de champs complexes comme les champs ADT et les champs de tableaux.

Voici les divers opérateurs et fonctions qu'un ensemble de données client peut utiliser dans les filtres, accompagnés d'une comparaison avec les autres ensembles de données prenant en charge les filtres :

Tableau 23.1 Support des filtres dans les ensembles de données client

Opérateur ou fonction	Exemple	Pris en charge par d'autres ensembles de données	Commentaire
Comparaisons			
=	State = 'CA'	Oui	
<>	State <> 'CA'	Oui	
>=	DateArrivée >= '1/1/1998'	Oui	
<=	Total <= 100000	Oui	

Tableau 23.1 Support des filtres dans les ensembles de données client (suite)

Opérateur ou fonction	Exemple	Pris en charge par d'autres ensembles de données	Commentaire
>	Pourcentage > 50	Oui	
<	Champ1 < Champ2	Oui	
BLANK	State <> 'CA' or State = BLANK	Oui	Les enregistrements vierges ne s'affichent que s'ils sont explicitement inclus dans le filtre.
IS NULL	Champ1 IS NULL	Non	
IS NOT NULL	Champ1 IS NOT NULL	Non	
Opérateurs logiques			
and	State = 'CA' and Country = 'US'	Oui	
or	State = 'CA' or State = 'MA'	Oui	
not	not (State = 'CA')	Oui	
Opérateurs arithmétiques			
+	Total + 5 > 100	Dépend du pilote	S'applique aux nombres, aux chaînes ou aux dates (heures) plus un nombre.
-	Champ1 - 7 <> 10	Dépend du pilote	S'applique aux nombres, aux dates ou aux dates (heures) moins un nombre.
*	Remise * 100 > 20	Dépend du pilote	S'applique aux nombres uniquement.
/	Remise > Total / 5	Dépend du pilote	S'applique aux nombres uniquement.
Fonctions String			
Upper	Upper(Champ1) = 'TOUJOURS'	Non	
Lower	Lower(Champ1 + Champ2) = 'josp'	Non	
Substring	Substring(ChampDate,8) = '1998' Substring(ChampDate,1,3) = 'JAN'	Non	La valeur va de la position du second argument à la fin ou au nombre de caractères dans le troisième argument. Le premier caractère a la position 1.

Tableau 23.1 Support des filtres dans les ensembles de données client (suite)

Opérateur ou fonction	Exemple	Pris en charge par d'autres ensembles de données	Commentaire
Trim	Trim(Champ1 + Champ2) Trim(Champ1, '-')	Non	Supprime le troisième argument au début et à la fin. S'il n'y a pas de troisième argument, supprime les espaces.
TrimLeft	TrimLeft(ChampString) TrimLeft(Champ1, '$') <> ''	Non	Voir Trim.
TrimRight	TrimRight(ChampString) TrimRight(Champ1, '.') <> ''	Non	Voir Trim.
Fonctions DateTime			
Year	Year(ChampDate) = 2001	Non	
Month	Month(ChampDate) <> 12	Non	
Day	Day(ChampDate) = 1	Non	
Hour	Hour(ChampDate) < 16	Non	
Minute	Minute(ChampDate) = 0	Non	
Second	Second(ChampDate) = 30	Non	
GetDate	GetDate - DateField > 7	Non	Représente la date et l'heure en cours.
Date	ChampDate = Date(GetDate)	Non	Renvoie la partie date d'une valeur date/heure.
Time	ChampHeure > Time(GetDate)	Non	Renvoie la partie heure d'une valeur date/heure.
Divers			
Like	Memo LIKE '%filters%'	Non	Fonctionne comme SQL-92 sans la clause ESC. Lorsqu'elle est appliquée à des champs BLOB, FilterOptions détermine la prise en compte de la casse.
In	Day(ChampDate) in (1,7)	Non	Fonctionne comme SQL-92. Le second argument est une liste de valeurs toutes du même type.
*	State = 'M*'	Oui	Caractère générique pour les comparaisons partielles.

Lorsque vous appliquez des portées ou des filtres, l'ensemble de données client stockera tous les enregistrements en mémoire. La portée ou le filtre détermine simplement quels sont les enregistrements accessibles aux contrôles pour la navigation ou l'affichage des données issues de l'ensemble client.

Remarque Lors de l'extraction de données en provenance d'un fournisseur, vous pouvez aussi limiter les données stockées par l'ensemble de données client en transmettant des paramètres au fournisseur. Pour plus d'informations sur les portées, voir "Limitation des enregistrements avec des paramètres" à la page 23-34.

Edition des données

Les ensembles de données client représentent leurs données sous la forme d'un paquet de données en mémoire. Ce paquet est la valeur de la propriété *Data* de l'ensemble de données client. Par défaut, cependant, les modifications ne sont pas enregistrées dans la propriété *Data*. Les insertions, suppressions et modifications (faites par l'utilisateur ou programmées) sont stockées dans un journal interne de modifications, représenté par la propriété *Delta*. L'utilisation d'un journal de modifications a une double finalité :

- Le journal de modifications est nécessaire pour appliquer des mises à jour à un serveur de base de données ou à un composant fournisseur externe.
- Le journal de modifications constitue un outil élaboré pour l'annulation de modifications.

La propriété *LogChanges* vous permet de désactiver temporairement le journal de modifications. Lorsque *LogChanges* vaut *True*, les modifications sont enregistrées dans le journal. Si *LogChanges* vaut *False*, les modifications sont directement réalisées dans la propriété *Data*. Vous pouvez désactiver le journal de modifications dans les applications basées sur des fichiers si vous n'avez pas besoin de la fonctionnalité d'annulation.

Les modifications restent dans le journal de modifications jusqu'à ce qu'elles soient supprimées par l'application. Les applications suppriment les modifications lors des tâches suivantes :

- Annulation des modifications
- Enregistrement des modifications

Remarque L'enregistrement de l'ensemble de données client dans un fichier ne supprime pas les modifications du journal. Lorsque vous rechargez l'ensemble de données, les propriétés *Data* et *Delta* sont inchangées par rapport à ce qu'elles étaient lors de l'enregistrement des données.

Annulation des modifications

Même si la version originale d'un enregistrement reste inchangée dans la propriété *Data*, l'utilisateur voit la toute dernière version de l'enregistrement à chaque fois qu'il modifie l'enregistrement, le laisse et le sélectionne à nouveau. Si un utilisateur ou une application modifie plusieurs fois un enregistrement, chaque version de l'enregistrement est stockée dans une entrée différente du journal de modifications.

La possibilité de stocker chaque modification apportée à un enregistrement permet de supporter les opérations d'annulation à plusieurs niveaux :

- Pour supprimer la dernière modification apportée à un enregistrement, appelez *UndoLastChange*. Cette méthode accepte un paramètre booléen, *FollowChange*, qui indique si le curseur doit être repositionné sur l'enregistrement restauré (*True*) ou s'il doit être laissé sur l'enregistrement en cours (*False*). Si un enregistrement a subi plusieurs modifications, chaque appel à *UndoLastChange* annule un niveau de modification. *UndoLastChange* renvoie une valeur booléenne indiquant si l'annulation a réussi ou échoué. En cas de réussite, *UndoLastChange* renvoie *True*. Utilisez la propriété *ChangeCount* pour déterminer si d'autres modifications doivent être annulées. *ChangeCount* indique le nombre de modifications stockées dans le journal de modifications.
- Plutôt que de supprimer chaque niveau de modification l'un après l'autre, vous pouvez les supprimer tous en une seule fois. Pour supprimer toutes les modifications apportées à un enregistrement, sélectionnez-le et appelez *RevertRecord*. Cette méthode supprime toutes les modifications apportées à l'enregistrement en cours.
- Pour restaurer un enregistrement supprimé, définissez d'abord la propriété *StatusFilter* par [*usDeleted*], ce qui rend "visibles" les enregistrements supprimés. Ensuite, naviguez jusqu'à l'enregistrement que vous voulez restaurer et appelez *RevertRecord*. Enfin, restaurez la propriété *StatusFilter* en [*usModified*, *usInserted*, *usUnmodified*] de sorte que la version modifiée de l'ensemble de données (contenant maintenant l'enregistrement restauré) soit à nouveau visible.
- A tout moment pendant les modifications, vous pouvez enregistrer l'état courant du journal de modifications à l'aide de la propriété *SavePoint*. La lecture de *SavePoint* renvoie un marqueur à la position courante dans le journal de modifications. Ultérieurement, si vous souhaitez annuler toutes les modifications opérées depuis la lecture du point de sauvegarde, attribuez à *SavePoint* la valeur lue. Votre application peut obtenir des valeurs pour plusieurs points de sauvegarde. Toutefois, lorsque vous sauvegardez le journal de modifications au niveau d'un point de sauvegarde, les valeurs de tous les points de sauvegarde postérieurement lues par votre application ne sont plus valides.
- Toutes les modifications enregistrées dans le journal de modifications peuvent être abandonnées en appelant *CancelUpdates*. Cette méthode efface le journal de modifications et annule tous les changements apportés à l'ensemble des enregistrements. *CancelUpdates* doit être employée avec précaution. Après avoir appelé la méthode *CancelUpdates*, il est impossible de récupérer les modifications.

Enregistrement des modifications

Les ensembles de données client utilisent différents mécanismes pour intégrer les modifications à partir du journal de modifications, selon qu'ils stockent leurs données dans un fichier ou représentent des données émanant d'un serveur.

Quel que soit le mécanisme utilisé, le journal de modifications est automatiquement vidé lorsque les mises à jour ont été intégrées.

Les applications basées sur des fichiers peuvent simplement fusionner les modifications dans la mémoire cache locale représentée par la propriété *Data*. Elles ne sont pas concernées par la résolution de modifications locales à partir de changements réalisés par d'autres utilisateurs. Pour fusionner le journal de modifications dans la propriété *Data*, appelez la méthode *MergeChangeLog*. "Fusion des modifications dans les données" à la page 23-41 décrit ce processus.

Vous ne pouvez pas utiliser *MergeChangeLog* si vous utilisez l'ensemble de données client pour mettre en cache des mises à jour ou pour représenter les données issues d'un composant fournisseur externe. Les informations contenues dans le journal de modifications sont requises pour répercuter les enregistrements mis à jour sur les données stockées dans la base de données (ou l'ensemble de données source). A la place, appelez *ApplyUpdates*, qui essaie d'écrire les modifications sur le serveur de base de données ou l'ensemble de données source, et met à jour la propriété *Data* uniquement lorsque les modifications ont été validées avec succès. Voir "Application des mises à jour" à la page 23-24, pour plus d'informations sur ce processus.

Définition de contraintes pour les valeurs des données

Les ensembles de données client peuvent imposer des contraintes aux modifications qu'un utilisateur peut effectuer sur les données. Ces contraintes sont appliquées lorsque l'utilisateur essaie de valider des changements dans le journal de modifications. Vous pouvez toujours fournir des contraintes personnalisées. Elles vous permettent d'imposer vos propres limites, définies par l'application, sur les valeurs qu'un utilisateur peut valider dans un ensemble de données client.

De plus, lorsque des ensembles de données client représentent des données d'un serveur accédé en utilisant le BDE, elles imposent également les contraintes sur les données importées du serveur de base de données. Si l'ensemble de données client fonctionne avec un composant fournisseur externe, le fournisseur peut contrôler si ces contraintes sont envoyées à l'ensemble de données client, et l'ensemble de données client peut contrôler s'il les utilise. Pour savoir comment le fournisseur contrôle si les contraintes sont incluses dans les paquets de données, voir "Gestion des contraintes du serveur" à la page 24-15. Pour savoir comment et pourquoi l'ensemble de données client peut désactiver l'application des contraintes du serveur, voir "Gestion des contraintes liées au serveur" à la page 23-35.

Spécification de contraintes personnalisées

Vous pouvez utiliser les propriétés des composants champ de l'ensemble de données client pour imposer vos propres contraintes quant aux données que

l'utilisateur peut saisir. Chaque composant champ possède deux propriétés qui peuvent être utilisées pour spécifier des contraintes :

- La propriété *DefaultExpression* définit une valeur par défaut qui est attribuée au champ si l'utilisateur n'en saisit pas une. Remarquez que si le serveur de base de données ou l'ensemble de données source attribue aussi une expression par défaut au champ, celle de l'ensemble de données client est prioritaire car elle est attribuée avant que la mise à jour ne soit appliquée en retour sur le serveur de base de données ou dans l'ensemble de données source.
- La propriété *CustomConstraint* vous permet d'imposer une condition à remplir pour qu'une valeur de champ puisse être validée. Les contraintes personnalisées définies de cette façon sont appliquées en plus des contraintes importées du serveur. Pour plus d'informations sur la manipulation des contraintes personnalisées sur les composants champ, voir "Création de contrainte personnalisée" à la page 19-25.

Au niveau de l'enregistrement, vous pouvez spécifier des contraintes à l'aide de la propriété *Constraints* de l'ensemble de données client. *Constraints* est une collection d'objets *TCheckConstraint*, dans laquelle chacun d'eux représente une condition. Utilisez la propriété *CustomConstraint* d'un objet *TCheckConstraint* pour ajouter vos propres contraintes, qui sont vérifiées lorsque vous validez les enregistrements.

Tri et indexation

L'utilisation d'index présente plusieurs avantages pour vos applications :

- Ils permettent aux ensembles de données client de localiser les données rapidement.
- Ils permettent d'appliquer des portées pour limiter les enregistrements disponibles.
- Ils permettent à votre application de définir des relations entre les autres ensembles de données, telles que des tables de référence ou des liens maître/détail.
- Ils spécifient l'ordre dans lequel les enregistrements apparaissent.

Si un ensemble de données client représente les données d'un serveur ou utilise un fournisseur externe, il hérite d'un index et d'un ordre de tri par défaut, basés sur les données qu'il reçoit. L'index par défaut s'appelle DEFAULT_ORDER. Il est possible d'utiliser ce classement, mais il est impossible de modifier ou de supprimer l'index.

En plus de l'index par défaut, l'ensemble de données client gère un deuxième index, appelé CHANGEINDEX, à partir des enregistrements stockés dans le journal de modifications (propriété *Delta*). CHANGEINDEX classe tous les enregistrements de l'ensemble de données tels qu'ils apparaîtraient si les modifications de *Delta* étaient appliquées. CHANGEINDEX est basé sur l'ordre qu'il a hérité de DEFAULT_ORDER. Comme pour DEFAULT_ORDER, il est impossible de modifier ou de supprimer l'index CHANGEINDEX.

Vous pouvez utiliser d'autres index existants ou créer vos propres index. Les sections suivantes décrivent comment créer et utiliser des index avec des ensembles de données client.

Remarque Vous pouvez également vouloir revoir les documents sur les index dans les ensembles de données de type table, ce qui s'applique aussi aux ensembles de données client. Vous trouverez ces informations dans "Tri des enregistrements avec des index" à la page 18-30 et "Limitation des enregistrements avec des portées" à la page 18-35.

Ajout d'un nouvel index

Il y a trois manières d'ajouter des index à un ensemble de données client :

- Pour créer un index temporaire à l'exécution afin de trier les enregistrements de l'ensemble de données client, vous pouvez utiliser la propriété *IndexFieldNames*. Spécifiez les noms de champs en les séparant par des points-virgules. L'ordre des noms de champs dans la liste détermine leur ordre dans l'index.

  C'est la méthode d'ajout d'index la moins puissante. En effet, vous ne pouvez pas spécifier un index décroissant ou ne tenant pas compte des différences majuscules/minuscules et les index générés ne gèrent pas le regroupement. Ces index ne sont pas préservés quand vous fermez l'ensemble de données et ne sont pas enregistrés quand vous enregistrez l'ensemble de données client dans un fichier.

- Pour créer à l'exécution un index utilisable pour les regroupements, appelez la méthode *AddIndex*. *AddIndex* vous permet de spécifier les propriétés de l'index, dont :

  - Le nom de l'index. Il permet de permuter les index à l'exécution.
  - Les champs qui composent l'index. L'index utilise ces champs pour trier les enregistrements et localiser les enregistrements dont les champs indexés présentent une valeur particulière.
  - La façon dont l'index trie les enregistrements. Par défaut, les index imposent un ordre de tri croissant (selon la configuration de la machine). Cet ordre de tri par défaut tient compte de la casse. Vous pouvez définir des options pour que la totalité de l'index fasse la différence entre les majuscules et les minuscules ou pour trier par ordre décroissant. Vous pouvez aussi spécifier une liste de champs à trier sans tenir compte de la casse et une autre liste de champs à trier par ordre décroissant.
  - Le niveau par défaut de regroupement pris en charge par l'index.

  Les index créés avec *AddIndex* ne sont pas persistants après la fermeture de l'ensemble de données client. (C'est à dire qu'ils sont perdus lorsque vous rouvrez l'ensemble de données client). Vous ne pouvez pas appeler *AddIndex* quand l'ensemble de données client est fermé. Les index ajoutés en employant *AddIndex* ne sont pas enregistrés quand vous enregistrez l'ensemble de données client dans un fichier.

- La troisième méthode pour créer un index intervient lors de la création de l'ensemble de données client. Avant de créer l'ensemble de données client, spécifiez les index souhaités en utilisant la propriété *IndexDefs*. Les index sont alors créés en même temps que l'ensemble de données sous-jacent lors de l'appel de *CreateDataSet*. Pour plus d'informations sur la création des ensembles de données client, voir "Création et suppression des tables" à la page 18-44.

  Comme pour *AddIndex*, les index créés en même temps que l'ensemble de données prennent en charge le regroupement et peuvent être triés en ordre décroissant sur certains champs et ne pas prendre en compte les différences majuscules-minuscules pour d'autres. Les index créés de cette manière sont toujours conservés et enregistrés quand vous enregistrez l'ensemble de données client dans un fichier.

Conseil L'indexation et le tri peuvent s'effectuer sur des champs calculés en interne avec des ensembles de données client.

Suppression et permutation d'index

Pour supprimer un index ayant été créé pour un ensemble de données client, appelez *DeleteIndex* et spécifiez le nom de l'index à supprimer. Les index DEFAULT_ORDER et CHANGEINDEX ne peuvent pas être supprimés.

Lorsque plusieurs index sont disponibles, il est possible de changer d'index en utilisant la propriété *IndexName*. Lors de la phase de conception, les index disponibles peuvent être sélectionnés dans la liste déroulante de la propriété *IndexName* dans l'inspecteur d'objets.

Utilisation des index pour regrouper les données

Lorsque vous utilisez un index dans votre ensemble de données client, il impose automatiquement un ordre de tri sur les enregistrements. En raison de cet ordre de tri, des enregistrements adjacents contiennent généralement les mêmes valeurs dans les champs qui composent l'index. Par exemple, considérons la portion de table de commandes suivante indexée sur les champs SalesRep et Customer :

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

En raison de l'ordre de tri, les valeurs identiques dans la colonne SalesRep apparaissent regroupées. A l'intérieur des champs relatifs au représentant (SalesRep) 1, les valeurs identiques de la colonne des clients (Customer) apparaissent regroupées. En d'autres termes, les données sont regroupées par représentant (SalesRep) puis, dans le groupe SalesRep, par client (Customer). A chaque regroupement est associé un niveau. Dans notre exemple, le groupe

SalesRep est le niveau 1 (car il n'est imbriqué dans aucun autre groupe) et le groupe Customer est le niveau 2 (car il est imbriqué dans le groupe de niveau 1). Le niveau de regroupement correspond à l'ordre des champs dans l'index.

Les ensembles de données client vous permettent de déterminer la position de l'enregistrement en cours dans un niveau de regroupement donné. Cela permet à votre application d'afficher les enregistrements différemment, suivant qu'ils se trouvent en tête, au milieu ou à la fin d'un groupe. Par exemple, vous pouvez afficher une valeur de champ que si elle figure dans le premier enregistrement d'un groupe et éliminer ainsi les doublons. Le résultat est le suivant à partir de la table précédente :

SalesRep	Customer	OrderNo	Amount
1	1	5	100
		2	50
	2	3	200
		6	75
2	1	1	10
	3	4	200

Pour déterminer la position de l'enregistrement en cours dans un groupe, utilisez la méthode *GetGroupState*. *GetGroupState* accepte une valeur entière représentant le niveau du groupe et renvoie une valeur indiquant la position de l'enregistrement en cours dans le groupe (première position, dernière position ou aucune des deux positions).

Lorsque vous créez un index, vous pouvez spécifier le niveau de regroupement qu'il prend en charge (jusqu'au nombre de champs présents dans l'index). *GetGroupState* ne peut fournir d'information sur les groupes au delà de ce niveau, même si l'index classe les enregistrements sur d'autres champs.

Représentation des valeurs calculées

Comme pour tout ensemble de données, vous pouvez ajouter des champs calculés à votre ensemble de données client. Il s'agit de champs dont les valeurs sont calculées dynamiquement, généralement en fonction des valeurs d'autres champs du même enregistrement. Pour plus d'informations sur l'utilisation des champs calculés, voir "Définition d'un champ calculé" à la page 19-8.

Toutefois, les ensembles de données client vous permettent de mieux définir à quel moment les champs sont calculés par l'utilisation de champs calculés de façon interne. Pour plus d'informations sur les champs calculés de façon interne, voir "Utilisation de champs calculés de façon interne dans les ensembles de données client" ci-après.

Vous pouvez aussi indiquer aux ensembles de données client de créer des valeurs calculées qui résument les données de plusieurs enregistrements à l'aide des agrégats maintenus. Pour plus d'informations sur les agrégats maintenus, voir "Utilisation des agrégats maintenus" à la page 23-13.

Utilisation de champs calculés de façon interne dans les ensembles de données client

Dans les autres ensembles de données, votre application doit déterminer la valeur des champs calculés chaque fois que l'enregistrement change ou que l'utilisateur modifie l'un des champs de l'enregistrement en cours. Elle réalise cela dans un gestionnaire d'événement *OnCalcFields*.

Bien que vous puissiez utiliser ce procédé dans les ensembles de données client, ces derniers, en enregistrant les valeurs calculées dans leurs données, vous permettent de réduire au minimum le nombre de fois où les champs calculés doivent être recalculés. Lorsque les valeurs calculées sont enregistrées avec l'ensemble de données client, elles doivent toujours être recalculées lorsque l'utilisateur modifie l'enregistrement en cours mais votre application n'a pas besoin de recalculer les valeurs chaque fois que l'enregistrement en cours change. Pour enregistrer les valeurs calculées dans les données de l'ensemble de données client, utilisez des champs calculés de façon interne à la place de champs calculés.

Les champs calculés de façon interne, comme les champs calculés, sont calculés dans un gestionnaire d'événement *OnCalcFields*. Toutefois, vous pouvez optimiser votre gestionnaire d'événement en vérifiant la propriété *State* de votre ensemble de données client. Lorsque *State* vaut *dsInternalCalc*, vous devez recalculer les champs calculés de façon interne. Lorsque *State* vaut *dsCalcFields*, il suffit de recalculer les champs calculés ordinaires.

Pour utiliser des champs calculés de façon interne, vous devez définir les champs devant être calculés de façon interne avant de créer l'ensemble de données client. Selon que vous utilisez des champs persistants ou des définitions de champs, vous ferez cela d'une des façons suivantes :

- Si vous utilisez des champs persistants, définissez les champs devant être calculés de façon interne en sélectionnant InternalCalc dans l'éditeur de champs.
- Si vous utilisez des définitions de champs, attribuez la valeur *True* à la propriété *InternalCalcField* de la définition de champ adéquate.

Remarque D'autres types d'ensembles de données utilisent des champs calculés de façon interne. Mais, vous ne calculez pas les valeurs de ces champs dans un gestionnaire d'événement *OnCalcFields*. Elles sont automatiquement calculées par le BDE ou par le serveur de base de données distant.

Utilisation des agrégats maintenus

Les ensembles de données client permettent de résumer les données émanant de différents groupes d'enregistrements. Comme ces résumés sont automatiquement mis à jour au fur et à mesure que sont modifiées les données dans l'ensemble de données, ces données de résumé sont appelées "agrégats maintenus".

Dans leur forme la plus simple, les agrégats maintenus vous permettent d'obtenir des informations telles que la somme de toutes les valeurs d'une colonne de l'ensemble de données client. Ils sont suffisamment souples, toutefois,

pour supporter un large éventail de calculs résumé et déterminer des sous-totaux englobant différents groupes d'enregistrements définis par un champ de l'index supportant le regroupement.

Spécification d'agrégats

Pour spécifier que vous voulez opérer des calculs synthétiques à partir des enregistrements d'un ensemble de données client, utilisez la propriété *Aggregates*. *Aggregates* est un ensemble de spécifications d'agrégat (*TAggregate*). Vous pouvez ajouter des spécifications d'agrégat à votre ensemble de données client à l'aide de l'éditeur de collection lors de la conception ou en utilisant la méthode *Add* de *Aggregates* lors de l'exécution. Si vous souhaitez créer des composants champ pour les agrégats, créez des champs persistants pour les valeurs synthétisées dans l'éditeur de champs.

Remarque Lorsque vous créez des champs synthétisés, les objets agrégat appropriés sont automatiquement ajoutés à la propriété *Aggregates* de l'ensemble de données client. Ne les ajoutez pas de façon explicite lors de la création des champs persistants synthétisés. Pour plus de détails sur la création des champs persistants synthétisés, voir "Définition d'un champ agrégat" à la page 19-12.

Pour chaque agrégat, la propriété *Expression* indique le calcul synthétique qu'elle représente. *Expression* peut contenir une simple expression synthétique telle que

```
Sum(Champ1)
```

ou une expression complexe qui combine les informations de plusieurs champs, telle que

```
Sum(Qté * Prix) - Sum(MontantPayé)
```

Les expressions d'agrégat incluent un ou plusieurs opérateurs de synthèse du tableau suivant :

Tableau 23.2 Opérateurs de synthèse pour les agrégats maintenus

Opérateur	Rôle
Sum	Somme des valeurs d'un champ numérique ou d'une expression
Avg	Valeur moyenne d'un champ numérique ou date/heure ou d'une expression
Count	Spécification du nombre de valeurs exprimées pour un champ ou pour une expression
Min	Valeur minimale d'un champ chaîne, numérique ou date/heure ou d'une expression
Max	Valeur maximale d'un champ chaîne, numérique ou date/heure ou d'une expression

Les opérateurs de synthèse portent sur des valeurs de champ ou sur des expressions conçues à partir de valeurs de champ à l'aide des mêmes opérateurs que ceux utilisés pour la création de filtres. Vous ne pouvez pas, toutefois, imbriquer des opérateurs de synthèse. Vous pouvez créer des expressions avec des opérateurs à partir de valeurs synthétisées, ou de valeurs synthétisées et de constantes. Toutefois, vous ne pouvez pas combiner des valeurs synthétisées et des valeurs de champ, car de telles expressions sont ambiguës (rien n'indique

quel enregistrement doit fournir la valeur de champ). Ces règles sont illustrées dans les expressions suivantes :

```
Sum(Qty * Price)          {autorisé -- synthèse d'une expression portant sur des champs }
Max(Champ1) - Max(Champ2) {autorisé -- expression à partir de synthèses }
Avg(TauxRemise) * 100     {autorisé -- expression à partir d'une synthèse et d'une constante }
Min(Sum(Champ1))          {non autorisé -- synthèses imbriquées }
Count(Champ1) - Champ2    {non autorisé -- expression à partir d'une synthèse et d'un champ }
```

Agrégats de groupes d'enregistrements

Par défaut, les agrégats maintenus sont calculés afin qu'ils synthétisent tous les enregistrements de l'ensemble de données client. Toutefois, vous pouvez spécifier que l'opération ne porte que sur les enregistrements d'un groupe. Cela vous permet d'obtenir des synthèses intermédiaires, comme des sous-totaux impliquant des groupes d'enregistrements ayant une valeur de champ commune.

Pour spécifier un agrégat maintenu sur un groupe d'enregistrements, vous devez disposer d'un index à partir duquel peut s'opérer le regroupement. Voir "Utilisation des index pour regrouper les données" à la page 23-11, pour plus d'informations sur le regroupement.

Une fois que vous disposez d'un index qui regroupe les données en fonction de la synthèse que vous voulez opérer, spécifiez les propriétés *IndexName* et *GroupingLevel* d'agrégat pour indiquer l'index à utiliser et le groupe ou sous-groupe de cet index qui définit les enregistrements à synthétiser.

Par exemple, considérons la portion de table de commandes suivante triée par représentants (SalesRep) puis par clients (Customer) :

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

Le code suivant définit un agrégat maintenu qui indique le montant total des ventes réalisé par chaque représentant :

```
Agg.Expression := 'Sum(Amount)';
Agg.IndexName := 'SalesCust';
Agg.GroupingLevel := 1;
Agg.AggregateName := 'Total for Rep';
```

Pour ajouter un agrégat qui synthétise chaque client pour un représentant donné, créez un agrégat maintenu de niveau 2.

Les agrégats maintenus qui synthétisent un groupe d'enregistrements sont associés à un index spécifique. La propriété *Aggregates* peut inclure des agrégats qui utilisent différents index. Toutefois, seuls les agrégats qui synthétisent la totalité de l'ensemble de données client et ceux qui utilisent l'index en cours sont valides. La modification de l'index en cours détermine les agrégats valides. Pour déterminer les agrégats valides à un moment donné, utilisez la propriété *ActiveAggs*.

Obtention de valeurs d'agrégat

Pour obtenir la valeur d'un agrégat maintenu, appelez la méthode *Value* de l'objet *TAggregate* qui représente l'agrégat. *Value* renvoie l'agrégat maintenu du groupe qui contient l'enregistrement en cours de l'ensemble de données client.

Lorsque la synthèse porte sur la totalité de l'ensemble de données client, vous pouvez appeler *Value* à tout moment pour obtenir l'agrégat maintenu. Toutefois, lorsque la synthèse porte sur des informations regroupées, vous devez veiller à ce que l'enregistrement en cours se trouve dans le groupe à synthétiser. Aussi est-il judicieux d'obtenir les valeurs d'agrégat à des moments précis, comme lorsque vous vous positionnez sur le premier ou le dernier enregistrement d'un groupe. Utilisez la méthode *GetGroupState* pour déterminer la position de l'enregistrement en cours dans un groupe.

Pour afficher les agrégats maintenus dans les contrôles orientés données, utilisez l'éditeur de champs pour créer un composant champ agrégat persistant. Lorsque vous spécifiez un champ agrégat dans l'éditeur de champs, la propriété *Aggregates* de l'ensemble de données client est automatiquement mise à jour pour intégrer la spécification d'agrégat appropriée. La propriété *AggFields* contient le nouveau composant champ agrégat tandis que la méthode *FindField* le renvoie.

Copie de données d'un autre ensemble de données

Pour copier les données d'un autre ensemble de données lors de la conception, cliquez avec le bouton droit sur l'ensemble de données client et choisissez Affecter données locales. Une boîte de dialogue apparaît, affichant tous les ensembles de données disponibles dans votre projet. Sélectionnez les données et la structure que vous souhaitez copier et choisissez OK. Lorsque vous copiez l'ensemble de données source, votre ensemble de données client est automatiquement activé.

Pour copier à partir d'un autre ensemble de données à l'exécution, vous pouvez affecter ses données directement ou, si la source est un autre ensemble de données client, cloner le curseur.

Affectation directe des données

Vous pouvez utiliser la propriété *Data* de l'ensemble de données client pour affecter des données à un ensemble de données client depuis un autre ensemble de données. *Data* est un paquet de données qui se présente sous la forme d'un OleVariant. Un paquet de données peut émaner d'un autre ensemble de données client ou de tout autre ensemble de données avec l'aide d'un fournisseur. Une

fois qu'un paquet de données est affecté à *Data*, son contenu est automatiquement affiché dans les contrôles orientés données connectés à l'ensemble de données client par un composant source de données.

Lorsque vous ouvrez un ensemble de données client représentant des données du serveur ou utilisant un composant fournisseur externe, les paquets de données sont automatiquement affectés à *Data*.

Lorsque votre ensemble de données client n'utilise pas de fournisseur, vous pouvez copier les données à partir d'un autre ensemble de données client comme suit :

```
ClientDataSet1.Data := ClientDataSet2.Data;
```

Remarque Lorsque vous copiez la propriété *Data* d'un autre ensemble de données client, vous copiez également le journal de modifications, mais la copie ne reflète pas les filtres ni les portées ayant été appliqués. Pour inclure les filtres ou les portées, vous devez cloner le curseur de l'ensemble de données source.

Si vous copiez à partir d'un ensemble de données autre qu'un ensemble de données client, vous pouvez créer un composant fournisseur d'ensembles de données, le relier à l'ensemble de données source et copier ses données :

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SourceDataSet;
ClientDataSet1.Data := TempProvider.Data;
TempProvider.Free;
```

Remarque Lorsque vous affectez directement les données à la propriété *Data*, le nouveau paquet de données n'est pas fusionné dans les données existantes. Au lieu de cela, toutes les anciennes données sont remplacées.

Si vous souhaitez fusionner les modifications d'un autre ensemble de données, et non copier ses données, vous devez utiliser un composant fournisseur. Créez un fournisseur d'ensembles de données comme dans l'exemple précédent, mais attachez-le à l'ensemble de données de destination et au lieu de copier la propriété *Data*, utilisez la méthode *ApplyUpdates* :

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := ClientDataSet1;
TempProvider.ApplyUpdates(SourceDataSet.Delta, -1, ErrCount);
TempProvider.Free;
```

Clonage d'un curseur d'ensemble de données client

Les ensembles de données client utilisent la méthode *CloneCursor* qui vous permet de travailler avec une autre vue des données à l'exécution. *CloneCursor* permet à un deuxième ensemble de données client de partager les données de l'ensemble de données client original. Ceci est moins onéreux que de copier toutes les données originales mais, comme les données sont partagées, le second ensemble de données client ne peut pas modifier les données sans affecter l'ensemble de données client original.

CloneCursor prend trois paramètres : *Source* spécifie l'ensemble de données client à cloner. Les deux autres paramètres (*Reset* et *KeepSettings*) indiquent si d'autres informations que les données doivent être copiées. Il peut s'agir de tout filtre, de

l'index en cours, de liens vers une table maître (lorsque l'ensemble de données source est un ensemble détail), de la propriété *ReadOnly* et de tout lien vers un composant connexion ou un fournisseur.

Lorsque *Reset* et *KeepSettings* valent *False*, un ensemble de données client cloné est ouvert et les paramètres de l'ensemble de données client source sont utilisés pour définir les propriétés de l'ensemble de données destination. Lorsque *Reset* vaut *True*, les propriétés de l'ensemble de données client destination reçoivent les valeurs par défaut (aucun index ni filtre, aucune table maître, *ReadOnly* vaut *False* et aucun composant connexion ou fournisseur n'est spécifié). Lorsque *KeepSettings* vaut *True*, les propriétés de l'ensemble de données destination ne sont pas modifiées.

Ajout d'informations d'application aux données

Les développeurs d'applications peuvent ajouter des informations personnalisées à la propriété *Data* de l'ensemble de données client. Comme ces informations sont regroupées dans le paquet de données, elles sont incluses lorsque vous enregistrez les données dans un fichier ou un flux. Elles sont copiées lorsque vous copiez les données dans un autre ensemble de données. Elles peuvent aussi être intégrées dans la propriété *Delta* afin que le fournisseur puisse les lire lorsqu'il reçoit les mises à jour de l'ensemble de données client.

Pour enregistrer les informations d'application dans la propriété *Data*, utilisez la méthode *SetOptionalParam*. Cette méthode vous permet de stocker un OleVariant qui contient les données sous un nom spécifique.

Pour extraire ces informations d'application, utilisez la méthode *GetOptionalParam*, en transmettant le nom utilisé pour leur stockage.

Utilisation d'un ensemble de données client pour mettre en cache les mises à jour

Lorsque vous modifiez des données dans la majorité des ensembles de données, chaque fois que vous supprimez ou émettez un enregistrement, par défaut, l'ensemble de données génère une transaction, supprime ou écrit l'enregistrement sur le serveur de base de données, puis valide la transaction. S'il y a un problème lors de l'écriture des modifications dans la base de données, votre application reçoit immédiatement une notification : l'ensemble de données déclenche une exception lorsque vous émettez (post) l'enregistrement.

Si votre ensemble de données utilise un serveur de base de données distant, cette approche peut réduire les performances, à cause du trafic sur le réseau entre votre application et le serveur chaque fois que vous passez à un nouvel enregistrement après l'édition de l'enregistrement en cours. Pour réduire le trafic sur le réseau, vous pouvez effectuer les mises à jour dans un cache local. Lorsque vous mettez en cache les mises à jour, votre application récupère les données dans la base de données, les met en cache et les modifie en local, puis

applique les mises à jour du cache à la base de données en une seule transaction. Lorsque vous mettez en cache les mises à jour, les changements effectués sur un ensemble de données (comme émettre des modifications ou supprimer des enregistrements) sont stockés en local au lieu d'être écrits directement dans la table sous-jacente à l'ensemble de données. Lorsque les modifications sont terminées, votre application appelle une méthode qui écrit les modifications du cache dans la base de données et vide le cache.

Mettre en cache les mises à jour peut minimiser le temps de transaction et réduire le trafic sur le réseau. Cependant, les données mises en cache sont locales pour votre application et ne se trouvent plus sous le contrôle des transactions. Cela signifie que, pendant que vous travaillez sur votre copie des données dans votre mémoire locale, d'autres applications peuvent être en train de changer les données dans la table de la base de données sous-jacente. D'autre part, elles ne peuvent pas voir les modifications que vous effectuez tant que ne validez pas les mises à jour du cache. De ce fait, les mises à jour en cache ne conviennent pas aux applications impliquant des données trop versatiles, car vous risqueriez de créer ou de rencontrer trop de conflits au moment d'intégrer vos modifications dans la base de données.

Bien que le BDE et ADO fournissent d'autres mécanismes pour cacher les mises à jour, l'utilisation d'un ensemble de données client offre plusieurs avantages :

- Lorsque des ensembles de données sont liés par des relations maître/détail, l'application des mises à jour est gérée à votre place. Cela garantit que les mises à jour dans des ensembles de données multiples liés entre eux sont effectuées dans le bon ordre.
- Les ensembles de données client vous procurent le meilleur contrôle sur le processus de mise à jour. Vous pouvez définir des propriétés pour influencer le SQL généré pour mettre à jour les enregistrements, spécifier la table à utiliser dans le cas de mise à jour d'enregistrements à partir d'une jointure multitable, ou même appliquer les mises à jour manuellement à partir d'un gestionnaire d'événement *BeforeUpdateRecord*.
- Si des erreurs se produisent pendant l'application au serveur de la base de données des mises à jour du cache, seuls les ensembles de données client (et les fournisseurs d'ensembles de données) vous donnent des informations sur l'enregistrement en cours dans le serveur en plus de la valeur originale (non modifiée) de votre ensemble de données et de la nouvelle valeur (modifiée) de la mise à jour qui a échoué.
- Les ensembles de données client vous permettent de spécifier le nombre d'erreurs de mise à jour que vous pouvez accepter avant l'abandon de la totalité de la mise à jour.

Présentation de l'utilisation d'un cache pour les mises à jour

Pour utiliser les mises à jour en cache, les processus suivants doivent se produire successivement dans une application.

1 **Indiquer les données que vous voulez modifier.** La façon dont vous le ferez dépend du type d'ensemble de données client que vous utilisez :

   - Si vous utilisez *TClientDataSet*, spécifiez le composant fournisseur qui représente les données que vous voulez modifier. Cela est décrit dans "Spécification d'un fournisseur" à la page 23-30.
   - Si vous utilisez un ensemble de données client associé à un mécanisme d'accès aux données particulier, vous devez
     - Identifier le serveur de base de données en définissant la propriété *DBConnection* par un composant de connexion approprié.
     - Indiquer les données que vous voulez voir en spécifiant les propriétés *CommandText* et *CommandType*. *CommandType* indique si *CommandText* est une instruction SQL à exécuter, le nom d'une procédure stockée ou le nom d'une table. Si *CommandText* est une requête ou une procédure stockée, utilisez la propriété *Params* pour extraire les paramètres d'entrée.
     - Optionnellement, utilisez la propriété *Options* pour déterminer si les ensembles détails imbriqués et les données BLOB doivent figurer dans les paquets de données ou être extraits séparément, si les types spécifiques d'édition (insertions, modifications ou suppressions) sont désactivés, si une même mise à jour peut concerner plusieurs enregistrements du serveur et si les enregistrements de l'ensemble de données client sont rafraîchis lorsqu'il applique les mises à jour. *Options* est identique à la propriété *Options* du fournisseur. Par conséquent, vous pouvez définir des options non pertinentes ou inappropriées. Par exemple, il n'y a pas lieu d'inclure *poIncFieldProps*, car l'ensemble de données client n'extrait pas ses données d'un ensemble de données comportant des champs persistants. De même, il n'est pas souhaitable d'exclure *poAllowCommandText*, qui est inclus par défaut, car cela désactiverait la propriété *CommandText*, que l'ensemble de données client utilise pour spécifier quelles données il veut. Pour plus d'informations sur la propriété *Options* du fournisseur, voir "Initialisation des options contrôlant les paquets de données" à la page 24-6.

2 **Afficher et modifier les données,** permettre l'insertion de nouveaux enregistrements et la suppression d'enregistrements existants. A la fois la copie originale de chaque enregistrement et toutes ses modifications sont stockées en mémoire. Ce processus est décrit dans "Edition des données" à la page 23-6.

3 **Extraire des enregistrements supplémentaires si nécessaire.** Par défaut, les ensembles de données client extraient tous les enregistrements et les stockent en mémoire. Si un ensemble de données contient beaucoup d'enregistrements, ou des enregistrements avec des champs BLOB volumineux, vous souhaiterez probablement modifier ce comportement pour que l'ensemble de données clients n'extrait que les enregistrements requis pour l'affichage et recommence l'opération lorsque cela s'avère à nouveau nécessaire. Pour des détails sur la

façon de contrôler le processus d'extraction des enregistrements, voir "Extraction des données dans l'ensemble de données ou le document source" à la page 23-31.

4 **Optionnellement, rafraîchir les enregistrements.** Plus le temps passe, plus d'autres utilisateurs risquent de modifier les données sur le serveur de la base de données. Cela peut provoquer un écart de plus en plus important entre les données de l'ensemble client et celles du serveur, et une augmentation des risques d'erreurs au moment où vous appliquerez les mises à jour. Pour réduire ce problème, vous pouvez rafraîchir les enregistrements n'ayant pas été modifiés. Voir "Rafraîchissement des enregistrements" à la page 23-36, pour plus de détails.

5 **Appliquer les enregistrements du cache local à la base de données** ou annuler les mises à jour. Pour chaque enregistrement écrit dans la base de données, un événement *BeforeUpdateRecord* est déclenché. Si une erreur se produit lors de l'écriture d'un enregistrement dans la base de données, un événement *OnUpdateError* permet à l'application de corriger cette erreur, si possible, et de poursuivre la mise à jour. Lorsque les mises à jour sont terminées, toutes les mises à jour appliquées avec succès sont supprimées du cache local. Pour plus d'informations sur l'application des mises à jour dans la base de données, voir "Mise à jour des enregistrements" à la page 23-24.

   Au lieu d'appliquer les mises à jour, une application peut annuler les mises à jour, en vidant le journal de modifications sans les écrire dans la base de données. Vous pouvez annuler les mises à jour en appelant la méthode *CancelUpdates*. Tous les enregistrements supprimés dans le cache sont "dé-supprimés", les enregistrements modifiés sont ramenés à leurs valeurs originelles, et les enregistrements nouvellement insérés disparaissent tout simplement.

Choix du type d'ensemble de données pour les mises à jour en cache

Delphi inclut certains composants d'ensembles de données client spécialisés pour les mises à jour en cache. Chaque ensemble de données client est associé à un mécanisme d'accès aux données particulier. Ils sont décrits dans le tableau suivant :

Tableau 23.3 Ensembles de données client spécialisés pour les mises à jour en cache

Ensemble de données client	Mécanisme d'accès aux données
TBDEClientDataSet	Borland Database Engine
TSQLClientDataSet	dbExpress
TIBClientDataSet	InterBase Express

En outre, vous pouvez mettre en cache les mises à jour en utilisant l'ensemble de données client générique (*TClientDataSet*) avec un fournisseur externe et un ensemble de données source. Pour plus d'informations sur l'utilisation de

TClientDataSet avec un fournisseur externe, voir "Utilisation d'un ensemble de données client avec un fournisseur" à la page 23-29.

Remarque Les ensembles de données client spécialisés associés à chaque mécanisme d'accès aux données utilisent en fait un fournisseur et un ensemble de données source. Mais, et le fournisseur et l'ensemble de données source sont internes par rapport à l'ensemble de données client.

Le plus simple est d'utiliser un des ensembles de données client spécialisés pour mettre en cache les mises à jour. Mais, il est parfois préférable d'utiliser *TClientDataSet* avec un fournisseur externe :

- Si vous utilisez un mécanisme d'accès aux données n'ayant pas d'ensemble de données client spécialisé, vous devez utiliser *TClientDataSet* avec un composant fournisseur externe. Par exemple, si les données proviennent d'un document XML ou d'un ensemble de données personnalisé.
- Si vous travaillez avec des tables entre lesquelles a été établie une relation maître/détail, vous devez utiliser *TClientDataSet* et le connecter, au moyen d'un fournisseur, à la table maître de deux ensembles de données source liés dans une relation maître/détail. L'ensemble de données client voit l'ensemble détail comme le champ d'un ensemble de données imbriqué. Cette approche est nécessaire pour permettre l'application des mises à jour des tables maître et client dans l'ordre correct.
- Si vous voulez coder des gestionnaires d'événements répondant à la communication entre l'ensemble client et le fournisseur (par exemple, avant et après que l'ensemble client extrait des enregistrements du fournisseur), vous devez utiliser *TClientDataSet* avec un composant fournisseur externe. Les ensembles de données client spécialisés publient les événements les plus importants de l'application des mises à jour (*OnReconcileError*, *BeforeUpdateRecord* et *OnGetTableName*), mais ils ne publient pas les événements périphériques de la communication entre l'ensemble client et son fournisseur, car au départ ils ont été prévus pour les applications multiniveaux.
- Lorsque vous utilisez le BDE, vous pouvez vouloir utiliser un fournisseur externe et un ensemble de données source si vous avez besoin d'utiliser un objet mise à jour. Bien qu'il soit possible de coder un objet mise à jour à partir du gestionnaire d'événement *BeforeUpdateRecord* de *TBDEClientDataSet*, il est plus simple d'affecter la propriété *UpdateObject* de l'ensemble de données source. Pour plus d'informations sur l'utilisation des objets mise à jour, voir "Utilisation d'objets mise à jour pour mettre à jour un ensemble de données" à la page 20-45.

Indication des enregistrements modifiés

Lorsque l'utilisateur modifie un ensemble de données client, il vous semblera sans doute utile de fournir le feedback des modifications effectuées. Cela sera particulièrement utile si vous voulez autoriser l'utilisateur à défaire certaines

modifications en naviguant jusqu'à elles et en cliquant sur un bouton "Défaire" par exemple.

La méthode *UpdateStatus* et les propriétés *StatusFilter* sont utiles pour fournir un feedback sur les mises à jour qui se sont produites :

- *UpdateStatus* indique quel type de mise à jour s'est éventuellement produit sur l'enregistrement en cours. Cela peut être l'une des valeurs suivantes :
  - *usUnmodified* indique que l'enregistrement en cours est inchangé.
  - *usModified* indique que l'enregistrement en cours a été modifié.
  - *usInserted* indique qu'un enregistrement a été inséré par l'utilisateur.
  - *usDeleted* indique qu'un enregistrement a été supprimé par l'utilisateur.
- *StatusFilter* contrôle quels types de mises à jour sont visibles dans le journal de modifications. *StatusFilter* fonctionne sur les enregistrements en cache comme les filtres sur des données standard. *StatusFilter* est un ensemble qui peut inclure n'importe quelle combinaison des valeurs suivantes :
  - *usUnmodified* indique un enregistrement non modifié.
  - *usModified* indique un enregistrement modifié.
  - *usInserted* indique un enregistrement inséré.
  - *usDeleted* indique un enregistrement supprimé.

  Par défaut, *StatusFilter* est l'ensemble [*usModified*, *usInserted*, *usUnmodified*]. Vous pouvez ajouter *usDeleted* à cet ensemble afin de fournir le feedback des enregistrements supprimés en plus.

Remarque *UpdateStatus* et *StatusFilter* sont également utiles dans les gestionnaires des événements *BeforeUpdateRecord* et *OnReconcileError*. Pour plus d'informations sur *BeforeUpdateRecord*, voir "Intervention pendant l'application des mises à jour" à la page 23-25. Pour plus d'informations *OnReconcileError*, voir "Conciliation des erreurs de mise à jour" à la page 23-27.

L'exemple suivant montre comment fournir du feedback sur l'état de mise à jour des enregistrements, en utilisant la méthode *UpdateStatus*. Il suppose que vous avez changé la propriété *StatusFilter* afin qu'elle comprenne *usDeleted*, ce qui permet aux enregistrements supprimés de rester visibles dans l'ensemble de données. Il suppose de plus que vous avez ajouté un champ calculé à l'ensemble de données, appelé "Status".

```
procedure TForm1.ClientDataSet1CalcFields(DataSet: TDataSet);
begin
  with ClientDataSet1 do begin
    case UpdateStatus of
      usUnmodified: FieldByName('Status').AsString := '';
      usModified: FieldByName('Status').AsString := 'M';
      usInserted: FieldByName('Status').AsString := 'I';
      usDeleted: FieldByName('Status').AsString := 'D';
    end;
  end;
end;
```

Mise à jour des enregistrements

Le contenu du journal de modifications est stocké en tant que paquet de données dans la propriété *Delta* de l'ensemble de données client. Pour rendre permanents les changements stockés dans *Delta*, l'ensemble de données client doit les appliquer à la base de données (ou à l'ensemble de données source ou au document XML).

Lorsqu'un client applique les mises à jour au serveur, le processus est le suivant :

1. L'application client appelle la méthode *ApplyUpdates* d'un objet ensemble de données client. Cette méthode transmet le contenu de la propriété *Delta* de l'ensemble de données client au fournisseur (interne ou externe). *Delta* est un paquet de données qui contient les enregistrements mis à jour, insérés et supprimés dans un ensemble de données client.
2. Le fournisseur applique les mises à jour, en plaçant en mémoire cache tous les enregistrements problématiques qu'il ne peut pas résoudre lui-même. Voir "Comment répondre aux demandes de mise à jour des clients" à la page 24-9, pour plus de détails sur l'application des mises à jour par le fournisseur.
3. Le fournisseur renvoie tous les enregistrements non résolus à l'ensemble de données client dans un paquet de données *Result*. Le paquet de données *Result* contient tous les enregistrements non mis à jour. Il contient aussi les informations d'erreur, comme les messages d'erreur et les codes d'erreur.
4. L'ensemble de données client essaie de concilier les erreurs de mise à jour renvoyées dans le paquet de données *Result* enregistrement par enregistrement.

Application des mises à jour

Les changements apportés à la copie locale des données de l'ensemble client ne sont transmis au serveur de base de données (ou au document XML) que lorsque l'application client appelle la méthode*ApplyUpdates*. *ApplyUpdates* prend les modifications dans le journal de modifications et les envoie au fournisseur sous forme d'un paquet de données (nommé *Delta*). (Notez que, lors de l'utilisation de la majorité des ensembles de données client, le fournisseur est interne à l'ensemble de données client.)

ApplyUpdates accepte un paramètre unique, *MaxErrors*, qui indique le nombre maximum d'erreurs que le fournisseur peut tolérer avant de mettre fin au processus de mise à jour. Si *MaxErrors* est égal à *0*, tout le processus de mise à jour prend fin dès qu'une erreur de mise à jour se produit. Aucune modification n'est écrite dans la base de données et le journal de modifications de l'ensemble de données client reste inchangé. Si *MaxErrors* est égal à *-1*, un nombre quelconque d'erreurs est toléré et le journal de modifications contient tous les enregistrements n'ayant pas pu être appliqués. Si *MaxErrors* a une valeur positive et qu'il se produit davantage d'erreurs que le nombre autorisé par *MaxErrors*, toutes les mises à jour sont annulées. S'il se produit moins d'erreurs que le nombre spécifié par *MaxErrors*, tous les enregistrements qui s'appliquent

correctement sont automatiquement effacés du journal de modifications de l'ensemble de données client.

ApplyUpdates renvoie le nombre réel d'erreurs rencontrées, qui est toujours inférieur ou égal à *MaxErrors* plus un. Cette valeur de renvoi indique le nombre d'enregistrements qui n'ont pas pu être écrits dans la base de données.

La méthode *ApplyUpdates* de l'ensemble de données client effectue les opérations suivantes :

1. Elle appelle indirectement la méthode *ApplyUpdates* du fournisseur. La méthode *ApplyUpdates* du fournisseur écrit les mises à jour dans la base de données, dans l'ensemble de données source ou dans le document XML, et tente de corriger les erreurs rencontrées. Les enregistrements qu'elle ne peut appliquer à cause des erreurs sont renvoyés à l'ensemble de données client.
2. La méthode *ApplyUpdates* de l'ensemble de données client essaie alors de concilier ces enregistrements problématiques en appelant la méthode *Reconcile*. *Reconcile* est une routine de gestion d'erreur qui appelle le gestionnaire d'événement *OnReconcileError*. Vous devez écrire le code du gestionnaire d'événement *OnReconcileError* pour qu'il corrige les erreurs. Pour plus d'informations sur l'utilisation de *OnReconcileError*, voir "Conciliation des erreurs de mise à jour" à la page 23-27.
3. Enfin, *Reconcile* supprime du journal de modifications celles dont l'application a réussi et met à jour *Data* avec les nouveaux enregistrements mis à jour. Quand *Reconcile* est terminée, *ApplyUpdates* indique le nombre d'erreurs survenues.

Important Dans certains cas, le fournisseur ne peut déterminer comment appliquer les mises à jour (par exemple, lors de mises à jour à partir d'une procédure stockée ou d'une jointure multitable). Les ensembles de données client et les composants fournisseur génèrent des événements vous permettant de gérer de telles situations. Voir "Intervention pendant l'application des mises à jour" ci-après, pour plus de détails.

Conseil Si le fournisseur se trouve sur un serveur d'applications sans état, vous voudrez peut-être échanger avec lui des informations d'état persistantes, avant ou après l'application des mises à jour. *TClientDataSet* reçoit un événement *BeforeApplyUpdates* avant que les mises à jour ne soient envoyées, ce qui vous permet d'envoyer au serveur des informations d'état persistantes. Une fois que les mises à jour sont appliquées (mais avant le processus de conciliation), *TClientDataSet* reçoit un événement *AfterApplyUpdates*, ce qui vous permet de répondre à toute information d'état persistante renvoyée par le serveur d'applications.

Intervention pendant l'application des mises à jour

Lorsqu'un ensemble de données client applique ses mises à jour, le fournisseur détermine comment gérer l'écriture des insertions, des suppressions et des modifications sur le serveur de la base de données ou dans l'ensemble de données source. Lorsque vous utilisez *TClientDataSet* avec un composant fournisseur externe, vous pouvez utiliser les propriétés et les événements de ce

fournisseur pour agir sur la façon dont sont appliquées les mises à jour. Ces méthodes sont décrites dans "Comment répondre aux demandes de mise à jour des clients" à la page 24-9.

Mais, lorsque le fournisseur est interne, ce qui se passe pour tout ensemble de données client associé à un mécanisme d'accès aux données, vous ne pouvez pas définir ses propriétés ni fournir de gestionnaire d'événement. Il s'en suit que l'ensemble de données client publie une propriété et deux événements vous permettant d'agir sur la façon dont le fournisseur interne applique les mises à jour.

- *UpdateMode* détermine les champs utilisés pour rechercher des enregistrements dans les instructions SQL que génère le fournisseur pour appliquer les mises à jour. *UpdateMode* est identique à la propriété *UpdateMode* du fournisseur. Pour plus d'informations sur la propriété *UpdateMode* du fournisseur, voir "Comment contrôler l'application des mises à jour" à la page 24-11.
- *OnGetTableName* qui vous permet de fournir au fournisseur le nom de la table de base de données à laquelle il doit appliquer les mises à jour. Cela permet au fournisseur de générer les instructions SQL de mise à jour lorsqu'il ne peut pas identifier la table de la base de données à partir de la procédure stockée ou de la requête spécifiée par *CommandText*. Par exemple, si la requête exécute une jointure multitable qui n'implique des mises à jour que pour une seule table, la fourniture d'un gestionnaire d'événement *OnGetTableName* permet au fournisseur interne d'appliquer correctement les mises à jour.

  Le gestionnaire d'événement *OnGetTableName* reçoit trois paramètres : le composant fournisseur interne, l'ensemble de données interne ayant extrait les données du serveur, et un paramètre pour renvoyer le nom de la table à utiliser dans le SQL généré.
- *BeforeUpdateRecord* se produit pour chaque enregistrement du paquet delta. Cet événement permet d'effectuer tout changement de dernière minute avant que l'enregistrement ne soit inséré, supprimé ou modifié. En outre, il vous permet d'exécuter vos propres instructions SQL pour appliquer la mise à jour lorsque le fournisseur n'est pas en mesure de générer une instruction SQL correcte (c'est par exemple le cas des jointures multitables dans lesquelles plusieurs tables doivent être mises à jour.)

  Le gestionnaire d'événement *BeforeUpdateRecord* reçoit cinq paramètres : le composant fournisseur interne, l'ensemble de données interne qui a extrait les données du serveur, un paquet delta positionné sur l'enregistrement qui va être mis à jour, une indication de la nature de la mise à jour (insertion, suppression ou modification), et un paramètre qui renvoie l'indication que le gestionnaire d'événement a effectué la mise à jour. Tout cela est illustré dans le gestionnaire d'événement suivant. Pour simplifier, cet exemple suppose que les instructions SQL sont accessibles en tant que variables globales ne nécessitant que les valeurs des champs :

```
procedure TForm1.SQLClientDataSet1BeforeUpdateRecord(Sender: TObject;
   SourceDS: TDataSet; DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind;
   var Applied Boolean);
var
```

```
  SQL: string;
  Connection: TSQLConnection;
begin
  Connection := (SourceDS as TCustomSQLDataSet).SQLConnection;
  case UpdateKind of
  ukModify:
    begin
     { 1st dataset: update Fields[1], use Fields[0] in where clause }
      SQL := Format(UpdateStmt1, [DeltaDS.Fields[1].NewValue, DeltaDS.Fields[0].OldValue]);
      Connection.Execute(SQL, nil, nil);
     { 2nd dataset: update Fields[2], use Fields[3] in where clause }
      SQL := Format(UpdateStmt2, [DeltaDS.Fields[2].NewValue, DeltaDS.Fields[3].OldValue]);
      Connection.Execute(SQL, nil, nil);
    end;
  ukDelete:
    begin
     { 1st dataset: use Fields[0] in where clause }
      SQL := Format(DeleteStmt1, [DeltaDS.Fields[0].OldValue]);
      Connection.Execute(SQL, nil, nil);
     { 2nd dataset: use Fields[3] in where clause }
      SQL := Format(DeleteStmt2, [DeltaDS.Fields[3].OldValue]);
      Connection.Execute(SQL, nil, nil);
    end;
  ukInsert:
    begin
     { 1st dataset: values in Fields[0] and Fields[1] }
       SQL := Format(InsertStmt1, [DeltaDS.Fields[0].NewValue,
DeltaDS.Fields[1].NewValue]);
      Connection.Execute(SQL, nil, nil);
     { 2nd dataset: values in Fields[2] and Fields[3] }
      SQL := Format(InsertStmt2, [DeltaDS.Fields[2].NewValue, DeltaDS.Fields[3].NewValue]);
      Connection.Execute(SQL, nil, nil);
    end;
  end;
  Applied := True;
end;
```

Conciliation des erreurs de mise à jour

Deux événements vous permettent de gérer les erreurs qui se produisent pendant le processus de mise à jour :

- Pendant le processus de mise à jour, le fournisseur interne génère un événement *OnUpdateError* chaque fois qu'il rencontre une mise à jour qu'il ne peut pas traiter. Si vous corrigez le problème dans le gestionnaire de l'événement *OnUpdateError*, l'erreur n'est pas comptabilisée dans le nombre maximal d'erreurs passé à la méthode *ApplyUpdates*. Cet événement ne se produit que pour les ensembles de données client utilisant un fournisseur interne. Si vous utilisez *TClientDataSet*, vous pouvez utiliser à la place l'événement *OnUpdateError* du composant fournisseur.
- Lorsque l'opération de mise à jour est complètement terminée, l'ensemble de données client génère un événement *OnReconcileError* pour chaque

enregistrement que le fournisseur n'a pas pu appliquer sur le serveur de la base de données.

Il faut toujours coder un gestionnaire d'événement *OnReconcileError* ou *OnUpdateError*, même s'il ne fait qu'abandonner les enregistrements n'ayant pas pu être appliqués. Les gestionnaires des deux événements fonctionnent de manière identique. Ils incluent les paramètres suivants :

- *DataSet :* Un ensemble de données client qui contient l'enregistrement mis à jour qui n'a pas pu être appliqué. Vous pouvez utiliser les méthodes de l'ensemble de données client pour obtenir des informations sur l'enregistrement problématique et pour le modifier afin de résoudre tous les problèmes. Vous pouvez notamment utiliser les propriétés *CurValue*, *OldValue* et *NewValue* des champs de l'enregistrement en cours pour déterminer la cause du problème de mise à jour. Toutefois, vous ne pouvez pas appeler une méthode de l'ensemble de données client qui modifie l'enregistrement en cours dans votre gestionnaire d'événement.
- *E :* Un objet qui représente le problème qui s'est produit. Vous pouvez utiliser cette exception pour extraire un message d'erreur ou pour déterminer la cause d'une erreur de mise à jour.
- *UpdateKind :* Le type de mise à jour ayant généré l'erreur. *UpdateKind* peut être *ukModify* (problème survenu lors de la mise à jour d'un enregistrement existant modifié), *ukInsert* (problème survenu lors de l'insertion d'un nouvel enregistrement) ou *ukDelete* (problème survenu lors de la suppression d'un enregistrement existant).
- *Action :* Un paramètre **var** qui indique l'action à entreprendre à la fin de l'exécution du gestionnaire d'événement. Dans votre gestionnaire d'événement, vous initialisez ce paramètre pour
  - Ignorer l'enregistrement et le laisser dans le journal de modifications. (rrSkip ou raSkip)
  - Abandonner l'opération de conciliation entière. (rrAbort ou raAbort)
  - Fusionner la modification infructueuse avec l'enregistrement correspondant sur le serveur. (rrMerge ou raMerge) Cela ne fonctionne que si l'enregistrement du serveur n'inclut pas de modification sur les champs modifiés dans l'enregistrement de l'ensemble de données client.
  - Remplacer la mise à jour courante dans le journal de modifications par la valeur de l'enregistrement dans le gestionnaire d'événement, qui a normalement été corrigé. (rrApply ou raCorrect)
  - Ignorer l'erreur complètement. (rrIgnore) Cette possibilité n'existe que dans le gestionnaire d'événement *OnUpdateError*, elle est prévue pour le cas où le gestionnaire d'événement applique la mise à jour dans le serveur de la base de données. L'enregistrement mis à jour est supprimé du journal de modifications et fusionné dans *Data*, comme si le fournisseur avait appliqué la mise à jour.

- Retirer les modifications de l'enregistrement sur l'ensemble de données client, en réappliquant les valeurs initialement fournies. (raCancel) Cette possibilité n'existe que dans le gestionnaire d'événement *OnReconcileError*.
- Mettre à jour la valeur de l'enregistrement en cours en fonction de l'enregistrement du serveur. (raRefresh) Cette possibilité n'existe que dans le gestionnaire d'événement *OnReconcileError*.

Le code suivant montre un gestionnaire d'événement *OnReconcileError* qui utilise la boîte de dialogue d'erreur et de conciliation de l'unité RecError présente dans le répertoire objrepos. Pour utiliser cette boîte de dialogue, ajoutez RecError à votre clause uses.

```
procedure TForm1.ClientDataSetReconcileError(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind, var Action TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

Utilisation d'un ensemble de données client avec un fournisseur

Un ensemble de données client utilise un fournisseur pour se fournir en données et applique des mises à jour lorsque

- II met en cache les mises à jour d'un serveur de base de données ou d'un autre ensemble de données.
- Il représente les données d'un document XML.
- Il stocke les données de la partie client d'une application multiniveau.

Pour tout ensemble de données client autre que *TClientDataSet*, ce fournisseur est interne et, de cette façon, non directement accessible à l'application. Avec *TClientDataSet*, le fournisseur est un composant externe qui lie l'ensemble de données client à une source de données externe.

Un fournisseur externe peut résider dans la même application que l'ensemble de données client ou faire partie d'une application séparée exécutée sur un autre système. Pour plus d'informations sur les composants fournisseur, voir chapitre 24, "Utilisation des composants fournisseur". Pour plus d'informations sur les applications dans lesquelles le fournisseur réside sur une application séparée exécutée sur un système différent, voir chapitre 25, "Création d'applications multiniveaux".

Lorsque vous utilisez un fournisseur (interne ou externe), l'ensemble de données client met toujours en cache toutes les mises à jour. Pour plus d'informations, voir "Utilisation d'un ensemble de données client pour mettre en cache les mises à jour" à la page 23-18.

Les rubriques suivantes décrivent les propriétés et méthodes supplémentaires de l'ensemble de données client lui permettant de fonctionner avec un fournisseur.

Spécification d'un fournisseur

Contrairement aux ensembles de données client associés à un mécanisme d'accès aux données, *TClientDataSet* n'a pas de composant fournisseur interne pour empaqueter les données ou appliquer des mises à jour. Si vous voulez qu'il représente des données issues d'un ensemble de données source ou d'un document XML, vous devez associer l'ensemble client avec un composant fournisseur externe.

La façon dont vous associez *TClientDataSet* à un fournisseur dépend de l'endroit où se trouve le fournisseur : dans la même application que l'ensemble de données client ou sur un serveur d'applications distant exécuté sur un autre système.

- Si le fournisseur se trouve dans la même application que l'ensemble de données client, vous pouvez associer l'ensemble de données à un fournisseur en choisissant celui-ci dans la liste déroulante de la propriété *ProviderName* dans l'inspecteur d'objets. Ce procédé fonctionne sous réserve que le fournisseur possède le même *Owner* que l'ensemble de données client. L'ensemble de données client et le fournisseur possèdent le même *Owner* s'ils sont placés dans la même fiche ou dans le même module de données. Pour utiliser un fournisseur local possédant un *Owner* différent, vous devez établir l'association à l'exécution à l'aide de la méthode *SetProvider* de l'ensemble de données client.

  Si vous envisagez de passer à un fournisseur distant ou si vous voulez faire directement appel à l'interface *IAppServer*, vous pouvez définir la propriété *RemoteServer* par un composant *TLocalConnection*. Si vous utilisez *TLocalConnection*, l'instance *TLocalConnection* gère la liste des tous les fournisseurs locaux par rapport à votre application, et les appels *IAppServer* de l'ensemble de données client. Si vous n'utilisez pas *TLocalConnection*, Delphi crée un objet caché pour gérer les appels *IAppServer* de l'ensemble de données client.

- Si le fournisseur se trouve sur un serveur d'applications distant, outre la propriété *ProviderName*, vous devez spécifier un composant qui connecte l'ensemble de données client au serveur d'applications. Voici deux propriétés pouvant gérer cette tâche : *RemoteServer*, qui spécifie le nom d'un composant connexion à partir duquel obtenir une liste de fournisseurs, ou *ConnectionBroker*, qui spécifie un courtier ou agent centralisé fournissant un niveau d'indirection supplémentaire entre l'ensemble client et le composant connexion. Le composant connexion et le composant courtier, lorsqu'il est utilisé, se trouvent dans le même module de données que l'ensemble de données client. Le composant connexion établit et maintient la connexion au serveur d'applications, c'est pourquoi il est parfois appelé "le courtier ou l'agent des données". Pour plus d'informations, voir "Structure de l'application client" à la page 25-5.

  Après avoir spécifié en mode conception la propriété *RemoteServer* ou *ConnectionBroker*, vous pouvez sélectionner un fournisseur dans la liste déroulante de la propriété *ProviderName* dans l'inspecteur d'objets. Cette liste

contient les fournisseurs locaux (appartenant à la même fiche ou au même module de données) et les fournisseurs distants accessibles par le biais du composant connexion.

Remarque Si le composant connexion est une instance de *TDCOMConnection*, le serveur d'applications doit être recensé sur la machine client.

A l'exécution, vous pouvez passer d'un fournisseur à l'autre (local ou distant) en modifiant la propriété *ProviderName* dans le code.

Extraction des données dans l'ensemble de données ou le document source

Les ensembles de données client peuvent contrôler comment ils vont extraire les paquets de données d'un fournisseur. Par défaut, un ensemble de données client extrait tous les enregistrements de l'ensemble de données source. Cela est vrai si l'ensemble source et le fournisseur sont des composants internes (comme avec *TBDEClientDataSet, TSQLClientDataSet* et *TIBClientDataSet*), ou des composants séparés qui fournissent les données au *TClientDataSet*.

Vous pouvez changer la façon dont l'ensemble de données client lit les enregistrements, en utilisant les propriétés *PacketRecords* et *FetchOnDemand*.

Extractions incrémentales

En changeant la propriété *PacketRecords*, vous pouvez demander que l'ensemble de données client lise les données en fragments plus petits. *PacketRecords* spécifie la quantité d'enregistrements à extraire à la fois et le type des enregistrements à renvoyer. Par défaut, *PacketRecords* vaut *-1*, ce qui signifie que tous les enregistrements disponibles sont extraits à la fois, que ce soit à la première ouverture de l'ensemble de données client ou lorsque l'application appelle explicitement *GetNextPacket*. Lorsque *PacketRecords* vaut *-1*, l'ensemble de données client n'extrait pas de données supplémentaires après la première extraction des données, car il dispose de tous les enregistrements disponibles.

Pour extraire les enregistrements par petits lots, affectez à *PacketRecords* une valeur correspondant au nombre d'enregistrements voulu. Par exemple, l'instruction suivante fixe la taille de chaque paquet de données à dix enregistrements :

```
ClientDataSet1.PacketRecords := 10;
```

Ce processus d'extraction d'enregistrements par petits lots est appelé "extraction incrémentale". Les ensembles de données client utilisent l'extraction incrémentale lorsque *PacketRecords* est supérieur à zéro.

Pour lire chaque lot d'enregistrements, l'ensemble de données client appelle *GetNextPacket*. Les paquets extraits sont ajoutés à la fin des données se trouvant déjà dans l'ensemble de données client. *GetNextPacket* renvoie le nombre d'enregistrements extraits. Si la valeur renvoyée est équivalente à la valeur de *PacketRecords*, c'est que tous les enregistrements disponibles n'ont pas été traités. Si la valeur renvoyée est supérieure à *0* mais inférieure à *PacketRecords*, c'est que

le dernier enregistrement a été atteint durant l'opération d'extraction. Si *GetNextPacket* renvoie *0*, c'est qu'il n'y a plus aucun enregistrement à extraire.

Attention L'extraction incrémentale ne fonctionne pas si les données sont extraites d'un fournisseur distant situé sur un serveur d'applications sans état. Voir "Gestion des informations d'état dans les modules de données distants" à la page 25-23, pour plus d'informations sur l'utilisation de l'extraction incrémentale avec les modules de données distants sans état.

Remarque La propriété *PacketRecords* peut aussi être utilisée pour extraire des informations métadonnées sur l'ensemble de données source. Pour extraire des informations métadonnées, *PacketRecords* doit valoir *0*.

Extraction à la demande

L'extraction automatique d'enregistrements est contrôlée par la propriété *FetchOnDemand*. Lorsque *FetchOnDemand* vaut *True* (la valeur par défaut), l'ensemble de données client lit automatiquement les enregistrements en fonction des besoins. Pour empêcher l'extraction automatique des enregistrements, mettez *FetchOnDemand* à *False*. Si *FetchOnDemand* est à *False*, l'application doit appeler explicitement *GetNextPacket* pour extraire des enregistrements.

Par exemple, les applications qui doivent représenter des ensembles de données très volumineux accessibles en lecture seulement peuvent désactiver *FetchOnDemand* afin que les ensembles de données client n'essaient pas de charger plus de données que la mémoire ne peut en contenir. Entre les extractions, l'ensemble de données client libère sa mémoire cache à l'aide de la méthode *EmptyDataSet*. Cette approche, toutefois, ne fonctionne pas très bien lorsque le client doit renvoyer les mises à jour au serveur.

Le fournisseur contrôle si les enregistrements figurant dans les paquets de données comprennent des données BLOB et des ensembles de données détail imbriqués. Si le fournisseur exclut cette information des enregistrements, la propriété *FetchOnDemand* oblige l'ensemble de données client à extraire automatiquement les données BLOB et les ensembles de données détail au fur et à mesure des besoins. Si *FetchOnDemand* vaut *False,* et si le fournisseur ne contient pas de données BLOB ni d'ensembles de données détail avec les enregistrements, vous devez appeler explicitement la méthode *FetchBlobs* ou *FetchDetails* pour extraire ces informations.

Obtention de paramètres de l'ensemble de données source

Il existe deux cas dans lesquels un ensemble de données client doit obtenir des valeurs de paramètres :

- L'application a besoin de la valeur des paramètres de sortie d'une procédure stockée.
- Le client souhaite initialiser les paramètres d'entrée d'une requête ou d'une procédure stockée d'après les valeurs en cours dans l'ensemble de données source.

Les ensembles de données client stockent les valeurs des paramètres dans leur propriété *Params*. Ces valeurs sont rafraîchies par des paramètres de sortie quelconques chaque fois que l'ensemble de données client lit les données depuis l'ensemble de données source. Il peut cependant exister des cas où un composant *TClientDataSet* dans une application client a besoin de paramètres de sortie alors qu'il ne lit pas de données.

Pour lire des paramètres de sortie sans lire d'enregistrement, ou pour initialiser des paramètres d'entrée, l'ensemble de données client peut extraire des valeurs de paramètre de l'ensemble de données source en appelant la méthode *FetchParams*. Les paramètres sont renvoyés dans un paquet de données depuis le fournisseur et affectés à la propriété *Params* de l'ensemble de données client.

A la conception, la propriété *Params* peut être initialisée en cliquant avec le bouton droit sur l'ensemble de données client et en choisissant Récupérer les paramètres.

Remarque Il n'est jamais nécessaire d'appeler *FetchParams* lorsque l'ensemble de données client utilise un fournisseur interne et un ensemble de données source, car la propriété *Params* reflète toujours les paramètres de l'ensemble de données source interne. Dans le cas de *TClientDataSet*, la méthode *FetchParams* (ou la commande Récupérer les paramètres) ne fonctionne que si l'ensemble de données client est connecté à un fournisseur dont l'ensemble de données associé peut fournir des paramètres. Par exemple, si l'ensemble de données source est un ensemble de données de type table, il n'y a aucun paramètre à récupérer.

Si le fournisseur réside sur un système séparé dans le cadre d'un serveur d'applications sans état, vous ne pouvez pas utiliser *FetchParams* pour extraire des paramètres de sortie. Dans un serveur d'applications sans état, d'autres clients peuvent modifier et ré-exécuter la requête ou la procédure stockée, en changeant les paramètres de sortie avant l'appel de *FetchParams*. Pour extraire des paramètres de sortie depuis un serveur d'applications sans état, utilisez la méthode *Execute*. Si le fournisseur est associé à une requête ou à une procédure stockée, *Execute* indique au fournisseur d'exécuter la requête ou la procédure stockée et de renvoyer les éventuels paramètres de sortie. Ces paramètres renvoyés sont alors utilisés pour mettre automatiquement à jour la propriété *Params*.

Transmission de paramètres à l'ensemble de données source

Les ensembles de données client peuvent transmettre des paramètres à l'ensemble de données source pour spécifier les données qui doivent figurer dans les paquets de données qu'il envoie. Ces paramètres peuvent spécifier :

- Les valeurs des paramètres en entrée d'une requête ou d'une procédure stockée exécutée sur le serveur d'applications
- Les valeurs des champs qui limitent le nombre d'enregistrements envoyés dans les paquets de données

Vous pouvez spécifier les valeurs de paramètres que votre ensemble de données client envoie au fournisseur lors de la conception ou à l'exécution. Lors de la conception, sélectionnez l'ensemble de données client et double-cliquez sur la

propriété *Params* dans l'inspecteur d'objets. Cela appelle l'éditeur de collection, dans lequel vous pouvez ajouter, supprimer ou réorganiser les paramètres. Lorsque vous sélectionnez un paramètre dans l'éditeur de collection, vous pouvez en modifier les propriétés à l'aide de l'inspecteur d'objets.

A l'exécution, utilisez la méthode *CreateParam* de la propriété *Params* pour ajouter des paramètres à votre ensemble de données client. *CreateParam* renvoie un objet paramètre, doté d'un nom, d'un type de paramètre et d'un type de données particuliers. Vous pouvez alors utiliser les propriétés de cet objet paramètre pour affecter une valeur au paramètre.

Par exemple, le code suivant attribue la valeur 605 à un paramètre appelé CustNo :

```
with ClientDataSet1.Params.CreateParam(ftInteger, 'CustNo', ptInput) do
    AsInteger := 605;
```

Si l'ensemble de données client n'est pas actif, vous pouvez envoyer les paramètres au serveur d'applications et récupérer un paquet de données qui reflète ces valeurs de paramètre en mettant simplement la propriété *Active* à *True*.

Envoi de paramètres de requête ou de procédure stockée

Lorsque la propriété *CommandType* de l'ensemble de donnés client est *ctQuery* ou *ctStoredProc*, ou lorsque le fournisseur associé représente le résultat d'une requête ou d'une procédure stockée, si l'ensemble de données client est une instance de *TClientDataSet*, vous pouvez utiliser la propriété *Params* pour spécifier la valeur des paramètres. Quand l'ensemble client demande des données à l'ensemble source ou utilise sa méthode *Execute* pour exécuter une requête ou une procédure stockée ne renvoyant pas d'ensemble de données, il transmet les valeurs de ces paramètres en même temps que la demande des données ou que la commande execute. Lorsque le fournisseur reçoit ces paramètres, il les affecte à l'ensemble de données qui lui est associé. Il indique alors à l'ensemble de données d'exécuter la requête ou la procédure stockée en utilisant les valeurs de ces paramètres et, si l'ensemble de données client a demandé des données, il commence à fournir celles-ci, en commençant par le premier enregistrement de l'ensemble de résultats.

Remarque Les noms de paramètres doivent correspondre aux noms des paramètres correspondants dans l'ensemble de données source.

Limitation des enregistrements avec des paramètres

Si l'ensemble de données client est

- une instance de *TClientDataSet* dont le fournisseur associé représente un composant *TTable* ou *TSQLTable*
- une instance de *TSQLClientDataSet* or *TBDEClientDataSet* dont la propriété *CommandType* est *ctTable*

il peut alors utiliser la propriété *Params* pour limiter les enregistrements mis en mémoire cache. Chaque paramètre représente la valeur d'un champ, cette valeur

doit se trouver dans un enregistrement pour que celui-ci soit inclus dans les données de l'ensemble client. Cela fonctionne comme un filtre, sauf qu'avec un filtre, les enregistrements sont encore en mémoire cache mais considérés comme non disponibles.

Le nom de chaque paramètre doit correspondre à un nom de champ. Lorsque vous utilisez *TClientDataSet*, ce sont les noms des champs du composant *TTable* ou *TSQLTable* associé au fournisseur. Lorsque vous utilisez *TSQLClientDataSet* ou *TBDEClientDataSet*, ce sont les noms des champs de la table sur le serveur de base de données. L'ensemble client ne contient alors que les enregistrements dont les valeurs dans les champs équivalents correspondent à celles des paramètres.

Par exemple, supposons une application qui affiche les commandes d'un seul client. Lorsque l'utilisateur identifie le client, l'ensemble de données client définit sa propriété *Params* pour inclure un paramètre unique nommé CustID, dont la valeur identifie le client dont les commandes doivent être affichées. Quand l'ensemble de données client demande les données de l'ensemble de données source, il transmet la valeur de ce paramètre. Le fournisseur n'envoie ensuite que les enregistrements concernant le client identifié. Ceci est beaucoup plus efficace que la solution dans laquelle tous les enregistrements de commande sont envoyés par le fournisseur à l'application client puis filtrés à l'aide de l'ensemble de données client.

Gestion des contraintes liées au serveur

Lorsqu'un serveur de base de données définit des contraintes sur la validité des données, il est utile que l'ensemble de données client les connaisse. Ainsi, l'ensemble de données client peut garantir que les modifications apportées par l'utilisateur n'enfreignent pas les contraintes établies par le serveur. Il s'en suit que de telles violations ne sont jamais transmises au serveur de la base de données où elles auraient été rejetées. Cela signifie que les mises à jour générant des erreurs lors du processus de mise à jour sont moins nombreuses.

Indépendamment de la source des données, vous pouvez dupliquer les contraintes du serveur en les ajoutant explicitement à l'ensemble de données client. Ce processus est décrit dans "Spécification de contraintes personnalisées" à la page 23-8.

C'est cependant plus pratique si les contraintes du serveur sont automatiquement incluses dans les paquets de données. Car vous n'avez pas à spécifier explicitement les expressions ni les contraintes par défaut, et l'ensemble de données client change les valeurs qu'il rend obligatoires lorsque les contraintes du serveur changent. Par défaut, voici ce qu'il se produit exactement : si l'ensemble de données source est averti des contraintes du serveur, le fournisseur les inclut automatiquement dans les paquets de données et l'ensemble de données client les impose lorsque l'utilisateur poste ses modifications dans le journal de modifications.

Remarque Seuls les ensembles de données qui utilisent le BDE peuvent importer des contraintes à partir du serveur. Cela signifie que les contraintes du serveur ne sont incluses dans les paquets de données que si vous utilisez *TBDEClientDataSet*

ou *TClientDataSet* avec un fournisseur qui représente une base de données basée sur le BDE. Pour plus d'informations sur la façon d'importer les contraintes du serveur et d'empêcher un fournisseur de les inclure dans les paquets de données, voir "Gestion des contraintes du serveur" à la page 24-15.

Remarque Pour plus d'informations sur l'utilisation des contraintes lorsqu'elles ont été importées, voir "Utilisation des contraintes du serveur" à la page 19-25.

Bien que l'importation des contraintes et des expressions du serveur soit une fonctionnalité de grand intérêt, qui aide une application à préserver l'intégrité des données, il y a des circonstances où apparaît la nécessité de les désactiver de manière provisoire. Par exemple, si une contrainte du serveur est basée sur la valeur maximale en cours dans un champ et si l'ensemble de données client utilise la lecture incrémentale, la valeur maximale en cours dans l'ensemble de données client peut être différente de la valeur maximale sur le serveur de la base de données et les contraintes appelées différemment. Dans un autre cas, si un ensemble de données client applique un filtre aux enregistrements quand des contraintes sont activées, ce filtre peut provoquer des interférences indésirables avec les conditions des contraintes. Dans chacun de ces cas, une application peut désactiver le contrôle des contraintes.

Pour désactiver temporairement les contraintes, appelez la méthode *DisableConstraints*. A chaque appel de *DisableConstraints*, un compteur de références est incrémenté. Tant que la valeur de ce compteur est supérieure à zéro, les contraintes ne sont pas appliquées sur l'ensemble de données client.

Pour réactiver les contraintes pour l'ensemble de données client, appelez la méthode *EnableConstraints* de l'ensemble de données. Chaque appel à *EnableConstraints* décrémente le compteur de références. Quand ce compteur atteint la valeur zéro, les contraintes sont à nouveau activées.

Conseil Appelez toujours *DisableConstraints* et *EnableConstraints* dans des blocs appariés pour garantir que les contraintes sont activées quand vous souhaitez qu'elles le soient.

Rafraîchissement des enregistrements

Les ensembles de données client manipulent une photographie mémorisée des données de l'ensemble de données source. Si l'ensemble source représente un serveur de base de données, au fur et à mesure que le temps passe, d'autres utilisateurs peuvent modifier ces données. Les données de l'ensemble de données client deviennent une représentation de moins en moins fidèle des données sous-jacentes.

Comme tout ensemble de données, les ensembles de données client disposent d'une méthode *Refresh* qui met à jour ses enregistrements en fonction des valeurs courantes sur le serveur. Toutefois, l'appel de *Refresh* ne fonctionne que si le journal de modifications ne contient aucun changement. L'appel de *Refresh* alors que des modifications n'ont pas été appliquées déclenche une exception.

Les ensembles de données client peuvent aussi mettre à jour les données sans toucher au journal de modifications. Pour ce faire, appelez la méthode

RefreshRecord. A la différence de la méthode *Refresh*, *RefreshRecord* ne met à jour que l'enregistrement en cours dans l'ensemble de données client. *RefreshRecord* modifie la valeur d'enregistrement initialement obtenue du fournisseur mais laisse intacts tous les changements contenus dans le journal de modifications.

Attention Il n'est pas approprié d'appeler systématiquement *RefreshRecord*. Si les modifications de l'utilisateur entrent en conflit avec celles apportées à l'ensemble de données sous-jacent par d'autres utilisateurs, l'appel de *RefreshRecord* masque ce conflit. Lorsque l'ensemble de données client applique ses mises à jour, aucune erreur de conciliation ne se produit et l'application ne peut pas résoudre le conflit.

Pour éviter que les erreurs de mise à jour ne soient masquées, vous pouvez vérifier qu'aucune mise à jour n'est en attente avant d'appeler *RefreshRecord*. Par exemple, ci-dessous, *AfterScroll* rafraîchit l'enregistrement en cours chaque fois que l'utilisateur se déplace sur un nouvel enregistrement (ce qui garantit la valeur la plus récente), mais seulement lorsqu'il n'est pas dangereux de procéder ainsi :

```
procedure TForm1.ClientDataSet1AfterScroll(DataSet: TDataSet);
begin
  if ClientDataSet1.UpdateStatus = usUnModified then
    ClientDataSet1.RefreshRecord;
end;
```

Communication avec des fournisseurs à l'aide d'événements personnalisés

Les ensembles de données client communiquent avec un composant fournisseur par le biais d'une interface spéciale appelée *IAppServer*. Si le fournisseur est local, *IAppServer* est l'interface vers un objet auto-généré qui gère toute la communication entre l'ensemble de données client et son fournisseur. Si le fournisseur est distant, *IAppServer* est l'interface vers un module de données distant sur le serveur d'applications.

TClientDataSet offre de nombreuses possibilités de personnalisation de la communication qui utilise l'interface *IAppServer*. Avant et après chaque appel de méthode *IAppServer* dirigé vers le fournisseur de l'ensemble de données client, *TClientDataSet* reçoit des événements spéciaux qui lui permettent d'échanger des informations quelconques avec son fournisseur. Ces événements correspondent à des événements similaires du fournisseur. Et, par exemple, quand l'ensemble de données client appelle sa méthode *ApplyUpdates*, les événements suivants se produisent :

1. L'ensemble de données client reçoit un événement *BeforeApplyUpdates*, où il spécifie des informations personnalisées dans un OleVariant nommé *OwnerData*.
2. Le fournisseur reçoit un événement *BeforeApplyUpdates*, où il peut répondre à l'*OwnerData* depuis l'ensemble de données client et mettre à jour la valeur de l'*OwnerData* avec les nouvelles informations.

3. Le fournisseur poursuit son processus normal d'assemblage d'un paquet de données (comprenant tous les événements qui l'accompagnent).
4. Le fournisseur reçoit un événement *AfterApplyUpdates*, où il peut répondre à la valeur courante de l'*OwnerData* et la mettre à jour en lui donnant une valeur pour l'ensemble de données client.
5. L'ensemble de données client reçoit un événement *AfterApplyUpdates*, où il peut répondre à la valeur renvoyée de l'*OwnerData*.

Les appels aux autres méthodes d'*IAppServer* sont accompagnés d'un ensemble semblable d'événements *BeforeXXX* et *AfterXXX* qui vous permettent de personnaliser la communication entre l'ensemble de données client et son fournisseur.

En outre, l'ensemble de données client dispose d'une méthode spéciale, *DataRequest*, dont le seul but est de permettre une communication avec le fournisseur spécifique à l'application. Quand l'ensemble de données client appelle *DataRequest*, il transmet un OleVariant en tant que paramètre pouvant contenir les informations que vous voulez. En retour, est généré sur le fournisseur l'événement *OnDataRequest*, où vous pouvez répondre, de la façon que vous avez définie dans l'application, et renvoyer une valeur à l'ensemble de données client.

Redéfinition de l'ensemble de données source

Les ensembles de données client associés à un mécanisme particulier d'accès aux données utilisent les propriétés *CommandText* et *CommandType* pour spécifier les données qu'ils représentent. Cependant lorsque vous utilisez *TClientDataSet*, les données sont spécifiées par l'ensemble de données source, et non par l'ensemble de données client. Habituellement, cet ensemble de données source possède une propriété qui spécifie l'instruction SQL générant les données, le nom d'une table dans la base de données ou le nom d'une procédure stockée.

Si le fournisseur le permet, *TClientDataSet* peut redéfinir sur l'ensemble de données source la propriété qui indique les données qu'il représente. C'est-à-dire que, si le fournisseur le permet, la propriété *CommandText* de l'ensemble de données client remplace la propriété sur l'ensemble de données du fournisseur qui spécifie les données qu'il représente. Cela permet à l'ensemble de données client de spécifier de façon dynamique les données qu'il veut voir.

Par défaut, les composants fournisseur externe ne permettent pas aux ensembles de données client d'utiliser la valeur *CommandText* de cette façon. Pour permettre à *TClientDataSet* d'utiliser sa propriété *CommandText*, vous devez ajouter *poAllowCommandText* à la propriété *Options* du fournisseur. Sinon, la valeur de *CommandText* est ignorée.

Remarque Ne supprimez jamais *poAllowCommandText* de la propriété *Options* de *TSQLClientDataSet, de TBDEClientDataSet* ou de *TIBClientDataSet*. La propriété *Options* de l'ensemble de données client est adressée au fournisseur interne, aussi la suppression de *poAllowCommandText* empêche l'ensemble de données client d'indiquer à quelles données accéder.

L'ensemble de données client envoie sa chaîne *CommandText* au fournisseur à deux moments :

- Quand l'ensemble de données client est ouvert pour la première fois. Une fois que le premier paquet de données est extrait du fournisseur, l'ensemble de données client n'envoie plus *CommandText* lorsqu'il lit les paquets de données suivants.
- Quand l'ensemble de données client envoie une commande *Execute* au fournisseur.

Pour envoyer une commande SQL ou pour changer le nom d'une table ou d'une procédure stockée à un autre moment, vous devez utiliser explicitement l'interface *IAppServer* par le biais de la propriété *AppServer*. Cette propriété représente l'interface par laquelle l'ensemble de données client communique avec son fournisseur.

Utilisation d'un ensemble de données client avec des données basées sur des fichiers

Les ensembles de données client peuvent fonctionner avec des fichiers sur disque dédiés tout comme il le font avec les données issues d'un serveur. Cela leur permet d'être utilisés dans les applications de bases de données de type fichier et les applications modèle "briefcase". Les fichiers spéciaux que les ensembles de données client utilisent pour leurs données sont appelés MyBase.

Conseil Tous les ensembles de données client sont appropriés à une application du modèle "briefcase", mais pour une application MyBase pure (une application qui n'utilise pas de fournisseur), il est préférable d'utiliser *TClientDataSet*, car il implique une charge système minimale.

Dans une application MyBase pure, l'application client ne peut pas obtenir de définitions de table ni de données du serveur et aucun serveur ne peut recevoir ses mises à jour. A la place, l'ensemble de données doit indépendamment :

- Définir et créer les tables
- Charger les données enregistrées
- Fusionner les modifications dans ses données
- Enregistrer les données

Création d'un nouvel ensemble de données

Il existe trois façons de définir et créer des ensembles de données client qui ne représentent pas les données d'un serveur :

- Vous pouvez définir et créer un nouvel ensemble de données client en utilisant des champs persistants ou des définitions de champs et d'index. Suivez le même processus que pour la création d'un ensemble de données de type table. Pour davantage d'informations, voir "Création et suppression des tables" à la page 18-44.

- Vous pouvez copier un ensemble de données existant (à la conception ou à l'exécution). Pour plus d'informations sur la copie des ensembles de données existants, voir "Copie de données d'un autre ensemble de données" à la page 23-16.
- Vous pouvez créer un ensemble de données client à partir de n'importe quel document XML. Voir "Conversion de documents XML en paquets de données" à la page 26-7, pour plus de détails.

Une fois l'ensemble de donnée créé, vous pouvez l'enregistrer dans un fichier. A partir de ce moment-là, vous n'avez plus besoin de recréer la table ; il vous suffit de la charger depuis le fichier que vous avez enregistré. Lorsque vous commencez à élaborer une application de base de données de type fichier, vous pouvez créer et enregistrer des fichiers vides pour vos ensembles de données avant d'écrire l'application elle-même. Ainsi, vous démarrez en ayant les métadonnées de votre ensemble de données client déjà définies, ce qui facilite la définition de l'interface utilisateur.

Chargement des données depuis un fichier ou un flux

Pour charger des données depuis un fichier, appelez la méthode *LoadFromFile* de l'ensemble de données client. *LoadFromFile* accepte un paramètre, une chaîne qui spécifie le fichier où lire les données. Le cas échéant, le nom de fichier peut être un nom de chemin d'accès qualifié. Si vous chargez toujours les données de l'ensemble de données client à partir du même fichier, vous pouvez utiliser à la place la propriété *FileName*. Si *FileName* nomme un fichier existant, les données sont automatiquement chargées à l'ouverture de l'ensemble de données client.

Pour charger des données à partir d'un flux, appelez la méthode *LoadFromStream* de l'ensemble de données client. *LoadFromStream* accepte un paramètre : un objet flux qui fournit les données.

Les données chargées par *LoadFromFile* (*LoadFromStream)* doivent avoir été sauvegardées dans le format ensemble de données client, par cet ensemble de données client ou par un autre, en utilisant la méthode *SaveToFile (SaveToStream),* ou générées à partir d'un document XML. Pour plus d'informations sur la sauvegarde des données dans un fichier ou dans un flux, voir "Sauvegarde des données dans un fichier ou un flux" à la page 23-41. Pour plus d'informations sur la création des données d'un ensemble de données client à partir d'un document XML, voir chapitre 26, "Utilisation de XML dans les applications de bases de données".

Lorsque vous appelez *LoadFromFile* ou *LoadFromStream*, toutes les données du fichier sont lues dans la propriété *Data*. Toutes les modifications qui figuraient dans le journal de modifications lorsque les données ont été sauvegardées sont lues dans la propriété *Delta*. Mais, les seuls index lus dans le fichier sont ceux qui ont été créés avec l'ensemble de données.

Fusion des modifications dans les données

Lorsque vous modifiez les données d'un ensemble client, toutes les modifications sont consignées dans un journal de modifications qui n'existe qu'en mémoire. Ce journal est géré séparément des données elles-mêmes bien que cela soit complètement transparent pour les objets qui utilisent l'ensemble de données client. En d'autres termes, les contrôles qui permettent la navigation dans l'ensemble de données client ou qui affichent ses données ont une vue des données qui intègre les modifications. Si vous ne souhaitez pas annuler les modifications, vous devez fusionner le journal de modifications avec les données de l'ensemble de données client en appelant la méthode *MergeChangeLog*. *MergeChangeLog* écrase les enregistrements de *Data* avec les valeurs des champs du journal de modifications.

Après l'exécution de *MergeChangeLog*, la propriété *Data* contient un amalgame constitué des données existantes et des modifications issues du journal de modifications. Cet amalgame devient la nouvelle valeur de la propriété *Data* (elle sert ensuite de référence aux futures modifications). *MergeChangeLog* efface tous les enregistrements du journal de modifications et réinitialise la propriété *ChangeCount* à *0*.

Attention N'appelez pas *MergeChangeLog* pour les ensembles de données client qui utilisent un fournisseur. Dans ce cas, vous devez appeler la méthode *ApplyUpdates* pour écrire les modifications dans la base de données. Pour plus d'informations, voir "Application des mises à jour" à la page 23-24.

Remarque Il est également possible de fusionner les modifications dans les données d'un ensemble de données client séparé si ce dernier a initialement fourni les données dans la propriété *Data*. Pour ce faire, vous devez utiliser un fournisseur d'ensembles de données. Voir un exemple dans "Affectation directe des données" à la page 23-16.

Si vous ne souhaitez pas utiliser les capacités Défaire fournies par le journal de modifications, vous pouvez définir la propriété *LogChanges* de l'ensemble de données client par *False*. Lorsque *LogChanges* vaut *False*, les modifications sont automatiquement fusionnées lorsque vous postez des enregistrements et il est inutile d'appeler *MergeChangeLog*.

Sauvegarde des données dans un fichier ou un flux

Lorsque vous avez fusionné les modifications avec les données d'un ensemble de données client, ces dernières n'existent qu'en mémoire. Bien qu'elles ne soient pas perdues si vous fermez puis ouvrez à nouveau l'ensemble de données client dans votre application, l'arrêt de celle-ci les fera disparaître. Pour que ces données soient permanentes, elles doivent être écrites sur disque. Enregistrez les modifications sur disque à l'aide de la méthode *SaveToFile*.

SaveToFile accepte un paramètre : une chaîne qui spécifie le fichier dans lequel les données sont écrites. Le cas échéant, le nom de fichier peut être un nom de chemin d'accès qualifié. Si le fichier existe déjà, son contenu actuel est écrasé.

Remarque *SaveToFile* ne conserve aucun index ajouté à l'ensemble de données client à l'exécution, sauf les index ajoutés lorsque vous avez créé l'ensemble de données client.

Si vous sauvegardez toujours les données dans le même fichier, vous pouvez utiliser à la place la propriété *FileName.* Si *FileName* est initialisée, les données sont automatiquement sauvegardées dans le fichier nommé à la fermeture de l'ensemble de données client.

Vous pouvez aussi sauvegarder les données dans un flux à l'aide de la méthode *SaveToStream*. *SaveToStream* accepte un paramètre : un objet flux qui reçoit les données.

Remarque Si vous sauvegardez un ensemble de données client alors que des modifications existent encore dans le journal de modifications, ces modifications ne sont pas fusionnées avec les données. Lorsque vous rechargez les données à l'aide de la méthode *LoadFromFile* ou *LoadFromStream*, le journal de modifications contient toujours les modifications non fusionnées. Ce point est important pour les applications qui supportent le modèle "briefcase", dans lequel les modifications finissent par être appliquées à un composant fournisseur sur le serveur d'applications.

Chapitre

24

Utilisation des composants fournisseur

Les composants fournisseur (*TDataSetProvider* et *TXMLTransformProvider*) définissent le mécanisme qu'utilisent le plus couramment les ensembles de données client pour obtenir leurs données. Les fournisseurs

- reçoivent les demandes de données d'un ensemble de données client (ou agent XML), extraient les données demandées, empaquètent les données en un paquet de données transportable et renvoient les données à l'ensemble de données client (ou agent XML). Cette activité est appelée "fourniture".
- reçoivent les données mises à jour d'un ensemble de données client (ou agent XML), appliquent les mises à jour au serveur de base de données, à l'ensemble de données source ou au document XML, et consignent toutes les mises à jour inapplicables en renvoyant les mises à jour non résolues à l'ensemble de données client en vue d'une régularisation ultérieure. Cette activité est appelée "résolution".

L'essentiel de l'activité d'un composant fournisseur s'effectue automatiquement. Vous n'avez pas besoin d'écrire de code dans le fournisseur pour créer les paquets de données à partir des données d'un ensemble de données ou d'un document XML ou pour appliquer les mises à jour. Toutefois, les composants fournisseur comprennent une série d'événements et de propriétés qui permettent à votre application de contrôler plus directement les informations empaquetées pour les clients et la façon dont elle répond aux requêtes client.

Lorsque vous utilisez *TBDEClientDataSet*, *TSQLClientDataSet* ou *TIBClientDataSet*, le fournisseur est interne à l'ensemble de données client, et l'application ne peut pas y accéder directement. Lorsque vous utilisez *TClientDataSet* ou *TXMLBroker*, toutefois, le fournisseur est un composant séparé qui vous permet de contrôler les informations empaquetées pour les clients et les réponses aux événements qui se produisent au cours des processus de fourniture et de résolution. Les ensembles de données client qui possèdent un fournisseur interne mettent à

disposition une partie des propriétés et événements du fournisseur interne comme leurs propres propriétés et événements mais, pour un contrôle maximal, vous pouvez utiliser *TClientDataSet* avec un composant fournisseur séparé.

Lorsque vous utilisez un composant fournisseur séparé, il peut résider dans la même application que l'ensemble de données client (ou agent XML) ou sur un serveur d'applications dans le cadre d'une application multiniveau.

Ce chapitre décrit comment utiliser un composant fournisseur pour contrôler les interactions avec les ensembles de données client ou les agents XML.

Spécification de la source de données

Quand vous utilisez un composant fournisseur, vous devez spécifier la source qu'il utilise pour extraire les données que le composant assemble en paquets de données. Suivant votre version de Delphi, vous pouvez spécifier la source ainsi :

- Pour fournir les données à partir d'un ensemble de données, utilisez *TDataSetProvider*.
- Pour fournir les données à partir d'un document XML, utilisez *TXMLTransformProvider*.

Utilisation d'un ensemble de données comme source des données

Si le fournisseur est un fournisseur d'ensemble de données (*TDataSetProvider*), attribuez à sa propriété *DataSet* l'ensemble de données source. A la conception, sélectionnez-le parmi les ensembles de données disponibles dans la liste déroulante de la propriété *DataSet* dans l'inspecteur d'objets.

TDataSetProvider interagit avec l'ensemble de données source à l'aide de l'interface *IProviderSupport*. Cette interface est introduite par *TDataSet*, elle est donc disponible dans tous les ensembles de données. Cependant, les méthodes *IProviderSupport* implémentées dans *TDataSet* sont essentiellement des squelettes qui ne font rien ou déclenchent des exceptions.

Les classes d'ensemble de données fournies avec Delphi (ensembles de données BDE, ADO, *dbExpress* et InterBase Express) redéfinissent ces méthodes pour implémenter l'interface *IProviderSupport* d'une manière plus utile. Les ensembles de données client n'ajoutent rien à l'implémentation *IProviderSupport* héritée mais peuvent être néanmoins utilisés comme ensemble de données source sous réserve que la propriété *ResolveToDataSet* du fournisseur ait pour valeur *True*.

Les concepteurs de composants qui créent leur propre descendant personnalisé de *TDataSet* doivent redéfinir les méthodes appropriées de *IProviderSupport* si leurs ensembles de données doivent fournir des données à un fournisseur. Si le fournisseur fournit uniquement des paquets de données en mode lecture seule, (c'est-à-dire s'il n'applique pas les modifications), les méthodes *IProviderSupport* implémentées par *TDataSet* peuvent suffire.

Utilisation d'un document XML comme source des données

Si le fournisseur est un fournisseur XML, attribuez à sa propriété *XMLDataFile* le document source.

Etant donné que les fournisseurs XML doivent transformer le document source en paquets de données, vous devez, en plus d'indiquer le document source, spécifier comment celui-ci doit être transformé. Cette transformation est gérée par la propriété *TransformRead* du fournisseur. *TransformRead* représente un objet *TXMLTransform*. Vous pouvez définir ses propriétés pour spécifier la transformation à utiliser et recourir à ses événements pour fournir votre propre entrée à la transformation. Pour plus d'informations sur l'utilisation des fournisseurs XML, voir "Utilisation d'un document XML comme source pour un fournisseur" à la page 26-9.

Communication avec l'ensemble de données client

Toute la communication entre un fournisseur et un ensemble de données client ou agent XML s'effectue par le biais d'une interface *IAppServer*. Si le fournisseur figure dans la même application que le client, cette interface est implémentée par un objet caché automatiquement généré ou par un composant *TLocalConnection*. Si le fournisseur fait partie d'une application multiniveau, il s'agit de l'interface du module de données distant du serveur d'application.

La plupart des applications n'utilisent pas *IAppServer* directement, mais l'invoquent par le biais des propriétés et des méthodes de l'ensemble de données client ou de l'agent XML. Toutefois, lorsque cela est nécessaire, vous pouvez appeler directement l'interface *IAppServer* à l'aide de la propriété *AppServer* d'un ensemble de données client.

Le tableau suivant présente les méthodes de l'interface *IAppServer*, ainsi que les méthodes et les événements correspondants du composant fournisseur et de l'ensemble de données client. Ces méthodes *IAppServer* comprennent un paramètre *Provider*. Dans les applications multiniveaux, ce paramètre indique le fournisseur sur le serveur d'applications avec lequel l'ensemble de données client communique. La plupart des méthodes comprennent également un paramètre OleVariant appelé *OwnerData* qui permet à un ensemble de données client et à un fournisseur d'échanger des informations personnalisées. *OwnerData* n'est pas utilisé par défaut, mais est transmis à tous les gestionnaires d'événement, de sorte que vous pouvez écrire du code qui permet à votre fournisseur de s'adapter aux informations définies par l'application avant et après chaque appel émanant d'un ensemble de données client.

Tableau 24.1 Membres de l'interface AppServer

IAppServer	composant fournisseur	TClientDataSet
AS_ApplyUpdates (méthode)	ApplyUpdates (méthode), BeforeApplyUpdates (événement), AfterApplyUpdates (événement)	ApplyUpdates (méthode), BeforeApplyUpdates (événement), AfterApplyUpdates (événement).
AS_DataRequest (méthode)	DataRequest (méthode), OnDataRequest (événement)	DataRequest (méthode).
AS_Execute (méthode)	Execute (méthode), BeforeExecute (événement), AfterExecute (événement)	Execute (méthode), BeforeExecute (événement), AfterExecute (événement).
AS_GetParams (méthode)	GetParams (méthode), BeforeGetParams (événement), AfterGetParams (événement)	FetchParams (méthode), BeforeGetParams (événement), AfterGetParams (événement).
AS_GetProviderNames (méthode)	Utilisé pour identifier tous les fournisseurs disponibles.	Utilisé pour créer une liste à la conception pour la propriété ProviderName.
AS_GetRecords (méthode)	GetRecords (méthode), BeforeGetRecords (événement), AfterGetRecords (événement)	GetNextPacket (méthode), Data (propriété), BeforeGetRecords (événement), AfterGetRecords (événement)
AS_RowRequest (méthode)	RowRequest (méthode), BeforeRowRequest (événement), AfterRowRequest (événement)	FetchBlobs (méthode), FetchDetails (méthode), RefreshRecord (méthode), BeforeRowRequest (événement), AfterRowRequest (événement)

Détermination du mode d'application des mises à jour à l'aide d'un fournisseur d'ensemble de données

TXMLTransformProvider appliquent toujours les mises à jour au document XML associé. Toutefois, lorsque vous utilisez *TDataSetProvider*, vous pouvez choisir le mode d'application des mises à jour. Par défaut, quand les composants *TDataSetProvider* appliquent les modifications et résolvent les erreurs de modification, ils communiquent directement avec le serveur de bases de données en utilisant des instructions SQL générées dynamiquement. Cette approche présente cet avantage que l'application serveur n'a pas besoin de fusionner les modifications deux fois (d'abord pour l'ensemble de données, puis sur le serveur distant).

Cependant, vous pouvez préférer une autre approche : par exemple, vous pouvez utiliser certains des événements sur le composant ensemble de données. Il se peut aussi que l'ensemble de données utilisé ne gère pas l'utilisation d'instructions SQL (par exemple, si la fourniture est effectuée à partir d'un composant *TClientDataSet*).

TDataSetProvider vous laisse décider si vous souhaitez appliquer les modifications sur le serveur de bases de données en utilisant le SQL ou sur l'ensemble de

données source en définissant la propriété *ResolveToDataSet*. Quand cette propriété a la valeur *True*, les modifications sont appliquées à l'ensemble de données. Lorsqu'elle a pour valeur *False*, les mises à jour sont directement appliquées au serveur de bases de données sous-jacent.

Contrôle des informations placées dans les paquets de données

Lorsque vous utilisez un fournisseur d'ensemble de données, il y a plusieurs moyens de contrôler quelles informations sont placées dans les paquets de données envoyés et reçus par le client. C'est-à-dire :

- Spécifier les champs apparaissant dans les paquets de données
- Spécifier des options caractérisant les paquets de données
- Ajouter des informations personnalisées aux paquets de données

Remarque Ces techniques de contrôle du contenu des paquets de données sont uniquement disponibles pour les fournisseurs d'ensemble de données. Lorsque vous utilisez *TXMLTransformProvider*, vous pouvez uniquement gérer le contenu des paquets de données en contrôlant le fichier de transformation utilisé par le fournisseur.

Spécification des champs apparaissant dans les paquets de données

Lorsque vous utilisez un fournisseur d'ensemble de données, vous pouvez contrôler les champs apparaissant dans les paquets de données en créant des champs persistants dans l'ensemble de données qu'utilise le fournisseur pour construire les paquets de données. Le fournisseur peut alors inclure uniquement ces champs. Les champs dont la valeur est générée dynamiquement par l'ensemble de données source (comme les champs calculés ou les champs de référence) peuvent être utilisés, mais ils apparaissent aux ensembles de données client comme des champs statiques en lecture seule. Pour plus d'informations sur les champs persistants, voir "Champs persistants" à la page 19-3.

Si l'ensemble de données client modifie les données et applique les modifications, vous devez inclure suffisamment de champs pour qu'il n'y ait pas d'enregistrements doublon dans le paquet de données. Sinon, lors de l'application des modifications, il est impossible de déterminer les enregistrements à actualiser. Si vous ne voulez pas que l'ensemble de données client accède à des champs uniquement spécifiés pour assurer l'unicité, initialisez la propriété *ProviderFlags* de ces champs en spécifiant *pfHidden*.

Remarque La nécessité de spécifier suffisamment de champs pour éviter des enregistrements en doublon se pose également lorsque l'ensemble de données source du fournisseur représente une requête. Vous devez définir la requête de telle sorte qu'elle comprenne suffisamment de champs pour assurer l'unicité de tous les enregistrements, même si votre application n'utilise pas tous les champs.

Initialisation des options contrôlant les paquets de données

La propriété *Options* d'un fournisseur d'ensemble de données permet de spécifier si les champs BLOB ou les tables détail imbriquées sont également envoyés, si les propriétés d'affichage des champs sont inclues, le type de modifications autorisées, etc. Le tableau suivant énumère les valeurs possibles pouvant être placées dans *Options*.

Tableau 24.2 Options d'un fournisseur

Valeur	Signification
poAutoRefresh	Le fournisseur rafraîchit l'ensemble de données client avec les valeurs en cours pour les enregistrements à chaque fois qu'il applique les modifications.
poReadOnly	L'ensemble de données client ne peut pas appliquer de mises à jour au fournisseur.
poDisableEdits	Les ensembles de données client ne peuvent modifier les valeurs de données existantes. Si l'utilisateur tente de modifier un champ, l'ensemble de données client déclenche une exception. Cela n'affecte pas la capacité de l'ensemble de données client à insérer ou supprimer des enregistrements.
poDisableInserts	Les ensembles de données client ne peuvent pas insérer de nouveaux enregistrements. Si l'utilisateur tente d'insérer un enregistrement, l'ensemble de données client déclenche une exception. Cela n'affecte pas la capacité de l'ensemble de données client à supprimer des enregistrements ou à modifier des données existantes.
poDisableDeletes	Les ensembles de données client ne peuvent pas supprimer d'enregistrements. Si l'utilisateur tente de supprimer un enregistrement, l'ensemble de données client déclenche une exception. Cela n'affecte pas la capacité de l'ensemble de données client à insérer ou modifier des enregistrements.
poFetchBlobsOnDemand	Les valeurs des champs BLOB ne sont pas incluses dans les paquets de données. Les ensembles de données client doivent demander ces valeurs explicitement quand elles en ont besoin. Si la propriété *FetchOnDemand* de l'ensemble de données client a la valeur *True*, il demande ces valeurs automatiquement. Sinon, l'application doit appeler la méthode *FetchBlobs* de l'ensemble de données client pour lire les données BLOB.
poFetchDetailsOnDemand	Quand l'ensemble de données du fournisseur représente le maître dans une relation maître/détail, les valeurs de détail imbriquées ne sont pas incluses dans les paquet de données. Les ensembles de données client doivent demander ces valeurs explicitement. Si la propriété *FetchOnDemand* de l'ensemble de données client a la valeur *True*, il demande ces valeurs automatiquement. Sinon, l'application doit appeler la méthode *FetchDetails* de l'ensemble de données client pour lire les détails imbriqués.

Tableau 24.2 Options d'un fournisseur (suite)

Valeur	Signification
poIncFieldProps	Le paquet de données contient les propriétés de champs suivantes (quand elles sont pertinentes) : *Alignment*, *DisplayLabel*, *DisplayWidth*, *Visible*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, *Currency*, *EditMask* et *DisplayValues.*
poCascadeDeletes	Quand l'ensemble de données du fournisseur représente le maître dans une relation maître-détail, les enregistrements du détail sont automatiquement supprimés par le serveur quand des enregistrements maître sont supprimés. Pour utiliser cette option, le serveur de bases de données doit être configuré pour effectuer la suppression en cascade pour la relation d'intégrité référentielle.
poCascadeUpdates	Quand l'ensemble de données du fournisseur représente le maître dans une relation maître-détail, les valeurs de clé des tables détail sont actualisées automatiquement quand les valeurs correspondantes sont modifiées dans les enregistrements maître. Pour utiliser cette option, le serveur de bases de données doit être configuré pour effectuer les mises à jour en cascade pour la relation d'intégrité référentielle.
poAllowMultiRecordUpdates	Une seule actualisation peut entraîner la modification de plusieurs enregistrements de la table de base de données sous-jacente. Cela peut être fait par des déclencheurs, l'intégrité référentielle, des instructions SQL sur l'ensemble de données source, etc. Attention, s'il se produit une erreur, les gestionnaires d'événements donnent accès à l'enregistrement actualisé et pas aux autres enregistrements modifiés en conséquence.
poNoReset	Les ensembles de données client ne peuvent pas demander au fournisseur de repositionner le curseur sur le premier enregistrement avant de fournir des données.
poPropogateChanges	Les modifications apportées par le serveur aux enregistrements modifiés dans le cadre du processus de mise à jour sont renvoyées au client et fusionnées dans l'ensemble de données client.
poAllowCommandText	Le client peut redéfinir le texte de l'instruction SQL de l'ensemble de données associé ou le nom de la table ou procédure stockée qu'il représente.
poRetainServerOrder	L'ensemble de données client ne doit pas retrier les enregistrements de l'ensemble de données pour imposer un ordre par défaut.

Ajout d'informations personnalisées aux paquets de données

Les fournisseurs d'ensembles de données peuvent ajouter dans des paquets de données des informations définies par l'application en utilisant l'événement *OnGetDataSetProperties*. Ces informations sont codées sous la forme d'un OleVariant et stockées sous le nom spécifié. Les ensembles de données client peuvent alors lire ces informations en utilisant leur méthode *GetOptionalParam*. Vous pouvez également spécifier que les informations doivent être placées dans

les paquets delta envoyés par les ensembles de données client lors de la mise à jour des enregistrements. Dans ce cas, l'ensemble de données client peut ne jamais exploiter ces informations que le fournisseur s'envoie à lui-même.

Quand vous ajoutez des informations personnalisées dans l'événement *OnGetDataSetProperties*, chaque attribut individuel (appelé parfois un "paramètre optionnel") est spécifié en utilisant un tableau Variant contenant trois éléments : le nom (une chaîne), la valeur (un Variant) et un indicateur booléen indiquant si les informations doivent être placées dans les paquets delta lorsque le client applique les mises à jour. Ajoutez plusieurs attributs en créant un tableau Variant de tableaux Variant. Par exemple, le gestionnaire d'événement *OnGetDataSetProperties* suivant envoie deux valeurs : l'heure à laquelle les données ont été fournies et le nombre total d'enregistrements dans l'ensemble de données source. Seule l'heure à laquelle les données ont été fournies est renvoyée quand les ensembles de données client appliquent les mises à jour :

```
procedure TMyDataModule1.Provider1GetDataSetProperties(Sender: TObject; DataSet: TDataSet;
out Properties: OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] := VarArrayOf(['TimeProvided', Now, True]);
  Properties[1] := VarArrayOf(['TableSize', DataSet.RecordCount, False]);
end;
```

Quand l'ensemble de données client applique les mises à jour, l'heure à laquelle les enregistrements ont été initialement fournis peut être lue dans l'événement *OnUpdateData* du fournisseur :

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
var
  WhenProvided: TDateTime;
begin
  WhenProvided := DataSet.GetOptionalParam('TimeProvided');
  ...
end;
```

Comment répondre aux demandes de données des clients

Généralement, les demandes de données de la part des clients sont gérées automatiquement. Un ensemble de données client ou agent XML demande un paquet de données en appelant (indirectement via l'interface *IAppServer*) la méthode *GetRecords*. Le fournisseur répond automatiquement en lisant les données de l'ensemble de données ou du document XML associé, en créant un paquet de données qui est envoyé au client.

Le fournisseur a la possibilité de modifier les données après les avoir assemblées dans un paquet, mais avant d'envoyer le paquet au client. Par exemple, vous pouvez retirer des enregistrements du paquet en vous basant sur un critère (par exemple, le niveau d'accès d'un utilisateur) ou, dans une application multiniveau, crypter des données confidentielles avant de les envoyer au client.

Pour modifier le paquet de données avant de l'envoyer au client, écrivez un gestionnaire de l'événement *OnGetData*. Les gestionnaires d'événement *OnGetData* transmettent le paquet de données comme paramètre sous la forme d'un ensemble de données client. En utilisant les méthodes de l'ensemble de données client, vous pouvez modifier les données avant de les expédier au client.

Comme tous les appels de méthode effectués via l'interface *IAppServer*, le fournisseur peut communiquer des informations d'état persistantes à un ensemble de données client avant et après l'appel de *GetRecords*. Cette communication s'effectue en utilisant les gestionnaires d'événements *BeforeGetRecords* et *AfterGetRecords*. Pour davantage d'informations sur la persistance d'informations d'état dans les serveurs d'applications, voir "Gestion des informations d'état dans les modules de données distants" à la page 25-23.

Comment répondre aux demandes de mise à jour des clients

Un fournisseur applique les mises à jour aux enregistrements de la base de données en se basant sur un paquet de données *Delta* envoyé par un ensemble de données client ou agent XML. L'ensemble de données client demande des mises à jour en appelant (indirectement, via l'interface *IAppServer*) la méthode *ApplyUpdates*.

Comme tous les appels de méthode effectués via l'interface *IAppServer*, le fournisseur peut communiquer des informations d'état persistantes à un ensemble de données client avant et après l'appel de *ApplyUpdates*. Cette communication s'effectue en utilisant les gestionnaires d'événements *BeforeApplyUpdates* et *AfterApplyUpdates*. Pour davantage d'informations sur la persistance d'informations d'état dans les serveurs d'applications, voir "Gestion des informations d'état dans les modules de données distants" à la page 25-23.

Si vous utilisez un fournisseur d'ensemble de données, des événements supplémentaires vous apportent un contrôle accru :

Quand un fournisseur d'ensemble de données reçoit une demande de mise à jour, il génère un événement *OnUpdateData* dans lequel vous pouvez modifier le paquet Delta avant qu'il ne soit écrit dans l'ensemble de données ou déterminer comment les mises à jour sont appliquées. Après l'événement *OnUpdateData*, le fournisseur écrit les modifications dans la base de données ou l'ensemble de données source.

Le fournisseur effectue la mise à jour enregistrement par enregistrement. Avant d'appliquer chaque enregistrement, le fournisseur d'ensemble de données génère un événement *BeforeUpdateRecord* que vous pouvez utiliser pour filtrer avant que les modifications ne soient appliquées. Si une erreur se produit lors de la résolution d'un enregistrement, le fournisseur reçoit un événement *OnUpdateError* dans lequel il peut résoudre l'erreur. Généralement, il se produit des erreurs car la modification viole une contrainte du serveur ou parce qu'un enregistrement a été modifié avec une autre application après sa lecture par le fournisseur, mais avant la demande de l'ensemble de données client d'appliquer la mise à jour.

Les erreurs de mise à jour peuvent être traitées par le fournisseur d'ensemble de données ou l'ensemble de données client. Lorsque le fournisseur fait partie d'une application multiniveau, il doit gérer toutes les erreurs de mise à jour qui ne nécessitent pas d'interaction avec l'utilisateur pour être résolues. Quand le fournisseur ne peut résoudre une condition d'erreur, il stocke une copie temporaire de l'enregistrement posant problème. Quand le traitement des enregistrements est terminé, le fournisseur renvoie à l'ensemble de données client le nombre d'erreurs ayant eu lieu et copie les enregistrements non résolus dans un paquet de données résultat qu'il renvoie à l'ensemble de données client pour que celui-ci termine la réconciliation.

Les gestionnaires d'événements de tous les événements du fournisseur reçoivent les mises à jour sous la forme d'un ensemble de données client. Si le gestionnaire d'événement ne traite que certains types de mise à jour, vous pouvez filtrer l'ensemble de données en vous basant sur le type de mise à jour des enregistrements. Le filtrage des enregistrements évite au gestionnaire d'événement de parcourir des enregistrements qu'il n'utilise pas. Pour filtrer l'ensemble de données client d'après le type de mise à jour de ses enregistrements, définissez sa propriété *StatusFilter*.

Remarque Les applications doivent proposer des fonctions supplémentaires quand les mises à jour sont dirigées vers un ensemble de données qui représente plusieurs tables. Pour des détails sur la manière de procéder, voir "Application des mises à jour à des ensembles de données représentant plusieurs tables" à la page 24-13.

Modification des paquets delta avant la mise à jour de la base de données

Avant qu'un fournisseur d'ensemble de données n'applique les mises à jour à la base de données, il génère un événement *OnUpdateData*. Le gestionnaire d'événement *OnUpdateData* reçoit en paramètre une copie du paquet *Delta* sous la forme d'un ensemble de données client.

Dans le gestionnaire d'événement *OnUpdateData*, vous pouvez utiliser toutes les propriétés et méthodes de l'ensemble de données client pour modifier le paquet *Delta* avant de l'écrire dans l'ensemble de données. Une propriété s'avère particulièrement utile : *UpdateStatus*. *UpdateStatus* indique le type de modification que représente l'enregistrement en cours dans le paquet delta. Elle peut prendre l'une des valeurs énumérées dans le tableau suivant :

Tableau 24.3 Valeurs de UpdateStatus

Valeur	Description
usUnmodified	Le contenu de l'enregistrement n'a pas été modifié.
usModified	Le contenu de l'enregistrement a été modifié.
usInserted	L'enregistrement a été inséré.
usDeleted	L'enregistrement a été supprimé.

Le gestionnaire d'événement *OnUpdateData* suivant insère la date en cours dans chaque nouvel enregistrement inséré dans la base de données :

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
begin
  with DataSet do
  begin
    First;
    while not Eof do
    begin
      if UpdateStatus = usInserted then
      begin
        Edit;
        FieldByName('DateCreated').AsDateTime := Date;
        Post;
      end;
      Next;
    end;
end;
```

Comment contrôler l'application des mises à jour

L'événement *OnUpdateData* permet également au fournisseur d'indiquer comment les enregistrements d'un paquet delta sont appliqués à la base de données.

Par défaut, les modifications placées dans le paquet delta sont écrites dans la base de données en utilisant des instructions SQL UPDATE, INSERT ou DELETE générées automatiquement, par exemple :

```
UPDATE EMPLOYEES
  set EMPNO = 748, NAME = 'Smith', TITLE = 'Programmer 1', DEPT = 52
WHERE
  EMPNO = 748 and NAME = 'Smith' and TITLE = 'Programmer 1' and DEPT = 47
```

Si vous ne spécifiez pas le contraire, tous les champs des enregistrements du paquet delta sont utilisés dans les clauses UPDATE et WHERE. Vous pouvez néanmoins exclure certains champs. Vous pouvez le faire en utilisant la propriété *UpdateMode* du fournisseur. *UpdateMode* peut prendre l'une des valeurs suivantes :

Tableau 24.4 Valeurs de UpdateMode

Valeur	Signification
upWhereAll	Tous les champs sont utilisés pour la recherche (dans la clause WHERE).
upWhereChanged	Seuls les champs clés et les champs modifiés sont utilisés pour la recherche.
upWhereKeyOnly	Seuls les champs clé sont utilisés pour la recherche.

Vous pouvez avoir besoin d'un contrôle encore plus précis. Par exemple, dans l'instruction précédente, vous pouvez empêcher que le champ EMPNO ne soit modifié en le retirant de la clause UPDATE et enlever les champs TITLE et DEPT de la clause WHERE pour empêcher des conflits de mise à jour quand d'autres applications ont modifié les données. Pour spécifier la clause dans laquelle un champ spécifique apparaît, utilisez la propriété *ProviderFlags*. *ProviderFlags* est un ensemble qui peut inclure les valeurs du tableau suivant :

Tableau 24.5 Valeurs de ProviderFlags

Valeur	Description
pfInWhere	Le champ apparaît dans la clause WHERE des instructions INSERT, DELETE et UPDATE générées lorsque *UpdateMode* a pour valeur *upWhereAll* ou *upWhereChanged*.
pfInUpdate	Le champ apparaît dans la clause UPDATE des instructions UPDATE générées.
pfInKey	Le champ est utilisé dans la clause WHERE des instructions générées lorsque *UpdateMode* a pour valeur *upWhereKeyOnly*.
pfHidden	Le champ est inclus dans les enregistrements afin de garantir l'unicité, mais il ne doit pas être visible ou utilisé du côté client.

Par exemple, le gestionnaire d'événement *OnUpdateData* suivant permet au champ TITLE d'être mis à jour et utilise les champs EMPNO et DEPT pour rechercher l'enregistrement désiré. Si une erreur se produit et qu'une seconde tentative de recherche est effectuée uniquement basée sur la clé, l'instruction SQL générée recherche uniquement le champ EMPNO :

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
begin
  with DataSet do
  begin
    FieldByName('TITLE').ProviderFlags := [pfInUpdate];
    FieldByName('EMPNO').ProviderFlags := [pfInWhere, pfInKey];
    FieldByName('DEPT').ProviderFlags := [pfInWhere];
  end;
end;
```

Remarque Vous pouvez utiliser la propriété *UpdateFlags* pour contrôler comment les mises à jour sont appliquées même si vous actualisez un ensemble de données sans utiliser d'instructions SQL générées dynamiquement. Ces indicateurs indiquent quand même les champs utilisés pour rechercher des enregistrements et les champs actualisés.

Filtrage des mises à jour

Avant l'application de chaque mise à jour, un fournisseur d'ensemble de données reçoit un événement *BeforeUpdateRecord*. Vous pouvez utiliser cet événement pour modifier les enregistrements avant qu'ils ne soient appliqués, tout comme vous utilisez l'événement *OnUpdateData* pour modifier tout le paquet delta. Par exemple, le fournisseur ne compare pas les champs BLOB (comme les mémos)

quand il teste les conflits de mise à jour. Si vous voulez tester les erreurs de mise à jour concernant des champs BLOB, vous pouvez utiliser l'événement *BeforeUpdateRecord*.

Vous pouvez également utiliser cet événement pour appliquer vous-même les mises à jour ou pour filtrer et rejeter les mises à jour. Le gestionnaire d'événement *BeforeUpdateRecord* vous permet de signaler qu'une mise à jour a déjà été gérée et qu'elle ne doit pas être appliquée. Le fournisseur passe cet enregistrement sans le considérer comme une erreur de mise à jour. Cet événement vous permet d'appliquer les mises à jour dans une procédure stockée (qui ne peut être mise à jour automatiquement), en permettant au fournisseur de ne pas effectuer le traitement automatique une fois l'enregistrement mis à jour dans le gestionnaire d'événement.

Résolution des erreurs de mise à jour par le fournisseur

Si une condition d'erreur apparaît alors qu'un fournisseur d'ensemble de données tente d'appliquer un enregistrement du paquet delta, un événement *OnUpdateError* se produit. Quand le fournisseur ne peut résoudre une condition d'erreur, il stocke une copie temporaire de l'enregistrement posant problème. Quand le traitement des enregistrements est terminé, il renvoie le nombre d'erreurs ayant eu lieu et copie les enregistrements non résolus dans un paquet de données résultat qu'il envoie au client pour que celui-ci termine la réconciliation.

Dans les applications multiniveaux, ce mécanisme vous permet de gérer toutes les erreurs de mise à jour qui ne peuvent être résolues mécaniquement par le serveur d'applications tout en permettant à l'utilisateur de l'application client de corriger des conditions d'erreur.

Le gestionnaire d'événement *OnUpdateError* reçoit en paramètre une copie de l'enregistrement qui n'a pu être modifié, un code d'erreur de la base de données et une valeur indiquant si la résolution tentait d'insérer, de supprimer ou de modifier l'enregistrement. L'enregistrement posant problème est transmis dans un ensemble de données client. Vous ne devez jamais utiliser les méthodes de parcours des données dans cet ensemble de données. Par contre, pour chaque champ de cet ensemble de données vous pouvez utiliser les propriétés *NewValue*, *OldValue*, et *CurValue* afin de déterminer la cause du problème et effectuer les modifications nécessaires à la résolution de l'erreur de modification. Si le gestionnaire d'événement *OnUpdateError* peut corriger le problème, il doit définir la valeur du paramètre *Response* afin d'appliquer l'enregistrement rectifié.

Application des mises à jour à des ensembles de données représentant plusieurs tables

Quand un fournisseur d'ensemble de données génère des instructions SQL qui appliquent directement les mises à jour dans un serveur de bases de données, il lui faut le nom de la table de la base de données qui contient les enregistrements. Cela peut être automatiquement géré pour de nombreux

ensembles de données tels que les ensembles de données de type table ou les composants *TQuery* "dynamiques". Toutefois, les mises à jour automatiques représentent un problème dans le cas où le fournisseur doit appliquer les mises à jour aux données sous-jacentes à une procédure stockée renvoyant un ensemble de résultats ou à une requête multitable. Il n'est pas évident de déterminer le nom de la table dans laquelle appliquer les mises à jour.

Si la requête ou procédure stockée est un ensemble de données BDE (*TQuery* ou *TStoredProc*) et qu'elle possède un objet mise à jour associé, le fournisseur utilise cet objet. Toutefois, en l'absence d'objet mise à jour, vous devez spécifier le nom de la table par code dans un gestionnaire d'événement *OnGetTableName*. Quand ce gestionnaire d'événement a indiqué un nom de table, le fournisseur peut générer les instructions SQL appropriées pour appliquer les mises à jour.

La solution consistant à fournir le nom de la table ne fonctionne que si les mises à jour doivent être appliquées à une seule table de base de données (s'il suffit de modifier les enregistrements d'une seule table). Si la mise à jour nécessite la modification de plusieurs tables de la base de données, vous devez explicitement appliquer les mises à jour en utilisant l'événement *BeforeUpdateRecord* du fournisseur. Quand ce gestionnaire d'événement a appliqué une mise à jour, vous devez initialiser son paramètre *Applied* à *True* afin que le fournisseur ne génère pas une erreur.

Remarque Si le fournisseur est associé à un ensemble de données BDE, vous pouvez utiliser un objet mise à jour dans le gestionnaire d'événement *BeforeUpdateRecord* pour appliquer les mises à jour à l'aide d'instructions SQL personnalisées. Voir "Utilisation d'objets mise à jour pour mettre à jour un ensemble de données" à la page 20-45, pour plus de détails.

Comment répondre aux événements générés par le client

Les composants fournisseur implémentent un événement générique qui vous permet de créer vos propres appels du fournisseur directement par les ensembles de données client. C'est l'événement *OnDataRequest*.

OnDataRequest ne rentre pas dans le fonctionnement normal d'un fournisseur. C'est simplement un point d'entrée qui permet à vos ensembles de données client de communiquer directement avec les fournisseurs. Ce gestionnaire d'événement attend comme paramètre en entrée un OleVariant et renvoie un OleVariant. En utilisant des valeurs OleVariant, vous disposez d'une interface suffisamment générale pour vous adapter à tout type d'information reçue ou envoyée par le fournisseur.

Pour générer un événement *OnDataRequest*, l'application client appelle la méthode *DataRequest* de l'ensemble de données client.

Gestion des contraintes du serveur

La plupart des systèmes de gestion de bases de données relationnelles implémentent des contraintes sur les tables afin de garantir l'intégrité des données. Une contrainte est une règle contrôlant les valeurs des tables et des colonnes et les relations entre les données de colonnes de différentes tables. Ainsi, la plupart des bases de données compatibles SQL-92 gèrent les contraintes suivantes :

- NOT NULL, afin de garantir que la valeur d'une colonne est définie.
- NOT NULL UNIQUE, afin de garantir qu'une colonne contient une valeur et que cette valeur n'existe pas dans cette colonne pour un autre enregistrement.
- CHECK, afin de garantir que la valeur d'une colonne se trouve dans un intervalle spécifié ou qu'elle appartient à une plage limitée de valeurs autorisées.
- CONSTRAINT, une contrainte portant sur la table s'appliquant à plusieurs colonnes.
- PRIMARY KEY, pour désigner une ou plusieurs colonnes comme clé primaire jouant un rôle dans l'indexation.
- FOREIGN KEY, pour désigner une ou plusieurs colonnes dans une table en référençant une autre.

Remarque Cette liste n'est pas exhaustive. Votre serveur de bases de données peut gérer certaines ou toutes les contraintes précédentes, ainsi que d'autres. Pour davantage d'informations sur les contraintes qu'il gère, consultez la documentation de votre serveur.

Les contraintes d'un serveur de base de données prennent la place de nombreuses vérifications effectuées auparavant par les applications classiques de bases de données de bureau. Vous pouvez tirer profit des contraintes définies par le serveur dans vos applications de bases de données multiniveaux sans avoir à reproduire les contraintes dans le code du serveur d'applications ou dans celui de l'application client.

Si le fournisseur utilise un ensemble de données BDE, la propriété *Constraints* vous permet de reproduire et d'appliquer les contraintes du serveur aux données transmises ou reçues par les applications client. Si *Constraints* a la valeur *True* (cas par défaut), les contraintes du serveur stockées dans l'ensemble de données source figurent dans les paquets de données et affectent les tentatives de modification des données par les clients.

Important Avant que le fournisseur ne puisse transférer aux ensembles de données client les informations relatives aux contraintes, il doit lire ces informations sur le serveur de bases de données. Pour importer les contraintes de bases de données du serveur, utilisez l'explorateur SQL pour importer les contraintes du serveur de base de données et les expressions par défaut dans le dictionnaire de données. Les contraintes et les expressions par défaut du dictionnaire de données sont automatiquement proposées aux ensembles de données BDE.

Il est parfois nécessaire de ne pas appliquer les contraintes du serveur aux données envoyées à un ensemble de données client. Ainsi quand un ensemble de données client reçoit les données en paquets et autorise la modification locale des données avant la lecture de tous les enregistrements, il peut être nécessaire de désactiver certaines des contraintes du serveur qui risquent d'être déclenchées à cause d'un ensemble de données temporairement incomplet. Pour empêcher la réplication des contraintes du fournisseur par l'ensemble de données client, affectez la valeur *False* à *Constraints*. De toute façon, les ensembles de données client peuvent désactiver ou activer les contraintes en utilisant les méthodes *DisableConstraints* et *EnableConstraints*. Pour davantage d'informations sur l'activation et la désactivation des contraintes par les ensembles de données client, voir "Gestion des contraintes liées au serveur" à la page 23-35.

Chapitre

25

Création d'applications multiniveaux

Ce chapitre décrit la création d'une application de base de données client/serveur multiniveau. Une application client/serveur multiniveau est partitionnée en unités logiques, appelées niveaux, dont les différentes unités fonctionnent en coordination les unes avec les autres sur des machines séparées. Les applications multiniveaux partagent des données et communiquent par un réseau local voire même par Internet. Elles offrent de nombreux avantages, comme les applications clients simples et la logique d'entreprise centralisée.

Dans sa forme la plus simple, parfois appelée "modèle à niveau triple", une application multiniveau est partitionnée en trois niveaux :

- une application client : elle fournit une interface utilisateur sur la machine de l'utilisateur ;
- un serveur d'applications : il réside dans un emplacement central du réseau, accessible à tous les clients, et offre des services de données courants ;
- un serveur de bases de données distant : il supporte le système de gestion de base de données relationnelles (SGBDR).

Dans ce modèle à niveau triple, le serveur d'applications gère les flux de données entre les clients et le serveur de base de données distant ; il est donc parfois dénommé « courtier ou broker de données ». Avec Delphi, on ne crée généralement que le serveur d'applications et ses clients, bien que, si vous êtes réellement ambitieux, vous puissiez créer également votre propre dorsal de base de données.

Dans les applications multiniveaux plus complexes, les services supplémentaires se situent entre un client et un serveur de base de données distant. Vous pouvez, par exemple, disposer d'un courtier de services de sécurité pour prendre en charge la sécurité des transactions Internet ou de services de pont pour prendre en charge le partage de données avec des bases de données résidant sur d'autres plates-formes.

Le support Delphi pour le développement des applications multiniveaux étend la façon dont les ensembles de données client communiquent avec un composant fournisseur à l'aide de paquets de données transportables. Ce chapitre se focalise sur la création d'une application de bases de données à niveau triple. Quand vous savez créer et gérer une application à niveau triple, vous pouvez créer et ajouter des couches de services supplémentaires en fonction de vos besoins.

Avantages du modèle de base de données multiniveau

Le modèle de base de données multiniveau divise une application de base de données en unités logiques. L'application client peut se focaliser sur l'affichage des données et sur les interactions utilisateur. Dans sa forme idéale, elle ignore la façon dont les données sont stockées ou gérées. Le serveur d'applications (niveau intermédiaire) coordonne et traite les requêtes et les mises à jour de différents clients. Il gère tous les détails de la définition des ensembles de données et de l'interaction avec le serveur de bases de données.

Les avantages de ce modèle multiniveau sont les suivants :

- **Encapsulation de la logique de l'entreprise dans un niveau intermédiaire partagé**. Toutes les applications client accèdent au même niveau intermédiaire. Cela évite la redondance et le coût de gestion liés à la duplication des règles de l'entreprise pour chaque application.
- **Applications client simples**. Vous pouvez écrire vos applications client de sorte qu'elles occupent peu de place et déléguer ainsi une part plus importante du traitement aux niveaux intermédiaires. Non seulement les applications client sont de taille réduite, mais elles sont plus faciles à déployer car elles ne sont pas concernées par l'installation, la configuration et la gestion du logiciel de connectivité des bases de données (comme le moteur de bases de données Borland et les logiciels côté client du serveur de bases de données). Les applications client simples peuvent être distribuées sur Internet pour bénéficier de plus de souplesse.
- **Traitement distribué des données**. La répartition du travail d'une application entre plusieurs machines peut améliorer les performances par un meilleur équilibrage de la charge et permet aux systèmes redondants de prendre le relais en cas de défaillance d'un serveur.
- **Possibilité d'améliorer la sécurité**. Vous pouvez isoler les fonctionnalités sensibles dans des niveaux sur lesquels sont appliquées différentes restrictions d'accès. Vous bénéficiez ainsi de niveaux de sécurité souples et configurables. Les niveaux intermédiaires peuvent limiter les points d'entrée des supports sensibles, ce qui vous permet de mieux en contrôler l'accès. Si vous utilisez HTTP, CORBA ou COM+, vous pouvez tirer parti de leur modèle de sécurité.

Présentation des applications multiniveaux basées sur les fournisseurs

Le support de Delphi pour les applications multiniveaux utilise les composants de la page DataSnap et de la page Accès BD de la palette des composants ainsi qu'un module de données distant créé par un expert de la page Multi-niveaux de la boîte de dialogue Nouveaux éléments. Ils sont basés sur la possibilité des composants fournisseur d'assembler les données en paquets de données transportables et de traiter les mises à jour reçues en tant que paquets delta transportables.

Les composants requis pour une application multiniveau sont décrits dans le tableau suivant :

Tableau 25.1 Composants utilisés dans les applications multiniveaux

Composant	Description
Modules de données distants	Modules de données spécialisés agissant commeun serveur COM Automation, serveur SOAP ou serveur CORBA pour permettre aux applications client d'accéder à tous les fournisseurs qu'ils contiennent. Utilisé sur l'application serveur.
Composant fournisseur	Courtier de données qui offre des données en créant des paquets de données et qui résout les mise à jours client. Utilisé sur l'application serveur.
Composant ensemble de données client	Ensemble de données spécialisé qui utilise midas.dll ou midaslib.dcu pour gérer les données stockées sous forme de paquets de données. L'ensemble de données client est utilisé dans l'application client. Il place les mises à jour en mémoire cache localement et les applique dans des paquets delta sur le serveur d'applications.
Composants connexion	Famille de composants qui localisent le serveur, établissent les connexions et mettent l'interface *IAppServer* à la disposition des ensembles de données client. Chaque composant connexion utilise un protocole de communication particulier.

Les composants fournisseur et ensemble de données client requièrent midas.dll ou midaslib.dcu, qui gère les ensembles de données stockés sous forme de paquets de données. (Etant donné que le fournisseur est utilisé sur le serveur d'applications et que l'ensemble de données client est utilisé sur l'application client, si vous utilisez midas.dll, vous devez le déployer à la fois sur l'application serveur et sur l'application client.)

Si vous utilisez des ensembles de données BDE, le serveur d'applications peut nécessiter l'explorateur SQL qui facilite la gestion des bases de données et qui permet d'importer les contraintes serveur dans le dictionnaire des données, de sorte qu'elles peuvent être contrôlées à n'importe quel niveau d'une application multiniveau.

Remarque Vous devez acquérir des licences serveur pour déployer vos applications.

L'architecture qui accueille ces composants est décrite dans "Utilisation d'une architecture multiniveau" à la page 14-15.

Présentation d'une application à niveau triple

Les étapes numérotées suivantes illustrent une suite normale d'événements pour une application à niveau triple basée sur les fournisseurs :

1. Un utilisateur démarre l'application client. Le client se connecte au serveur d'applications (qui peut être spécifié lors de la conception ou lors de l'exécution). Si le serveur d'applications n'est pas déjà en fonctionnement, il démarre. Le client reçoit une interface *IAppServer* du serveur d'applications.
2. Le client demande des données au serveur d'applications. Un client peut demander toutes les données à la fois ou bien demander les données par fragments durant la session (extraction sur demande).
3. Le serveur d'applications récupère les données (si nécessaire, en établissant d'abord une connexion avec la base de données), les empaquette pour le client puis renvoie un paquet de données au client. D'autres informations (par exemple, les caractéristiques de l'affichage des champs) peuvent être incluses dans les métadonnées du paquet de données. Ce processus d'empaquetage des données est appelé "fourniture".
4. Le client décode le paquet de données et affiche les données à l'utilisateur.
5. Pendant que l'utilisateur interagit avec l'application client, les données sont mises à jour (des enregistrements sont ajoutés, supprimés ou modifiés). Ces modifications sont stockées par le client dans un journal de modifications.
6. Le client applique ensuite les mises à jour au serveur d'applications, généralement en réponse à une action de l'utilisateur. Pour appliquer les mises à jour, le client empaquette son journal des modifications et le transmet en tant que paquet de données au serveur.
7. Le serveur d'applications décode le paquet et émet les mises à jour (dans le contexte d'une transaction, le cas échéant). Si un enregistrement ne peut pas être validé (par exemple parce qu'une autre application a modifié l'enregistrement après que le client l'a demandé et avant que le client n'ait appliqué ses mises à jour), le serveur d'applications essaie de régulariser les modifications du client avec les données courantes ou enregistre les enregistrements qui n'ont pu être validés. Cette opération de validation des enregistrements et de sauvegarde des enregistrements à problème est dénommée "résolution".
8. Après l'opération de résolution, le serveur d'applications renvoie au client les enregistrements non transmis pour résolution ultérieure.
9. Le client régularise les enregistrements non résolus. Cela peut s'effectuer de plusieurs manières. Le client tente habituellement de remédier à la situation qui a empêché la validation des enregistrements ou bien annule les changements. Si l'erreur peut être corrigée, le client applique de nouveau les mises à jour.
10. Le client rafraîchit ses données à partir du serveur.

Structure de l'application client

Pour l'utilisateur final, l'application client d'une application multiniveau a l'apparence et le comportement d'une application à niveau double qui utilise les mises à jour en cache. L'interaction utilisateur s'opère par le biais de contrôles orientés données standard qui affichent les données à partir d'un composant *TClientDataSet*. Pour plus de détails sur l'utilisation des propriétés, événements et méthodes des ensembles de données client, voir chapitre 23, "Utilisation d'ensembles de données client".

TClientDataSet récupère les données d'un composant fournisseur, auquel il applique les mises à jour, à l'image des applications à niveau double qui utilisent un ensemble de données client et un fournisseur externe. Pour plus d'informations sur les fournisseurs, voir chapitre 24, "Utilisation des composants fournisseur". Pour plus d'informations sur les fonctionnalités d'un ensemble de données client qui facilitent sa communication avec un fournisseur, voir "Utilisation d'un ensemble de données client avec un fournisseur" à la page 23-29.

L'ensemble de données client communique avec le fournisseur par le biais de l'interface *IAppServer*. Il obtient cette interface auprès d'un composant connexion. Le composant connexion établit la connexion au serveur d'applications. Différents composants connexion sont disponibles suivant les protocoles de communication utilisés. Ces composants connexion sont présentés dans le tableau suivant :

Tableau 25.2 Composant connexion

Composant	Protocole
TDCOMConnection	DCOM
TSocketConnection	Windows sockets (TCP/IP)
TWebConnection	HTTP
TSOAPConnection	SOAP (HTTP et XML)
TCorbaConnection	CORBA (IIOP)

Remarque Delphi comprend également un composant connexion qui ne se connecte pas à un serveur d'applications mais qui fournit une interface *IAppServer* aux ensembles de données client afin qu'ils puissent communiquer avec les fournisseurs dans la même application. Ce composant, *TLocalConnection*, n'est pas requis mais facilite l'évolution ultérieure vers une application multiniveau.

Pour plus d'informations sur l'utilisation des composants connexion, voir "Connexion au serveur d'applications" à la page 25-27.

Structure du serveur d'applications

Lorsque vous configurez et exécutez un serveur d'applications, ce dernier n'établit pas de connexion avec les applications client. C'est au contraire les applications client qui initialisent et maintiennent la connexion. L'application

client utilise son composant connexion pour établir une connexion avec le serveur d'applications, qu'il utilise pour communiquer avec son fournisseur sélectionné. Tout ce processus est automatique et vous n'avez pas besoin d'écrire du code pour gérer les requêtes entrantes, ni pour fournir les interfaces.

Un serveur d'applications repose sur un module de données distant, module de données spécialisé qui gère l'interface *IAppServer*. Les applications client utilisent l'interface *IAppServer* pour communiquer avec les fournisseurs sur le serveur d'applications.

Il existe quatre types de modules de données distants :

- **TRemoteDataModule** : il s'agit d'un serveur Automation à double interface. Utilisez ce type de module de données distant si les clients utilisent DCOM, HTTP, sockets ou OLE pour se connecter au serveur d'applications, sauf si vous souhaitez installer le serveur d'applications avec MTS.
- **TMTSDataModule** : il s'agit d'un serveur Automation à double interface. Utilisez ce type de module de données distant si vous créez le serveur d'applications en tant que bibliothèque active (.DLL) installée avec MTS ou COM+. Vous pouvez utiliser des modules de données distants MTS avec DCOM, HTTP, sockets ou OLE.
- **TCorbaDataModule** : il s'agit d'un serveur CORBA. Utilisez ce type de module de données distant pour fournir des données aux clients CORBA.
- **TSoapDataModule** : il s'agit d'un module de données qui implémente un descendant de *IAppServer* comme interface invocable dans une application service Web. Utilisez ce type de module de données distant pour fournir des données aux clients accédant aux données en tant que service Web.

Si le serveur d'applications doit être déployé sur MTS ou COM+, le module de données distant contient des événements pour l'activation et la désactivation du serveur d'applications. Cela lui permet d'acquérir des connexions de bases de données quand il est activé et de les libérer quand il est désactivé.

Contenu du module de données distant

Vous pouvez inclure un composant non visuel dans le module de données distant comme dans n'importe quel module de données. Toutefois, vous devez inclure certains composants :

- Si le module de données distant expose des informations d'un serveur de bases de données, il doit comprendre un composant ensemble de données pour représenter les enregistrements de ce serveur de bases de données. D'autres composants, tel qu'un composant connexion de base de données d'un type particulier, peut être requis pour permettre à l'ensemble de données d'interagir avec un serveur de bases de données. Pour plus d'informations sur les ensembles de données, voir chapitre 18, "Présentation des ensembles de données". Pour plus d'informations sur les composants connexion de base de données, voir chapitre 17, "Connexion aux bases de données".

  Pour chaque ensemble de données que le module de données distant expose aux clients, il doit comprendre un fournisseur d'ensemble de données. Un

fournisseur d'ensemble de données rassemble les données en paquets de données qui sont envoyés aux ensembles de données client et applique les mises à jour reçues des ensembles de données client à un ensemble de données source ou à un serveur de bases de données. Pour plus d'informations sur les fournisseurs d'ensemble de données, voir chapitre 24, "Utilisation des composants fournisseur".

- Pour chaque document XML que le module de données distant expose aux clients, il doit comprendre un fournisseur XML. Un fournisseur XML se comporte comme un fournisseur d'ensemble de données, à la différence qu'il récupère les données d'un document XML et les applique à celui-ci, au lieu d'utiliser un serveur de bases de données. Pour plus d'informations sur les fournisseurs XML, voir "Utilisation d'un document XML comme source pour un fournisseur" à la page 26-9.

Remarque Ne confondez pas les composants connexion de base de données, qui connectent les bases de données à un serveur de bases de données, avec les composants connexion utilisés par les applications client d'une application multiniveau. Les composants connexion des applications multiniveaux sont disponibles sur la page DataSnap de la palette de composants.

Utilisation des modules de données transactionnels

Vous pouvez écrire un serveur d'applications tirant profit des services spécifiques pour les applications distribuées proposées par MTS (avant Windows 2000) ou COM+ (sous Windows 2000 ou plus). Pour ce faire, vous devez créer un module de données transactionnel au lieu d'un simple module de données distant.

Lors de l'écriture d'un module de données transactionnel, une application peut exploiter des services spécifiques :

- **Sécurité.** MTS et COM+ offrent au serveur d'applications une sécurité par rôle. Des rôles sont attribués aux clients pour déterminer comment ils peuvent accéder à l'interface du module de données MTS. Le module de données MTS implémente la méthode *IsCallerInRole* qui vous permet de déterminer le rôle du client connecté et d'autoriser un accès conditionnel à certaines fonctionnalités en fonction du rôle. Pour davantage d'informations sur la sécurité MTS et COM+, voir "Sécurité en fonction des rôles" à la page 39-15.
- **Regroupement des handles de bases de données.** Les modules de données transactionnels regroupent automatiquement les connexions de bases de données si vous utilisez ADO ou (si vous utilisez MTS en activant l'option MTS POOLING) le BDE. De sorte que, lorsqu'un client n'utilise plus une connexion de base de données, un autre client la réutilise. Ceci allège le trafic réseau car votre niveau intermédiaire n'a pas besoin de se déconnecter du serveur de bases de données distant et de s'y reconnecter. Lorsque les handles de bases de données sont regroupés, votre composant connexion de base de données doit affecter à sa propriété *KeepConnection*la valeur *False*, pour que votre application optimise le partage des connexions. Pour davantage

d'informations sur le regroupement des handles de bases de données, voir "Fournisseurs de ressources base de données" à la page 39-6.

- **Transactions**. Lorsque vous utilisez un module de données transactionnel, vous pouvez proposer une gestion améliorée des transactions ne se limitant pas à une seule connexion de base de données. Des modules de données transactionnels peuvent participer à des transactions recouvrant plusieurs bases de données ou contenir des fonctions qui n'impliquent aucune base de données. Pour davantage d'informations sur la gestion des transactions proposée par les objets transactionnels, voir "Gestion des transactions dans les applications multiniveaux" à la page 25-21.
- **Activation "juste à temps" et désactivation "dès que possible".** Vous pouvez écrire votre serveur de sorte que les instances du module de données distant soient activées et désactivées en fonction des besoins. Lorsque vous utilisez l'activation "juste à temps" et la désactivation "dès que possible", votre module de données distant est instancié uniquement lorsqu'il est requis pour traiter des requêtes client. Cela lui évite de mobiliser des ressources inutilisées comme les handles de bases de données.

  L'utilisation de l'activation "juste à temps" et de la désactivation "dès que possible" offre un juste milieux entre le routage de tous les clients par le biais d'une seule instance du module de données distant et la création d'une instance pour chaque connexion client. Avec une seule instance du module de données distant, le serveur d'applications doit traiter tous les appels de bases de données par le biais d'une seule connexion de base de données. Cela aboutit à un goulet d'étranglement et peut pénaliser les performances s'il y a trop de clients. Avec plusieurs instances du module de données distant, chaque instance peut gérer une connexion de base de données, ce qui évite de sérialiser les accès aux bases de données. Toutefois, cela monopolise les ressources car les autres clients ne peuvent pas utiliser la connexion à la base de données tant qu'elle est associée au module de données distant d'un client.

Pour tirer parti des transactions, de l'activation "juste à temps" et de la désactivation "dès que possible", les instances du module de données distant doivent être sans état. Cela signifie que vous devez fournir plus de support lorsque votre client utilise des informations d'état. Par exemple, le client doit passer les informations sur l'enregistrement en cours lors des lectures incrémentales. Pour plus de renseignements sur les informations d'état et les modules de données distants dans les applications multiniveaux, voir "Gestion des informations d'état dans les modules de données distants" à la page 25-23.

Par défaut, tous les appels à un module de données transactionnel générés automatiquement sont transactionnels (ils supposent qu'à la fin de l'appel le module de données peut être désactivé et que toutes les transactions en cours éventuelles peuvent être validées (commit) ou annulées (rollback). Vous pouvez écrire un module de données transactionnel dépendant d'informations d'état persistantes en initialisant à *False* la propriété *AutoComplete*, mais il ne supportera pas les transactions, ni l'activation "juste à temps", ni la désactivation "dès que possible" à moins d'utiliser une interface personnalisée.

Attention Les serveurs d'applications utilisant des modules de données transactionnels ne doivent pas ouvrir de connexions de bases de données avant l'activation du module de données. Pendant le développement de votre application, assurez-vous que tous les ensembles de données sont inactifs et que la base de données n'est pas connectée avant l'exécution de votre application. Dans l'application elle-même, vous devez ajouter du code pour ouvrir les connexions de bases de données lors de l'activation du module de données et pour les fermer lors de sa désactivation.

Regroupement des modules de données distants

Le regroupement d'objets permet de créer un cache des modules de données distants partagés par leurs clients, ce qui économise les ressources. Son fonctionnement dépend du type du module distant et du protocole de connexion.

Si vous créez un module de données transactionnel installé sur COM+, vous pouvez utiliser le gestionnaire de composants COM+ pour installer le serveur d'applications comme un objet regroupé. Pour davantage d'informations, voir "Regroupement d'objets" à la page 39-9.

Même si vous n'utilisez pas un module de données transactionnel, vous pouvez exploiter les avantages du regroupement d'objets si la connexion est constituée en utilisant HTTP (*TWebConnection* ou *TSOAPConnection*). Avec cet autre type de regroupement d'objets, vous limitez le nombre d'instances créées de votre module de données distant. Cela limite le nombre de connexions de bases de données à maintenir ainsi que d'autres ressources utilisées par le module de données distant.

Lorsque l'application serveur Web reçoit les demandes des clients (qui transmettent les appels à votre module de données distant), elle les transmet au premier module de données distant disponible du regroupement. S'il n'y a pas de module de données distant disponible, elle en crée un nouveau (jusqu'au nombre maximal que vous avez spécifié). C'est une solution intermédiaire entre le routage de tous les clients via une seule instance du module de données distant (qui peut créer un goulet d'étranglement) et la création d'une instance séparée pour chaque connexion client (qui gaspille de nombreuses ressources).

Si une instance de module de données distant du groupe ne reçoit pas de demande client pendant un moment, elle est automatiquement libérée. Cela empêche le groupe de monopoliser les ressources inutilement.

Pour configurer le regroupement d'objets avec une connexion Web (HTTP), votre module de données distant doit redéfinir la méthode *UpdateRegistry*. Dans la méthode redéfinie, appelez *RegisterPooled* quand le module de données distant est recensé et *UnregisterPooled* quand son recensement est annulé.Quand vous utilisez une méthode de regroupement d'objets, votre module de données distant doit être sans état. En effet, une seule instance peut éventuellement gérer des requêtes de plusieurs clients. Si elle dépend d'informations persistantes d'état, des clients peuvent interférer avec d'autres. Pour davantage d'informations, voir "Gestion des informations d'état dans les modules de données distants" à la page 25-23.

Sélection d'un protocole de connexion

Chaque protocole de communication que vous pouvez utiliser pour connecter vos applications client au serveur d'applications offre ses propres avantages. Avant de choisir un protocole, déterminez le nombre de clients attendus, le déploiement de votre application et les plans de développement futurs.

Utilisation de connexions DCOM

Le protocole DCOM offre l'approche de communication la plus directe car il ne requiert aucune application d'exécution particulière sur le serveur. Toutefois, comme DCOM n'est pas inclus dans Windows 95, il peut ne pas être installé sur d'anciennes machines client.

DCOM offre la seule approche qui vous permette d'utiliser les services de sécurité dans un module de données transactionnel. Ces services de sécurité reposent sur l'attribution de rôles aux appelants d'objets transactionnels. Si DCOM est utilisé, DCOM indique au système l'application client génératrice de l'appel à votre serveur d'applications (MTS ou COM+). Il est alors possible de déterminer précisément le rôle de l'appelant. Avec d'autres protocoles, toutefois, un exécutable runtime, séparé du serveur d'applications, reçoit les appels clients. Cet exécutable runtime réalise les appels COM vers le serveur d'applications pour le compte du client. De ce fait, il est impossible d'attribuer des rôles pour distinguer les clients : l'exécutable runtime est en fait le seul client. Pour davantage d'informations sur la sécurité et les objets transactionnels, voir "Sécurité en fonction des rôles" à la page 39-15.

Utilisation de connexions Socket

TCP/IP Sockets vous permet de créer des clients légers. Par exemple, si vous écrivez une application client pour le Web, vous ne pouvez pas être certain que les systèmes client supportent DCOM. Le protocole Sockets offre un plus petit dénominateur commun qui permet d'établir des connexions au serveur d'applications. Pour plus d'informations sur les sockets, voir chapitre 32, "Utilisation des sockets".

Au lieu d'instancier directement le module de données distant à partir du client (comme c'est le cas avec DCOM), le protocole Sockets utilise une application séparée sur le serveur (ScktSrvr.exe) qui accepte les requêtes client et instancie le module de données distant à l'aide de COM. Le composant connexion sur le client et ScktSrvr.exe sur le serveur sont responsables du tri des appels *IAppServer*.

Remarque ScktSrvr.exe peut s'exécuter en tant qu'application de service NT. Recensez-le dans le gestionnaire de services en le lançant avec l'option en ligne de commande -install. Vous pouvez le dérecenser avec l'option en ligne de commande -uninstall.

Avant de pouvoir utiliser une connexion socket, le serveur d'applications doit recenser sa disponibilité pour les clients utilisant une connexion socket. Par défaut, tous les nouveaux modules de données distants se recensent eux-mêmes en ajoutant un appel de la méthode *EnableSocketTransport* dans la méthode

UpdateRegistry. Vous pouvez retirer cet appel pour empêcher les connexions de socket à votre serveur d'applications.

Remarque Des serveurs plus anciens n'ajoutant pas ce recensement, vous pouvez désactiver la vérification que le serveur d'applications est recensé en désélectionnant dans ScktSrvr.exe l'élément de menu Connexions|Objets recensés seulement.

Tant qu'il n'a pas libéré une référence aux interfaces sur le serveur d'applications, le protocole Sockets n'offre aucune protection sur le serveur contre les défaillances système client. Tout en générant moins de trafic de messages que DCOM (qui envoie périodiquement des messages), l'utilisation de Sockets peut aboutir à la situation dans laquelle un serveur d'applications, non conscient de la déconnexion du client, ne libère pas ses ressources.

Utilisation de connexions Web

HTTP vous permet de créer des clients pouvant communiquer avec un serveur d'applications protégé par un coupe-feu. Les messages HTTP procurent un accès contrôlé aux applications internes afin que vos applications client soient distribuées largement en toute sécurité. Comme les connexions socket, les messages HTTP offre un plus petit dénominateur commun, que vous savez disponible, pour établir des connexions au serveur d'applications. Pour plus d'informations sur les messages HTTP, voir chapitre 27, "Création d'applications Internet".

Au lieu d'instancier directement le module de données distant à partir du client (comme cela se produit pour DCOM), les connexions basées sur HTTP utilisent un serveur d'applications Web sur le serveur (httpsrvr.dll), qui accepte les requêtes client et instancie le module de données distant à l'aide de COM. De ce fait, on les appelle également connexions Web. Le composant connexion sur le client et httpsrvr.dll sur le serveur sont responsables du tri des appels *IAppServer*.

Les connexions Web peuvent tirer parti de la sécurité SSL apportée par wininet.dll (une bibliothèque d'utilitaires Internet qui s'exécute sur le système client). Lorsque vous avez configuré le serveur Web sur le système serveur pour qu'il exige une authentification, vous pouvez spécifier le nom d'utilisateur et le mot de passe en utilisant les propriétés du composant de la connexion Web.

Comme autre mesure de sécurité, le serveur d'applications doit recenser sa disponibilité pour les clients utilisant une connexion Web. Par défaut, tous les nouveaux modules de données distants se recensent eux-mêmes en ajoutant un appel de la méthode *EnableWebTransport*dans la méthode *UpdateRegistry*. Vous pouvez retirer cet appel pour empêcher les connexions Web à votre serveur d'applications.Les connexions Web peuvent bénéficier du regroupement d'objets. Il permet à votre serveur de créer un groupe limité d'instances de module de données distant accessibles aux requêtes client. En groupant les modules de données distants, votre serveur utilise des ressources pour les modules de données et pour les connexions aux bases de données uniquement lorsque cela est nécessaire. Pour plus d'informations sur le regroupement des objets, voir "Regroupement des modules de données distants" à la page 25-9.

Au contraire de ce qui se passe avec la plupart des autres composants connexion, vous ne pouvez pas utiliser de rappels lorsque la connexion a été établie via HTTP.

Utilisation de connexions SOAP

SOAP est le protocole grâce auquel Delphi prend en charge les applications service Web. SOAP transfère les appels aux méthodes à l'aide d'un codage XML. Les connexions SOAP utilisent HTTP comme protocole de transport.

Les connexions SOAP présentent l'avantage de pouvoir fonctionner dans les applications multiplates-formes car elles sont prises en charge à la fois sous Windows et Linux. Les connexions SOAP utilisant HTTP, elles offrent les mêmes avantages que les connexions Web : HTTP fournit un plus petit dénominateur commun disponible sur tous les clients, qui peuvent communiquer avec un serveur d'applications protégé par un "coupe-feu". Pour plus d'informations sur l'utilisation de SOAP pour distribuer une application sous Delphi, voir chapitre 31, "Utilisation de services Web".

Comme dans le cas des connexions HTTP, vous ne pouvez pas utiliser de rappels lorsque la connexion a été établie via SOAP. Les connexions SOAP vous limite également à un seul module de données distant dans le serveur d'applications.

Utilisation de connexions CORBA

CORBA vous permet d'intégrer vos applications de bases de données multiniveaux à un environnement standardisé CORBA. Par exemple, lorsque vous utilisez une application client écrite en Java, seule la connexion CORBA est disponible. CORBA (et Java) étant accessible sur de multiples plates-formes, cela vous permet d'écrire des applications multiniveaux pour plusieurs plates-formes.

Grâce à CORBA, votre application bénéficie automatiquement de la répartition équilibrée de la charge, de la transparence de localisation et du basculement de serveur du logiciel runtime ORB. De plus, vous pouvez ajouter des hooks pour tirer parti des autres services CORBA.

Construction d'une application multiniveau

Les étapes générales de création d'une application de bases de données multiniveau sont les suivantes ;

1. Créez le serveur d'applications.
2. Recensez ou installez le serveur d'applications.
3. Créez une application client.

L'ordre de création est important. Vous devez créer et exécuter le serveur d'applications avant de créer un client. Lors de la conception, vous pouvez alors vous connecter au serveur d'applications pour tester votre client. Vous pouvez, bien entendu, créer un client sans spécifier le serveur d'applications lors de la conception et n'indiquer le nom du serveur qu'à l'exécution, mais cela ne vous

permet pas de voir si votre application fonctionne correctement et vous ne pourrez pas choisir les serveurs et les fournisseurs à l'aide de l'inspecteur d'objets.

Remarque Si vous ne créez pas l'application client sur le même système que le serveur, et si vous n'utilisez pas une connexion DCOM, vous pouvez recenser le serveur d'applications sur le système client. Ainsi, le composant connexion détecte le serveur d'applications à la conception, ce qui vous permet de choisir des noms de serveur et de fournisseur à partir d'une liste déroulante de l'inspecteur d'objets. Si vous utilisez une connexion Web, SOAP ou socket, le composant connexion lit le nom des serveurs recensés sur la machine serveur.

Création du serveur d'applications

La création d'un serveur d'applications est très similaire à la création de la plupart des applications de bases de données. La principale différence réside dans le fait que le serveur d'applications inclut utilise un module de données distant.

Pour créer un serveur d'applications, exécutez les étapes suivantes :

1 Démarrez un nouveau projet :

- Si vous utilisez SOAP comme protocole de transport, ce sera une nouvelle application service Web. Choisissez Fichier | Nouveau | Autre et, sur la page Services Web du dialogue Nouveaux éléments, choisissez Application service Web.
- Pour tout autre protocole de transport, vous n'avez qu'à choisir Fichier | Nouveau | Application.

Enregistrez le nouveau projet.

2 Ajoutez un nouveau module de données distant au projet. Dans le menu principal, choisissez Fichier | Nouveau | Autre et, dans la page Multiniveaux de la boîte de dialogue Nouveaux éléments, sélectionnez

- **Module de données distant** si vous créez un serveur COM Automation auxquels les clients peuvent accéder à l'aide de DCOM, HTTP, Sockets ou OLEnterprise ;
- **Module de données transactionnel** si vous créez un module de données distant exécuté dans MTS ou COM+. Les connexions peuvent être établies à l'aide de DCOM, HTTP ou de sockets. Mais seul DCOM gère les services de sécurité.
- **Module de données CORBA** si vous créez un serveur CORBA.
- **Module de données SOAP** si vous créez un serveur SOAP dans une application service Web.

Pour plus de détails sur la configuration d'un module de données distant, voir "Configuration du module de données distant" à la page 25-15.

Remarque Les modules de données distants sont plus que de simples modules de données. Le module de données CORBA fait office de serveur CORBA. Le module de données SOAP implémente une interface invocable dans une application service Web. Les autres modules de données sont des objets COM Automation.

3. Placez sur le module de données les composants ensemble de données appropriés, et configurez-les pour accéder au serveur de base de données.
4. Placez sur le module de données un composant *TDataSetProvider* pour chaque ensemble de données. Ce fournisseur est nécessaire au courtage des demandes client et à l'empaquetage des données. Attribuez à la propriété *DataSet* de chaque composant fournisseur le nom de l'ensemble de données devant être accessible. Vous pouvez définir d'autres propriétés pour le fournisseur. Voir chapitre 24, "Utilisation des composants fournisseur", pour plus de détails sur la configuration d'un fournisseur.

   Si vous manipulez des données de documents XML, vous pouvez utiliser un composant *TXMLTransformProvider* au lieu d'un ensemble de données et du composant *TDataSetProvider*. Lorsque vous utilisez *TXMLTransformProvider*, définissez la propriété *XMLDataFile* afin de spécifier le document XML dont proviennent les données et auquel les mises à jour sont appliquées.
5. Ecrivez du code pour le serveur d'applications afin d'implémenter les événements, les règles d'entreprise partagées, les validations de données partagées et la sécurité partagée. Lorsque vous écrivez ce code, vous pouvez
   - Etendre l'interface du serveur d'applications afin d'offrir à l'application client d'autres possibilités pour appeler le serveur. L'extension de l'interface du serveur d'applications est décrite dans "Extension de l'interface du serveur d'applications" à la page 25-19.
   - Fournir une gestion des transactions au-delà des transactions automatiquement créées lorsque les mises à jour sont appliquées. La gestion des transactions dans les applications de bases de données multiniveaux est décrite dans "Gestion des transactions dans les applications multiniveaux" à la page 25-21.
   - Créer des relations maître / détail entre les ensembles de données de votre serveur d'applications. Les relations maître / détail sont décrites dans "Gestion des relations maître / détail" à la page 25-22.
   - Faire en sorte que votre serveur d'applications soit sans état. La gestion des informations d'état est décrite dans "Gestion des informations d'état dans les modules de données distants" à la page 25-23.
   - Diviser votre serveur d'applications en plusieurs modules de données distants. L'utilisation de plusieurs modules de données distants est décrite dans "Utilisation de plusieurs modules de données distants" à la page 25-24.
6. Enregistrez, compilez et recensez ou installez le serveur d'applications. Le recensement d'un serveur d'applications est décrit dans "Recensement du serveur d'applications" à la page 25-25.

7 Si votre serveur d'applications n'utilise pas DCOM ou SOAP, vous devez installer le logiciel runtime qui reçoit les messages client, instancie le module de données distant et trie les appels d'interface.

- Pour le protocole TCP/IP Sockets, il s'agit d'une application de répartition de sockets, Scktsrvr.exe.
- Pour les connexions HTTP, il s'agit de httpsrvr.dll, une dll ISAPI/NSAPI qui doit être installée avec votre serveur Web.
- Pour CORBA, il s'agit du VisiBroker ORB.

Configuration du module de données distant

Lorsque vous créez le module de données distant, vous devez fournir certaines informations indiquant comment il répond aux requêtes client. Ces informations varient avec le type de module de données distant. Voir "Structure du serveur d'applications" à la page 25-5, pour plus d'informations sur le type de module de données distant nécessaire.

Configuration de TRemoteDataModule

Pour ajouter un composant *TRemoteDataModule* dans votre application, choisissez Fichier | Nouveau | Autre et sélectionnez Module de données distant dans la page Multi-niveaux de la boîte de dialogue Nouveaux éléments. L'expert Module de données distant apparaît.

Vous devez fournir un nom de classe pour votre module de données distant. Il s'agit du nom de base d'un descendant de *TRemoteDataModule* que votre application crée. Il s'agit aussi du nom de base de l'interface de cette classe. Par exemple, si vous spécifiez le nom de classe *MyDataServer*, l'expert crée une nouvelle unité en déclarant *TMyDataServer*, un descendant de *TRemoteDataModule*, *qui implémente IMyDataServer, un descendant de IAppServer*.

Remarque Vous pouvez ajouter vos propres propriétés et méthodes à la nouvelle interface. Pour plus d'informations, voir "Extension de l'interface du serveur d'applications" à la page 25-19.

Vous devez spécifier le modèle threading dans l'expert Module de données distant. Vous pouvez choisir Thread Unique, Thread Appartement, Thread Libre ou Les deux.

- Si vous choisissez Thread Unique, COM fait en sorte qu'une seule requête client soit traitée à un moment donné. Aucune requête client ne peut entrer en conflit avec une autre.
- Si vous choisissez Thread Appartement, COM fait en sorte que toute instance de votre module de données distant traite une seule requête à un moment donné. Lorsque vous écrivez du code dans une bibliothèque Thread Appartement, vous devez prévenir les conflits de thread si vous utilisez des variables globales ou des objets non contenus dans le module de données distant. C'est le modèle recommandé si vous utilisez des ensembles de données BDE. (Veuillez noter que vous aurez besoin d'un composant session

dont la propriété *AutoSessionName* vaut *True* pour gérer les problèmes de threading sur les ensembles de données BDE.)

- Si vous choisissez Thread Libre, votre application peut recevoir des requêtes client simultanées sur plusieurs threads. Vous devez faire en sorte que votre application soit compatible avec les threads. Comme plusieurs clients peuvent simultanément accéder à votre module de données distant, vous devez protéger vos données d'instance (propriétés, objets contenus, etc.) ainsi que les variables globales. C'est le modèle recommandé si vous utilisez des ensembles de données ADO.
- Si vous choisissez Les deux, votre bibliothèque fonctionne de la même façon que lorsque vous choisissez Thread Libre, à la différence que tous les rappels (appels vers les interfaces client) sont automatiquement sérialisés.
- Si vous choisissez Neutre, le module de données distant peut recevoir des appels simultanés dans des threads séparés, comme dans le modèle Thread Libre, mais COM s'assure qu'il n'y a pas deux threads accédant à la même méthode au même moment.

Si vous créez un fichier .EXE, vous devez également spécifier le type d'instanciation à utiliser. Vous pouvez choisir Instance unique ou Instance multiple (l'instanciation interne ne s'applique que si le code client fait partie du même espace de processus.)

- Si vous choisissez Instance unique, chaque connexion client lance sa propre instance du fichier exécutable. Ce processus instancie une seule instance du module de données distant, qui est dédié à la connexion client.
- Si vous choisissez Instance multiple, une seule instance de l'application (processus) instancie tous les modules de données distants créés pour les clients. Chaque module de données distant est dédié à une seule connexion client, mais ils partagent tous le même espace de processus.

Configuration de TMTSDataModule

Pour ajouter un composant *TMTSDataModule* dans votre application, choisissez Fichier|Nouveau|Autre et sélectionnez Module de données transactionnel dans la page Multi-niveaux de la boîte de dialogue Nouveaux éléments. L'expert Module de données transactionnel apparaît.

Vous devez fournir un nom de classe pour votre module de données distant. Il s'agit du nom de base d'un descendant de *TMTSDataModule* que votre application crée. Il s'agit aussi du nom de base de l'interface de cette classe. Par exemple, si vous spécifiez le nom de classe *MyDataServer*, l'expert crée une nouvelle unité en déclarant *TMyDataServer*, un descendant de *TMTSDataModule*, qui implémente *IMyDataServer*, un descendant de *IAppServer*.

Remarque Vous pouvez ajouter vos propres propriétés et méthodes à votre nouvelle interface. Pour plus d'informations, voir "Extension de l'interface du serveur d'applications" à la page 25-19.

Vous devez spécifier le modèle threading dans l'expert Module de données transactionnel. Choisissez Unique, Appartement ou Les deux.

- Si vous choisissez Unique, les requêtes client sont sérialisées afin que l'application n'en traite qu'une seule à la fois. Aucune requête client ne peut entrer en conflit avec une autre.
- Si vous choisissez Appartement, le système fait en sorte que toute instance de votre module de données distant traite une seule requête à un moment donné ; les appels utilisent toujours le même thread. Vous devez prévenir les conflits de thread si vous utilisez des variables globales ou des objets non contenus dans le module de données distant. Au lieu d'utiliser des variables globales, vous pouvez utiliser le gestionnaire de propriétés partagées. Pour plus d'informations sur le gestionnaire de propriétés partagées, voir "Gestionnaire de propriétés partagées" à la page 39-7.
- Si vous choisissez Les deux, MTS appelle l'interface du module de données distant de la même façon que lorsque vous choisissez Appartement. Toutefois, tous les rappels réalisés en direction des applications clients sont automatiquement sérialisés afin qu'aucun n'entre en conflit avec un autre.

Remarque Le modèle Appartement sous MTS ou COM + est différent du modèle correspondant sous DCOM.

Vous devez aussi spécifier les attributs des transactions de votre module de données distant. Vous pouvez choisir parmi les options suivantes :

- Requiert une transaction. Lorsque vous sélectionnez cette option, chaque fois qu'un client utilise l'interface de votre module de données distant, l'appel est exécuté dans le contexte d'une transaction. Si l'appelant fournit une transaction, il est inutile qu'une nouvelle transaction soit créée.
- Requiert une nouvelle transaction. Lorsque vous sélectionnez cette option, chaque fois qu'un client utilise l'interface de votre module de données distant, une nouvelle transaction est automatiquement créée pour l'appel.
- Supporte les transactions. Lorsque vous sélectionnez cette option, votre module de données distant peut être utilisé dans le contexte d'une transaction, mais l'appelant doit fournir la transaction lorsqu'il appelle l'interface.
- Ne supporte pas les transactions. Lorsque vous sélectionnez cette option, votre module de données distant ne peut pas être utilisé dans le contexte de transactions.

Configuration de TSoapDataModule

Pour ajouter un composant *TSoapDataModule* dans votre application, choisissez Fichier | Nouveau | Autre et sélectionnez Module de données SOAP dans la page Multi-niveaux de la boîte de dialogue Nouveaux éléments. L'expert Module de données SOAP apparaît.

Vous devez fournir un nom de classe pour votre module de données SOAP. Il s'agit du nom de base d'un descendant de *TSoapDataModule* que votre application crée. Il s'agit aussi du nom de base de l'interface de cette classe. Par exemple, si vous spécifiez le nom de classe *MyDataServer*, l'expert crée une nouvelle unité en déclarant *TMyDataServer*, un descendant de *TSoapDataModule*, qui implémente *IMyDataServer*, un descendant de *IAppServer*.

Vous pouvez modifier les définitions de l'interface générée et le descendant de *TSoapDataModule* en ajoutant vos propres propriétés et méthodes. Ces propriétés et méthodes ne sont pas appelées automatiquement, mais les applications client qui demandent votre nouvelle interface par son nom peuvent utiliser toutes les propriétés et toutes les méthodes que vous avez ajoutées.

Remarque Pour utiliser *TSoapDataModule*, le nouveau module de données doit avoir été ajouté à une application service Web. L'interface *IAppServer* est une interface invocable, recensée dans la section d'initialisation de la nouvelle unité. Cela permet au composant invocateur dans le module Web principal de faire suivre tous les appels entrants à votre module de données.

Configuration de TCorbaDataModule

Pour ajouter un composant *TCorbaDataModule* dans votre application, choisissez Fichier|Nouveau et sélectionnez Module de données CORBA dans la page Multi-niveaux de la boîte de dialogue Nouveaux éléments. L'expert Module de données CORBA apparaît.

Vous devez fournir un nom de classe pour votre module de données distant. Il s'agit du nom de base d'un descendant de *TCorbaDataModule* que votre application crée. Il s'agit aussi du nom de base de l'interface de cette classe. Par exemple, si vous spécifiez le nom de classe *MyDataServer*, l'expert crée une nouvelle unité en déclarant *TMyDataServer*, un descendant de *TSoapDataModule*, qui implémente *IMyDataServer*, un descendant de *IAppServer*.

Remarque Vous pouvez ajouter vos propres propriétés et méthodes à votre nouvelle interface. Pour plus d'informations sur l'ajout d'éléments à l'interface de votre module de données, voir "Extension de l'interface du serveur d'applications" à la page 25-19.

L'expert module de données CORBA vous permet de spécifier comment votre application serveur crée les instances du module de données distant. Vous pouvez choisir entre le mode partagé ou instance par client.

- Lorsque vous choisissez le mode partagé, votre application crée une instance unique du module de données distant qui gère toutes les requêtes client. Ce modèle est traditionnellement utilisé dans le développement CORBA.
- Lorsque vous choisissez le mode une instance par client, une nouvelle instance du module de données distant est créée pour chaque connexion client. Cette instance persiste jusqu'à ce que le délai d'inactivité soit écoulé sans message de la part du client. Cela permet certes au serveur de libérer les instances lorsqu'elles ne sont plus utilisées par les clients, mais présente le risque que le serveur soit libéré prématurément si le client n'utilise pas l'interface du serveur pendant une longue période.

Remarque Contrairement au modèle d'instanciation des serveurs COM, où est déterminé le nombre d'instances du processus qui sont exécutées, dans le modèle CORBA, l'instanciation détermine le nombre d'instances de votre objet qui sont créées. Elles sont toutes créées dans une seule instance de l'exécutable serveur.

Outre le modèle d'instanciation, vous devez spécifier le modèle threading dans l'expert Module de données CORBA. Vous pouvez choisir Monothread ou Multithread.

- Si vous choisissez Monothread, chaque instance du module de données distant est assurée de ne recevoir qu'une requête client à un moment donné. Vous pouvez accéder en sécurité aux objets contenus dans votre module de données distant. Toutefois, vous devez prévenir les conflits de thread lorsque vous utilisez des variables globales ou des objets non contenus dans le module de données distant.
- Si vous choisissez Multithread, chaque connexion client dispose de son propre thread. Toutefois, votre application peut être appelée par plusieurs clients simultanément, chacun sur un thread différent. Vous devez prévenir l'accès simultané aux données d'instance ainsi qu'à la mémoire globale. L'écriture de serveurs multithreads est délicate lorsque vous utilisez une instance partagée du module de données distant car vous devez protéger toute l'utilisation des objets contenus dans le module.

Extension de l'interface du serveur d'applications

Les applications client interagissent avec le serveur d'applications en créant, ou en s'y connectant, une instance du module de données distant. Elles utilisent son interface comme base de toute communication avec le serveur d'applications.

Vous pouvez effectuer un ajout à l'interface de votre module de données distant afin d'améliorer la prise en charge de vos applications client. Cette interface est un descendant de *IAppServer* ; elle est automatiquement créée par l'expert lorsque vous créez le module de données distant.

Pour effectuer un ajout à l'interface du module de données distant, vous pouvez

- Choisir la commande Ajouter à l'interface dans le menu Edition de l'EDI. Indiquez si vous ajoutez une procédure, une fonction ou une propriété et entrez la syntaxe. Lorsque vous cliquez sur OK, vous vous retrouvez dans l'éditeur de code sur l'implémentation du nouveau membre de votre interface.
- Utiliser l'éditeur de bibliothèque de types. Sélectionnez l'interface de votre serveur d'applications dans l'éditeur de bibliothèque de types et cliquez sur le bouton d'outil correspondant au type de membre d'interface (méthode ou propriété) que vous ajoutez. Nommez votre membre d'interface dans la page Attributs, spécifiez les paramètres et le type dans la page Paramètres puis rafraîchissez la bibliothèque de types. Pour plus d'informations sur l'utilisation de l'éditeur de bibliothèque de types, voir chapitre 34, "Utilisation des bibliothèques de types". Notez que de nombreuses fonctionnalités que vous pouvez spécifier dans l'éditeur de bibliothèque de types (comme l'aide contextuelle, la version, etc.) ne s'appliquent pas aux interfaces CORBA. Toutes les valeurs que vous spécifiez pour ces dernières dans l'éditeur de bibliothèque de types sont ignorées.

Remarque Aucune de ces approches ne fonctionne si vous implémentez *TSoapDataModule*. Pour les descendants de *TSoapDataModule*, vous devez modifier l'interface du serveur directement.

Le comportement de Delphi lorsque vous ajoutez de nouvelles entrées à l'interface, en utilisant l'éditeur de bibliothèque de types ou la commande Ajouter à l'interface, diffère selon que vous créez un serveur basé sur COM (*TRemoteDataModule* ou *TMTSDataModule*) ou un serveur CORBA (*TCorbaDataModule*).

- Lorsque vous ajoutez une interface COM, vos modifications sont ajoutées au code source de votre unité et au fichier de la bibliothèque de types (.TLB).
- Lorsque vous effectuez un ajout à l'interface CORBA, vos modifications sont répercutées dans le code source de votre unité et dans l'unité _TLB automatiquement générée. L'unité _TLB est ajoutée à la clause **uses** de votre unité. Vous devez ajouter cette unité à la clause uses dans votre application client si sous souhaitez tirer parti de la liaison anticipée. De plus, vous pouvez enregistrer un fichier .IDL à partir de l'éditeur de bibliothèque de types à l'aide de la commande Fichier | Enregistrer sous. Le fichier .IDL est requis pour recenser l'interface dans le référentiel d'interfaces et dans le démon d'activation d'objets.

Remarque Vous devez enregistrer explicitement le fichier TLB en choisissant Rafraîchir dans l'éditeur de bibliothèque de types et en sauvegardant ensuite les modifications depuis l'EDI.

Une fois que vous avez effectué un ajout à l'interface de votre module de données distant, localisez les propriétés et les méthodes ajoutées à l'implémentation de votre module de données distant. Ajoutez du code pour terminer cette implémentation en remplissant le corps des nouvelles méthodes.

Les applications client appellent les extensions de votre interface à l'aide de la propriété *AppServer* de leur composant connexion. Pour plus d'informations sur la procédure à suivre, voir "Appel des interfaces serveur" à la page 25-33.

Ajout de rappels à l'interface du serveur d'applications

Vous pouvez faire que le serveur d'applications appelle votre application client en introduisant un rappel. Pour ce faire, l'application client passe une interface à l'une des méthodes du serveur d'applications, et le serveur d'applications appelle ensuite cette méthode. Cependant, si vos extensions à l'interface du module de données distant comprennent des rappels, vous ne pouvez pas utiliser de connexion HTTP ou SOAP. *TWebConnection* ne supporte pas les rappels. Si vous utilisez une connexion de type socket, les applications client doivent indiquer si elles utilisent ou non les rappels par la définition de la propriété *SupportCallbacks*. Tous les autres types de connexions supportent automatiquement les rappels.

Extension de l'interface d'un serveur d'applications transactionnel

Si vous utilisez les transactions ou l'activation juste à temps, vous devez vous assurer que toutes les nouvelles méthodes appellent *SetComplete*pour indiquer

leur achèvement. Cela permet l'arrêt des transactions et la désactivation du module de données distant.

De plus, vous ne pouvez pas renvoyer depuis ces nouvelles méthodes de valeurs qui rendraient possible une communication directe entre les clients et les objets ou interfaces du serveur d'applications sans fournir une référence fiable. Si vous utilisez un module de données MTS sans état, oublier d'utiliser une référence fiable peut provoquer des blocages car vous ne garantissez pas que le module de données distant soit actif. Pour davantage d'informations sur les références fiables, voir "Transfert de références d'objets" à la page 39-21.

Gestion des transactions dans les applications multiniveaux

Lorsque les applications client appliquent les mises à jour sur le serveur d'applications, le composant fournisseur enveloppe automatiquement le processus d'application des mises à jour et de résolution des erreurs dans une transaction. Cette transaction est validée si le nombre d'enregistrements problématiques n'est pas supérieur à la valeur *MaxErrors* spécifiée comme argument à la méthode *ApplyUpdates*. Sinon, elle est annulée.

De plus, vous pouvez améliorer la prise en charge des transactions sur votre application serveur en ajoutant un composant connexion de base de données ou en gérant la transaction directement en envoyant du code SQL au serveur de bases de données. La gestion des transactions est la même que dans une application à niveau double. Pour plus d'informations sur ce type de gestion de transactions, voir "Gestion des transactions" à la page 17-6.

Si vous utilisez un module de données distant transactionnel, vous pouvez améliorer la prise en charge des transactions à l'aide des transactions MTS ou COM+. Ces transactions peuvent inclure toute logique d'entreprise sur votre serveur d'applications et ne se limitent pas à la gestion de l'accès aux bases de données. De plus, comme elles gèrent la validation en deux phases, ces transactions peuvent englober plusieurs bases de données.

Seuls les composants d'accès aux données BDE et ADO gèrent les validations en deux phases. N'utilisez pas les composants InterbaseExpress ou dbExpress si vous utilisez des transactions à cheval sur plusieurs bases de données.

Attention Si vous utilisez le BDE, la validation en deux phases n'est entièrement implémentée que sur les bases de données Oracle7 et MS-SQL. Si votre transaction implique plusieurs bases de données et que certaines d'entre elles sont des serveurs distants autres que Oracle7 ou MS-SQL, il existe un risque minime d'échec partiel. Toutefois, dans toute base de données, vous pouvez utiliser les transactions.

Par défaut dans un module de données transactionnel, tous les appels de *IAppServer* sont transactionnels. Il suffit d'activer l'attribut de transaction de votre module de données pour indiquer qu'il doit participer aux transactions. De plus, vous pouvez étendre l'interface du serveur d'applications pour inclure des appels de méthode qui encapsulent des transactions que vous avez définies.

Si l'attribut de transaction indique que le module de données distantnécessite une transaction, tous les appels des clients sont automatiquement inclus dans une transaction jusqu'à ce que vous spécifiez que la transaction est achevée. Ces appels réussissent ou sont annulés en bloc.

Remarque Ne combinez pas les transactions MTS ou COM+ avec des transactions explicites créées par un composant connexion de base de données ou à l'aide de commandes SQL explicites. Lorsque votre module de données transactionnel est répertorié dans une transaction, il répertorie automatiquement tous les appels de votre base de données dans la transaction.

Pour plus d'informations sur l'utilisation des transactions MTS et COM+, voir "Support transactionnel MTS et COM+" à la page 39-9.

Gestion des relations maître / détail

Vous pouvez créer des relations maître/détail entre les ensembles de données client de votre application client de la même façon qu'avec tout ensemble de données de type table. Pour plus d'informations sur la définition de relations maître/détail de cette manière, voir "Création de relations maître/détail" à la page 18-40.

Toutefois, cette approche présente deux inconvénients majeurs :

- Tous les enregistrements de la table détail doivent provenir du serveur d'applications même si elle n'utilise qu'un ensemble détail à la fois. (Ce problème peut être atténué à l'aide de paramètres. Pour plus d'informations, voir "Limitation des enregistrements avec des paramètres" à la page 23-34.)
- Il est très difficile d'appliquer les mises à jour car les ensembles de données client les appliquent au niveau de l'ensemble de données alors que les mises à jour maître/détail englobent plusieurs ensembles de données. Même dans un environnement à niveau double, où vous pouvez utiliser le composant connexion de base de données pour appliquer les mises à jour pour plusieurs tables dans une seule transaction, l'application des mises à jour dans des fiches maître/détail est délicate.

Dans les applications multiniveaux, vous pouvez éviter ces problèmes en utilisant des tables imbriquées pour représenter la relation maître/détail. Lorsque les données proviennent d'ensembles de données, définissez une relation maître/détail entre les ensembles de données sur le serveur d'applications puis initialisez la propriété *DataSet* de votre composant fournisseur sur la table maître. Pour utiliser des tables imbriquées pour représenter les relations maître/détail lorsque les données proviennent de documents XML, utilisez un fichier de transformation qui définit les ensembles détails imbriqués.

Lorsque les clients appellent la méthode *GetRecords* du fournisseur, il inclut automatiquement l'ensemble de données détail en tant que champ ensemble de données dans les enregistrements du paquet de données. Lorsque les clients appellent la méthode *ApplyUpdates* du fournisseur, il traite automatiquement l'application des mises à jour dans l'ordre adéquat.

Gestion des informations d'état dans les modules de données distants

L'interface *IAppServer*, qui contrôle toute la communication entre les ensembles de données client et les fournisseurs sur le serveur d'applications, est généralement sans état. Lorsqu'une application est sans état, elle ne "mémorise" pas ce qui s'est produit lors des appels précédents effectués par le client. Cette caractéristique est utile si vous regroupez les connexions aux bases de données dans un module de données transactionnel, car votre serveur d'applications n'a pas besoin de différencier les connexions par rapport à des informations persistantes comme l'enregistrement en cours. De même, cette absence d'information d'état est importante lorsque vous partagez les instances des modules de données distants entre de nombreux clients, comme cela se produit avec l'activation "juste à temps", le regroupement des objets, ou les serveurs CORBA classique. Les modules de données SOAP doivent être sans état.

Dans certaines circonstances, toutefois, vous voudrez maintenir des informations d'état entre les appels au serveur d'applications. Par exemple, lorsque vous demandez des données à l'aide de la lecture incrémentale, le fournisseur sur le serveur d'applications doit "mémoriser" des informations sur les appels précédents (l'enregistrement en cours).

Avant et après tous les appels à l'interface *IAppServer* effectués par l'ensemble de données client (AS_*ApplyUpdates*, AS_*Execute*, AS_*GetParams*, AS_*GetRecords* ou AS_*RowRequest*), il reçoit un événement dans lequel il peut envoyer ou extraire des informations d'état personnalisées. De même, avant et après la réponse des fournisseurs aux appels générés par les clients, ils reçoivent des événements dans lesquels ils peuvent extraire ou envoyer des informations d'état personnalisées. Avec ce mécanisme, vous pouvez communiquer des informations d'état persistantes entre les applications client et le serveur d'applications, même si le serveur d'applications est sans état.

Par exemple, considérez un ensemble de données qui représente la requête paramétrée suivante :

```
SELECT * from CUSTOMER WHERE CUST_NO > :MinVal ORDER BY CUST_NO
```

Pour utiliser la lecture incrémentale dans un serveur d'applications sans état, vous pouvez faire ce qui suit :

- Lorsque le fournisseur rassemble un ensemble d'enregistrements dans un paquet de données, il note la valeur de CUST_NO sur le dernier enregistrement du paquet :

```
TRemoteDataModule1.DataSetProvider1GetData(Sender: TObject; DataSet: TCustomClientDataSet);
begin
  DataSet.Last; { se positionne sur le dernier enregistrement }
  with Sender as TDataSetProvider do
    Tag := DataSet.FieldValues['CUST_NO']; {enregistre la valeur de CUST_NO }
end;
```

- Le fournisseur envoie cette dernière valeur CUST_NO au client après avoir envoyé le paquet de données :

```
TRemoteDataModule1.DataSetProvider1AfterGetRecords(Sender: TObject;
                  var OwnerData: OleVariant);
begin
  with Sender as TDataSetProvider do
    OwnerData := Tag; {envoie la dernière valeur de CUST_NO }
end;
```

- Sur le client, l'ensemble de données client enregistre cette dernière valeur de CUST_NO :

```
TDataModule1.ClientDataSet1AfterGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
  with Sender as TClientDataSet do
    Tag := OwnerData; {enregistre la dernière valeur de CUST_NO }
end;
```

- Avant de lire un paquet de données, le client envoie la dernière valeur de CUST_NO qu'il a reçue :

```
TDataModule1.ClientDataSet1BeforeGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
  with Sender as TClientDataSet do
  begin
    if not Active then Exit;
    OwnerData := Tag; { envoie la dernière valeur de CUST_NO au serveur d'applications }
  end;
end;
```

- Enfin, sur le serveur, le fournisseur utilise la dernière valeur de CUST_NO envoyée comme valeur minimale dans la requête :

```
TRemoteDataModule1.DataSetProvider1BeforeGetRecords(Sender: TObject;
                   var OwnerData: OleVariant);
begin
  if not VarIsEmpty(OwnerData) then
    with Sender as TDataSetProvider do
      with DataSet as TSQLDataSet do
      begin
        Params.ParamValues['MinVal'] := OwnerData;
        Refresh; { oblige la requête à se réexécuter }
    end;
end;
```

Utilisation de plusieurs modules de données distants

Vous pouvez structurer votre serveur d'applications de telle sorte qu'il utilise plusieurs modules de données distants. Le recours à plusieurs modules de données distants vous permet de partitionner votre code, d'organiser un serveur d'applications volumineux en plusieurs unités relativement autonomes.

Bien que vous puissiez toujours créer sur le serveur d'applications plusieurs modules de données distants fonctionnant indépendamment, Delphi propose un

modèle dans lequel un module de données distant "parent" principal répartit les connexions des clients parmi les modules de données distants "enfant".

Pour créer le module de données distant parent, vous devez étendre son interface *IAppServer* en ajoutant les propriétés qui exposent les interfaces des modules de données distants enfant. En d'autres termes, pour chaque module de données distant enfant, ajoutez une propriété à l'interface du module de données parent dont la valeur représente l'interface *IAppServer* interface du module de données enfant. La méthode d'accès en lecture à une propriété doit ressembler à ceci :

```
function ParentRDM.Get_ChildRDM: IChildRDM;
begin
  {Le module de données distant parent utilise un composant fabricant défini dans
  l'unité du module de données distant enfant.
  Cela est plus efficace dans le cas de la création de plusieurs enfants pour différents
  clients }
  Result := ChildRDMFactory.CreateCOMObject(nil) as IChildRDM;
  Result.ParentRDM := Self;
end;
```

Pour plus d'informations sur l'extension de l'interface du module de données distant parent, voir "Extension de l'interface du serveur d'applications" à la page 25-19.

Conseil Vous pouvez également étendre l'interface de chaque module de données enfant en exposant l'interface du module de données parent ou les interfaces des autres modules de données enfant. Cela facilite la communication entre les différents modules de données de votre serveur d'applications.

Une fois que les propriétés qui représentent les modules de données distants enfant ont été ajoutées au module de données distant principal, les applications client n'ont pas besoin d'établir de connexions distinctes à chaque module de données distant sur le serveur d'applications. En effet, elles partagent une connexion unique au module de données distant parent, qui répartit ensuite les messages parmi les modules de données "enfant". Etant donné que chaque application client utilise la même connexion pour chaque module de données distant, les modules de données distants peuvent partager une connexion de base de données unique, ce qui permet d'économiser les ressources. Pour plus d'informations sur la partage d'une connexion unique par les applications enfant, voir "Connexion à un serveur d'applications qui utilise plusieurs modules de données" à la page 25-35.

Recensement du serveur d'applications

Pour que les applications client puissent localiser et utiliser un serveur d'applications, celui-ci doit être recensé ou installé. (Cela n'est pas absolument nécessaire pour les serveurs d'applications CORBA, mais le recensement demeure recommandé.)

- Si le serveur d'applications utilise DCOM, HTTP ou des sockets comme protocole de communication, il fait office de serveur Automation et doit être

recensé comme tout autre serveur COM. Pour plus d'informations sur le recensement d'un serveur COM, voir "Recensement d'un objet COM" à la page 36-18.

- Si vous utilisez un module de données transactionnel, il ne faut pas recenser le serveur d'applications. Vous devez à la place l'installer dans MTS ou COM+. Pour davantage d'informations sur l'installation d'objets transactionnels, voir "Installation d'objets transactionnels" à la page 39-24.
- Lorsque le serveur d'applications utilise SOAP, il doit être recensé afin que son interface soit reconnue par le répartiteur SOAP.
- Lorsque le serveur d'applications utilise SOAP, l'application doit être une application service Web. En tant que telle, elle doit être recensée sur votre serveur Web, afin qu'elle puisse recevoir les messages HTTP entrants. De plus, si vous souhaitez que des clients non écrits avec Delphi accèdent aux interfaces de votre application, vous pouvez publier un document WSDL décrivant les interfaces invocables de votre application. Pour plus informations sur l'exportation d'un document WSDL pour une application service Web, voir "Génération de documents WSDL pour une application service Web" à la page 31-8.
- Lorsque le serveur d'applications utilise CORBA, le recensement est facultatif. Si vous souhaitez que les applications client utilisent la liaison dynamique à votre interface, vous devez installer l'interface du serveur dans le référentiel d'interfaces. De plus, si vous souhaitez que les applications client démarrent le serveur d'applications lorsqu'il n'est pas encore exécuté, ce dernier doit être recensé avec l'OAD (Object Activation Daemon).

Création de l'application client

A bien des égards, la création d'une application client multiniveau est similaire à la création d'un client à niveau double qui utilise un ensemble de données client pour placer les mises à jour en mémoire cache. La différence majeure réside dans le fait qu'un client multiniveau utilise un composant connexion pour établir un canal de communication avec le serveur d'applications.

Pour créer une application client multiniveau, démarrez un nouveau projet et effectuez les étapes suivantes :

1. Ajoutez un nouveau module de données au projet.
2. Placez un composant connexion sur le module de données. Le type de composant connexion que vous ajoutez dépend du protocole de communication que vous souhaitez utiliser. Voir "Structure de l'application client" à la page 25-5, pour plus de détails.
3. Initialisez les propriétés sur votre composant connexion pour spécifier le serveur d'applications avec lequel il doit établir une connexion. Pour plus d'informations sur la configuration du composant connexion, voir "Connexion au serveur d'applications" à la page 25-27.

4. Initialisez les autres propriétés du composant connexion selon les besoins de votre application. Par exemple, vous pouvez initialiser la propriété *ObjectBroker* pour permettre au composant connexion de choisir dynamiquement parmi plusieurs serveurs. Pour plus d'informations sur l'utilisation des composants connexion, voir "Gestion des connexions serveur" à la page 25-32.
5. Placez autant de composants *TClientDataSet* que nécessaire sur le module de données, et initialisez la propriété *RemoteServer* pour chaque composant avec le nom du composant connexion placé lors de l'étape 2. Pour une présentation détaillée des ensembles de données client, voir chapitre 23, "Utilisation d'ensembles de données client".
6. Initialisez la propriété *ProviderName* de chaque composant *TClientDataSet*. Si votre composant connexion est connecté au serveur d'applications lors de la conception, vous pouvez choisir un fournisseur disponible dans la liste déroulante de la propriété *ProviderName*.
7. Poursuivez comme si vous créiez une application de base de données quelconque. Les clients d'applications multiniveaux disposent de quelques fonctionnalités supplémentaires :
   - Votre application peut appeler directement le serveur d'applications. "Appel des interfaces serveur" à la page 25-33 décrit la procédure à suivre.
   - Vous pouvez utiliser les fonctionnalités des ensembles de données client qui permettent de gérer leur interaction avec les composants fournisseur. Ces fonctionnalités sont décrites dans "Utilisation d'un ensemble de données client avec un fournisseur" à la page 23-29.

Connexion au serveur d'applications

Pour établir et maintenir une connexion avec un serveur d'applications, une application client utilise un ou plusieurs composants connexion. Ces composants se trouvent sur la page DataSnap de la palette des composants.

Utilisez un composant connexion pour

- Identifier le protocole utilisé pour communiquer avec le serveur d'applications. Chaque type de composant connexion représente un protocole de communication différent. Voir "Sélection d'un protocole de connexion" à la page 25-10, pour plus de détails sur les avantages et les limites des protocoles disponibles.
- Indiquer comment localiser la machine serveur. Les détails d'identification de la machine serveur varient selon le protocole.
- Identifier le serveur d'applications sur la machine serveur.

  Si vous n'utilisez pas CORBA, identifiez le serveur à l'aide de la propriété *ServerName* ou *ServerGUID*. *ServerName* identifie le nom de base de la classe que vous spécifiez lorsque vous créez le module de données distant sur le serveur d'applications. Voir "Configuration du module de données distant" à la page 25-15, pour plus de détails sur la spécification de cette valeur sur le

serveur. Si le serveur est recensé ou installé sur la machine client, ou si le composant connexion est connecté à la machine serveur, vous pouvez initialiser la propriété *ServerName* lors de la conception à partir d'une liste déroulante dans l'inspecteur d'objets. *ServerGUID* spécifie le GUID de l'interface du module de données distant. Vous pouvez rechercher cette valeur à l'aide de l'éditeur de bibliothèque de types.

Si vous utilisez CORBA, identifiez le serveur à l'aide de la propriété *RepositoryID*. *RepositoryID* spécifie l'Id de référentiel de l'interface usine du serveur d'applications, qui apparaît comme troisième argument dans l'appel de *TCorbaVCLComponentFactory.Create*, automatiquement ajouté à la section d'initialisation de l'unité d'implémentation du serveur CORBA. Vous pouvez aussi initialiser cette propriété sur le nom de base de l'interface du module de données CORBA (la même chaîne que la propriété *ServerName* des autres composants connexion) pour qu'elle soit automatiquement convertie en l'Id de référentiel approprié.

- Gérer les connexions serveur. Les composants connexion peuvent être utilisés pour créer ou abandonner des connexions et pour appeler des interfaces de serveur d'applications.

Généralement, le serveur d'applications et l'application client se trouvent sur des machines différentes. Mais même si le serveur réside sur la même machine que l'application client (par exemple, pendant la construction et le test de toute l'application multiniveau), vous pouvez utiliser le composant connexion pour identifier le serveur d'applications par son nom, spécifier une machine serveur et utiliser l'interface du serveur d'applications.

Spécification d'une connexion à l'aide de DCOM

Lorsque vous utilisez DCOM pour communiquer avec le serveur d'applications, l'application cliente inclut un composant *TDCOMConnection* pour s'y connecter. *TDCOMConnection* utilise la propriété *ComputerName* pour identifier la machine sur laquelle réside le serveur.

Lorsque *ComputerName* est vierge, le composant connexion DCOM considère que le serveur d'applications réside sur la machine client ou qu'il possède une entrée de registre système. Lorsque vous utilisez DCOM et que le serveur réside sur une machine différente de celle du client, vous devez fournir *ComputerName* si vous ne fournissez pas d'entrée de registre système pour le serveur d'applications sur le client.

Remarque Même lorsqu'il existe une entrée de registre système pour le serveur d'applications, vous pouvez spécifier *ComputerName* pour écraser cette entrée. Cela peut être particulièrement utile pendant le développement, le test et le débogage.

Si votre application client peut choisir parmi plusieurs serveurs, vous pouvez utiliser la propriété *ObjectBroker* au lieu de spécifier la valeur de *ComputerName*. Pour plus d'informations, voir "Courtage de connexions" à la page 25-31.

Si vous fournissez le nom d'un ordinateur hôte ou d'un serveur introuvable, le composant connexion DCOM déclenche une exception lorsque vous essayez d'ouvrir la connexion.

Spécification d'une connexion à l'aide de sockets

Vous pouvez établir une connexion à un serveur d'applications à l'aide de sockets depuis n'importe quelle machine disposant d'une adresse TCP/IP. Cette méthode présente l'avantage de pouvoir s'appliquer à davantage de machines, mais ne permet pas l'utilisation des protocoles de sécurité. Lorsque vous utilisez des sockets, incluez un composant *TSocketConnection* pour la connexion au serveur d'applications.

TSocketConnection identifie la machine serveur à l'aide de l'adresse IP ou du nom d'hôte du système serveur, et du numéro de port du programme de répartition de sockets (Scktsrvr.exe) exécuté sur la machine serveur. Pour plus d'informations sur les adresses IP et les valeurs de port, voir "Description des sockets" à la page 32-3.

Trois propriétés de *TSocketConnection* spécifient ces informations :

- *Address* spécifie l'adresse IP du serveur.
- *Host* spécifie le nom d'hôte du serveur.
- *Port* spécifie le numéro de port du programme de répartition de sockets sur le serveur d'applications.

Address et *Host* s'excluent l'une l'autre. Initialiser la valeur de l'une désinitialise la valeur de l'autre. Pour plus d'informations sur la propriété à utiliser, voir "Description des hôtes" à la page 32-4.

Si votre application client peut choisir parmi plusieurs serveurs, vous pouvez utiliser la propriété *ObjectBroker* au lieu de spécifier une valeur pour *Address* ou *Host*. Pour plus d'informations, voir "Courtage de connexions" à la page 25-31.

Par défaut, la valeur de *Port* est 211, c'est le numéro de port par défaut des programmes de répartition de sockets fournis avec Delphi. Si le répartiteur de sockets a été configuré pour utiliser un port différent, initialisez la propriété *Port* en fonction de cette valeur.

Remarque Vous pouvez configurer le port du répartiteur de sockets en cliquant avec le bouton droit sur l'icône du serveur de socket Borland et en choisissant Propriétés.

Bien que les connexions socket ne permettent pas l'utilisation des protocoles de sécurité, vous pouvez personnaliser la connexion socket en ajoutant votre propre cryptage. Pour cela,

1. Créez un objet COM supportant l'interface *IDataIntercept*. C'est une interface de cryptage et de décryptage des données.
2. Utilisez *TPacketInterceptFactory* comme fabricant de classes de cet objet. Si vous utilisez un expert pour créer l'objet COM à l'étape 1, remplacez la ligne de la section d'initialisation indiquant `TComponentFactory.Create(...)` par `TPacketInterceptFactory.Create(...)`.

3 Recensez votre nouveau serveur COM sur la machine client.

4 Définissez la propriété *InterceptName* ou *InterceptGUID* du composant connexion socket pour spécifier cet objet COM. Si vous avez utilisé *TPacketInterceptFactory* à l'étape 2, votre serveur COM apparaît dans la liste déroulante de l'inspecteur d'objets pour la propriété *InterceptName*.

5 Enfin, cliquez avec le bouton droit sur l'icône du serveur de socket Borland, choisissez Propriétés et, dans la page des propriétés, affectez aux propriétés InterceptName et InterceptGUID l'identificateur ProgId ou GUID de l'intercepteur.

Ce mécanisme peut servir également à la compression et à la décompression des données.

Spécification d'une connexion à l'aide de HTTP

Vous pouvez établir une connexion à un serveur d'applications à l'aide de HTTP depuis n'importe quelle machine disposant d'une adresse TCP/IP. Au contraire des sockets, HTTP vous permet de bénéficier de la sécurité SSL et de communiquer avec un serveur protégé par un coupe-feu. Lorsque vous utilisez HTTP, incluez un composant *TWebConnection* pour la connexion au serveur d'applications.

Le composant connexion Web établit une connexion vers l'application serveur Web (httpsrvr.dll), qui à son tour communique avec le serveur d'applications. *TWebConnection* localise httpsrvr.dll en utilisant une URL (Uniform Resource Locator). L'URL spécifie le protocole (http ou, si vous utilisez la sécurité SSL, https), le nom d'hôte de la machine exécutant le serveur Web et httpsrvr.dll, ainsi que le chemin d'accès à l'application serveur Web (httpsrvr.dll). Spécifiez cette valeur avec la propriété *URL*.

Remarque Lorsque vous utilisez *TWebConnection*, wininet.dll doit être installé sur la machine client. Si vous avez IE3 ou une version supérieure, wininet.dll se trouve dans le répertoire système de Windows.

Si le serveur Web nécessite une authentification, ou si vous utilisez un serveur proxy qui demande une authentification, vous devez définir la valeur des propriétés *UserName* et *Password* pour que le composant connexion établisse la connexion.

Si votre application client peut choisir parmi plusieurs serveurs, vous pouvez utiliser la propriété *ObjectBroker* au lieu de spécifier la valeur de *URL*. Pour plus d'informations, voir "Courtage de connexions" à la page 25-31.

Spécification d'une connexion à l'aide de SOAP

Vous pouvez établir une connexion à un serveur d'applications SOAP en utilisant le composant *TSoapConnection*. *TSoapConnection* est très semblable à *TWebConnection*, car il utilise également HTTP comme protocole de transport. Ainsi, vous pouvez utiliser *TSoapConnection* depuis n'importe quelle machine ayant une adresse TCP/IP et elle profitera de la sécurité SSL pour communiquer avec un serveur protégé par un pare-feu.

Le composant connexion SOAP établit une connexion avec une application serveur Web implémentant l'interface *IAppServer* comme service Web. *TSoapConnection* localise cette application serveur Web en utilisant une URL (Uniform Resource Locator). L'URL spécifie le protocole (http ou, si vous utilisez la sécurité SSL, https), le nom d'hôte de la machine exécutant le serveur Web, le nom de l'application service Web et le chemin correspondant au nom de chemin du *THTTPSoapDispatcher* sur le serveur d'applications. Spécifiez cette valeur dans la propriété *URL*.

Remarque Lorsque vous utilisez *TSoapConnection*, wininet.dll doit être installée sur la machine client. Si vous avez installé IE3, ou une version supérieure, wininet.dll se trouve dans le répertoire système de Windows.

Si le serveur Web exige une authentification, ou si vous utilisez un serveur proxy nécessitant une authentification, vous devez définir les valeurs des propriétés *UserName* et *Password* pour permettre au composant connexion de se connecter.

Spécification d'une connexion à l'aide de CORBA

Toutefois, vous pouvez limiter le nombre de serveurs auxquels votre application client peut se connecter à l'aide des autres propriétés du composant connexion CORBA. Si vous souhaitez spécifier une machine serveur particulière plutôt que laisser CORBA Smart Agent en localiser une de disponible, utilisez la propriété *HostName*. Si plusieurs instances d'objets implémentent l'interface de votre serveur, vous pouvez spécifier l'objet que vous souhaitez utiliser en initialisant la propriété *ObjectName*.

Le composant *TCorbaConnection* obtient une interface vers le module de données CORBA sur le serveur d'applications de deux manières :

- Si vous utilisez la liaison anticipée (statique), vous devez ajouter le fichier _TLB.pas (généré par l'éditeur de bibliothèque de types) à votre application client. La liaison anticipée est fortement recommandée pour la vérification du type à la compilation et parce qu'elle est beaucoup plus rapide que la liaison tardive (dynamique).
- Si vous utilisez la liaison tardive (dynamique), l'interface doit être recensée dans le référentiel d'interfaces.

Pour une comparaison des liaisons anticipée et tardive, voir "Appel des interfaces serveur" à la page 25-33.

Courtage de connexions

Si votre application client peut choisir parmi plusieurs serveurs, vous pouvez utiliser un courtier d'objets pour localiser un système serveur disponible. Le courtier d'objets gère une liste de serveurs disponibles pour le composant connexion. Lorsque le composant connexion a besoin de se connecter à un serveur d'applications, il demande au courtier d'objets un nom d'ordinateur (ou une adresse IP, un nom d'hôte, une URL). Le courtier fournit un nom puis le composant connexion établit la connexion. Si le nom fourni ne fonctionne pas

(par exemple si le serveur n'est pas opérationnel), le courtier fournit un autre nom et répète l'opération jusqu'à ce que la connexion soit établie.

Une fois que le composant connexion a établi une connexion avec un nom fourni par le courtier, il enregistre ce nom en tant que valeur de la propriété appropriée (*ComputerName*, *Address*, *Host, RemoteHost* ou *URL*). Si le composant connexion ferme la connexion puis a besoin de l'ouvrir à nouveau, il utilise cette valeur de propriété et ne demande un nouveau nom au courtier que si la connexion échoue.

Pour utiliser un courtier d'objets, spécifiez la propriété *ObjectBroker* de votre composant connexion. Lorsque la propriété *ObjectBroker* est initialisée, le composant connexion n'enregistre pas la valeur de *ComputerName*, *Address*, *Host*, *RemoteHost* ou *URL*.

Remarque N'utilisez pas la propriété *ObjectBroker* avec des connexions CORBA. CORBA possède son propre service de courtage.

Gestion des connexions serveur

La fonction principale des composants connexion est de localiser le serveur d'applications et de s'y connecter. Comme ils gèrent les connexions serveur, ils vous permettent également d'appeler les méthodes de l'interface du serveur d'applications.

Connexion au serveur

Pour localiser le serveur d'applications et vous y connecter, vous devez d'abord initialiser les propriétés du composant connexion pour identifier le serveur d'applications. Ce processus est décrit dans "Connexion au serveur d'applications" à la page 25-27. Avant d'ouvrir la connexion, tous les ensembles de données client qui utilisent le composant connexion pour communiquer avec le serveur d'applications doivent l'indiquer en initialisant leur propriété *RemoteServer*.

La connexion est automatiquement ouverte lorsque les ensembles de données client essaient d'accéder au serveur d'applications. Par exemple, l'initialisation à *True* de la propriété *Active* de l'ensemble de données client ouvre la connexion, pour autant que la propriété *RemoteServer* ait été définie.

Si vous ne liez aucun ensemble de données client au composant connexion, vous pouvez ouvrir la connexion en initialisant à *True* la propriété *Connected* du composant connexion.

Avant d'établir une connexion à un serveur d'applications, un composant connexion génère un événement *BeforeConnect*. Vous pouvez exécuter des actions particulières avant de vous connecter en les codant dans un gestionnaire *BeforeConnect*. Après avoir établi une connexion, le composant connexion génère un événement *AfterConnect* où vous pouvez également exécuter toute action nécessaire.

Fermeture ou changement de connexion serveur

Un composant connexion ferme une connexion à un serveur d'applications dans les circonstances suivantes :

- Quand vous initialisez la propriété *Connected* à *False*.
- Quand vous libérez le composant connexion. Un objet connexion est libéré automatiquement quand un utilisateur ferme l'application client.
- Quand vous modifiez les propriétés qui identifient le serveur d'applications (*ServerName*, *ServerGUID*, *ComputerName*, etc.). La modification de ces propriétés vous permet de basculer entre les serveurs d'applications disponibles lors de l'exécution. Le composant connexion ferme la connexion en cours et en établit une nouvelle.

Remarque Au lieu d'utiliser un seul composant connexion pour basculer entre les serveurs d'applications disponibles, une application client peut disposer de plusieurs composants connexion, chacun d'eux étant connecté à un serveur d'applications particulier.

Avant de fermer une connexion, un composant connexion appelle automatiquement son gestionnaire d'événement *BeforeDisconnect*, s'il existe. Pour exécuter des actions particulières avant la déconnexion, écrivez un gestionnaire *BeforeDisconnect*. De même, après la fermeture de la connexion, le gestionnaire d'événement *AfterDisconnect* est appelé. Pour exécuter des actions particulières après la déconnexion, écrivez un gestionnaire *AfterDisconnect*.

Appel des interfaces serveur

Les applications n'ont pas besoin d'appeler l'interface *IAppServer* directement, car les appels appropriés sont automatiquement réalisés lorsque vous utilisez les propriétés et méthodes de l'ensemble de données client. Cependant, alors qu'il n'est pas nécessaire de travailler directement avec l'interface *IAppServer*, vous pouvez avoir ajouté vos propres extensions à l'interface du module de données distant. Lorsque vous étendez l'interface du serveur d'applications, vous devez être en mesure d'appeler ces extensions à l'aide de la connexion créée par votre composant connexion. A moins que vous utilisez SOAP, vous pouvez utiliser à cet effet la propriété *AppServer* du composant connexion. Pour plus d'informations sur l'extension de l'interface du serveur d'applications, voir "Extension de l'interface du serveur d'applications" à la page 25-19.

AppServer est un Variant qui représente l'interface du serveur d'applications. Vous pouvez appeler une méthode d'interface à l'aide de *AppServer* en écrivant une instruction telle que

```
MyConnection.AppServer.SpecialMethod(x,y);
```

Toutefois, cette technique offre une liaison tardive (dynamique) de l'appel d'interface. En raison de cela, l'appel de la procédure *SpecialMethod* n'est pas liée avant l'exécution lorsque l'appel est exécuté. La liaison tardive est très souple mais son utilisation vous prive de nombreux avantages tels que code insight et la vérification de type. De plus, la liaison tardive est plus lente que la liaison

anticipée car le compilateur génère des appels supplémentaires vers le serveur pour configurer les appels d'interface avant de les réaliser.

Lorsque vous utilisez DCOM ou CORBA comme protocole de communication, vous pouvez utiliser la liaison anticipée d'appels *AppServer*. Utilisez l'opérateur **as** pour affecter à la variable *AppServer* le descendant *IAppServer* que vous avez créé lorsque vous avez créé le module de données distant. Par exemple :

```
with MyConnection.AppServer as IMyAppServer do
  SpecialMethod(x,y);
```

Pour utiliser la liaison anticipée sous DCOM, la bibliothèque de types du serveur doit être recensée sur la machine client. Vous pouvez utiliser TRegsvr.exe, fourni avec Delphi, pour recenser la bibliothèque de types.

Remarque Reportez-vous à la démo TRegSvr (qui offre le source de TRegsvr.exe) pour un exemple de recensement de la bibliothèque de types par programmation.

Pour utiliser la liaison anticipée avec CORBA, vous devez ajouter à votre projet l'unité _TLB générée par l'éditeur de bibliothèque de types. Pour ce faire, ajoutez cette unité à la clause **uses** de votre unité.

Lorsque vous utilisez TCP/IP ou HTTP, vous ne pouvez pas réellement utiliser la liaison anticipée mais, comme le module de données distant utilise une interface double, vous pouvez utiliser la dispinterface du serveur d'applications pour améliorer les performances d'une simple liaison tardive. La dispinterface porte le même nom que l'interface du module de données distant, suivi de la chaîne 'Disp'. Vous pouvez affecter à la propriété *AppServer* une variable de ce type pour obtenir la dispinterface. Par exemple :

```
var
  TempInterface: IMyAppServerDisp;
begin
  TempInterface :=IMyAppServerDisp(IDispatch(MyConnection.AppServer));
...
  TempInterface.SpecialMethod(x,y);
...
end;
```

Remarque Pour utiliser la dispinterface, vous devez ajouter l'unité _TLB générée lorsque vous enregistrez la bibliothèque de types à la clause **uses** de votre module client.

Si vous utilisez SOAP, vous ne pouvez pas utiliser la propriété *AppServer*. A la place, vous devez utiliser un objet à interface distante (*THTTPRio*) et effectuer des appels à liaison anticipée. Comme dans tous les appels à liaison anticipée, l'application client doit connaître la déclaration de l'interface du serveur d'applications à la compilation. Pour l'ajouter à votre application client, vous pouvez soit ajouter l'unité utilisée par le serveur pour déclarer et recenser l'interface à la clause uses du client, soit faire référence à un document WSDL décrivant l'interface. Pour plus d'informations sur l'importation d'un document WSDL décrivant l'interface du serveur, voir "Importation de documents WSDL" à la page 31-9.

Remarque L'unité qui déclare l'interface du serveur doit aussi la recenser dans le registre d'invocation. Pour savoir comment recenser des interfaces invocables, voir "Définition des interfaces invocables" à la page 31-4.

Une fois que votre application utilise l'unité du serveur qui déclare et recense l'interface, ou que vous avez importé un document WSDL pour générer une telle unité, créez une instance de *THTTPRio* pour l'interface souhaitée :

```
X := THTTPRio.Create(nil);
```

Ensuite, assignez l'URL utilisée par votre composant connexion à l'objet à interface distante :

```
X.URL := SoapConnection1.URL;
```

Vous pouvez alors utiliser l'opérateur **as** pour convertir l'instance de *THTTPRio* en l'interface du serveur d'applications :

```
InterfaceVariable := X as IMyAppServer;
InterfaceVariable.SpecialMethod(x,y);
```

Connexion à un serveur d'applications qui utilise plusieurs modules de données

Si le serveur d'applications utilise un module de données distant "parent" principal et plusieurs modules de données distants enfant, comme décrit dans "Utilisation de plusieurs modules de données distants" à la page 25-24, vous devez disposer d'un composant connexion séparé pour chaque module de données distant sur le serveur d'applications. Chaque composant connexion représente la connexion à un module de données distant unique.

Bien qu'il soit possible pour votre application client d'établir des connexions indépendantes à chaque module de données distant sur le serveur d'applications, il est plus efficace d'utiliser une connexion unique au serveur d'applications qui soit partagée par tous les composants connexion. Pour ce faire, vous ajoutez un composant connexion unique qui se connecte au module de données distant "principal" sur le serveur d'applications puis, pour chaque module de données distant "enfant", vous ajoutez un composant qui partage la connexion au module de données distant principal.

1. Pour la connexion au module de données distant principal, ajoutez et configurez un composant connexion tel que décrit dans "Connexion au serveur d'applications" à la page 25-27. La seule limite est que vous ne pouvez pa utiliser une connexion CORBA ou SOAP.
2. Pour chaque module de données distant enfant, utilisez un composant *TSharedConnection*.
   - Affectez à sa propriété *ParentConnection* le composant connexion ajouté à l'étape 1. Le composant *TSharedConnection* partage la connexion établie par cette connexion principale.

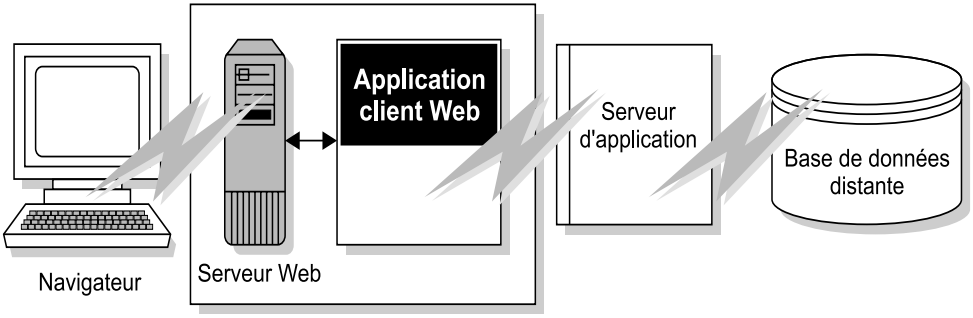
- Affectez à sa propriété *ChildName* le nom de la propriété sur l'interface du module de données distant principal qui expose l'interface du module de données distant enfant désiré.

Lorsque vous affectez le composant *TSharedConnection* placé à l'étape 2 comme valeur de la propriété *RemoteServer* d'un ensemble de données client, il fonctionne comme si vous utilisiez une connexion entièrement indépendante au module de données distant enfant. Toutefois, le composant *TSharedConnection* utilise la connexion établie par le composant placé à l'étape 1.

Ecriture des applications client Web

Si vous voulez créer des clients basés sur le Web pour votre application de bases de données multiniveau, vous devez remplacer le niveau client par une application Web spéciale qui agit simultanément en tant que client pour un serveur d'applications et en tant qu'application serveur Web installée sur la même machine que le serveur Web. Cette architecture est illustrée par la figure suivante.

Figure 25.1 Application de bases de données multiniveau basée sur le Web



Il existe deux façons d'envisager la construction de l'application Web :

- Vous pouvez combiner l'architecture de base de données multiniveau au support d'ActiveX fourni par Delphi pour distribuer l'application client en tant que contrôle ActiveX. Cela permet à n'importe quel navigateur supportant ActiveX d'exécuter votre application client en tant que serveur en processus.
- Vous pouvez utiliser les paquets de données XML pour construire une application InternetExpress. Cela permet aux navigateurs supportant javascript d'interagir avec votre application client par le biais de pages html.

Ces deux approches sont très différentes. Votre choix dépendra des considérations suivants :

- Chaque approche repose sur une technologie différente (ActiveX contre javascript et XML). Prenez en compte les systèmes de vos utilisateurs finals. La première approche exige un navigateur supportant ActiveX (ce qui limite vos clients aux plates-formes Windows). La seconde exige un navigateur

supportant javascript et les capacités DHTML introduites par Netscape 4 et Internet Explorer 4.

- Les contrôles ActiveX doivent être téléchargés sur le navigateur pour agir en tant que serveur en processus. Il s'en suit que les clients utilisant l'approche ActiveX exigent beaucoup plus de mémoire que les clients d'une application basée sur HTML.
- L'approche InternetExpress peut s'intégrer à d'autres pages HTML. Un client ActiveX doit être exécuté dans une fenêtre à part.
- L'approche InternetExpress utilise HTTP standard, excluant donc les problèmes de coupe-feu auxquels sont confrontées les applications ActiveX.
- L'approche ActiveX autorise une plus grande souplesse dans la façon de programmer votre application. Vous n'êtes pas limité par les capacités des bibliothèques javascript. Les ensembles de données client utilisés dans l'approche ActiveX exposent plus de fonctionnalités (filtres, fourchettes, globalisation, paramètres facultatifs, lecture différée des BLOB ou des détails imbriquées, etc.) que les navigateurs XML utilisés par l'approche InternetExpress.

Attention Votre application client Web risque d'apparaître et de se comporter de manière différente selon le navigateur qui l'affiche. Testez-la avec les navigateurs dont se serviront vos utilisateurs finals.

Distribution d'une application client en tant que contrôle ActiveX

L'architecture de base de données multiniveau peut être combinée avec les fonctionnalités ActiveX de Delphi pour distribuer une application client en tant que contrôle ActiveX.

Lorsque vous distribuez votre application client MIDAS en tant que contrôle ActiveX, créez le serveur d'applications comme dans toute autre application multiniveau. La seule contrainte est que vous serez amené à utiliser DCOM, HTTP, SOAP ou sockets comme protocole de communication, car le logiciel runtime CORBA n'est pas installé sur les machines client. Pour plus de détails sur la création du serveur d'applications, voir "Création du serveur d'applications" à la page 25-13.

Lorsque vous créez l'application client, vous devez utiliser une fiche active comme base au lieu d'une fiche ordinaire. Voir "Création d'une fiche active pour l'application client", pour plus de détails.

Une fois que vous avez construit et déployé votre application client, celle-ci est accessible depuis n'importe quelle machine munie d'un navigateur Web ActiveX. Pour qu'un navigateur Web lance votre application client, le serveur Web doit être exécuté sur la machine qui héberge l'application client.

Si l'application client utilise DCOM pour communiquer avec le serveur d'applications, la machine qui héberge le navigateur Web doit être activée pour fonctionner avec DCOM. Si la machine hébergeant le navigateur Web est une machine Windows 95, DCOM95, disponible auprès de Microsoft, doit y être installé.

Création d'une fiche active pour l'application client

1. Comme l'application client doit être déployée en tant que contrôle ActiveX, un serveur Web doit être exécuté sur le même système que l'application client. Vous pouvez utiliser un serveur immédiatement opérationnel, tel que Personal Web Server de Microsoft, ou écrire le vôtre à l'aide des composants socket décrits dans le chapitre 32, "Utilisation des sockets".
2. Créez l'application client en suivant la procédure décrite dans "Création de l'application client" à la page 25-26, à la différence toutefois que vous devez commencer en choisissant Fichier | Nouveau | Fiche active, plutôt qu'en démarrant le projet client comme un projet Delphi ordinaire.
3. Si votre application client utilise un module de données, ajoutez un appel pour créer de façon explicite le module de données dans l'initialisation de la fiche active.
4. Lorsque votre application client est achevée, compilez le projet et sélectionnez Projet | Options de déploiement Web. Dans la boîte de dialogue Options de déploiement Web, vous devez procéder comme suit :
   1. Sur la page Projet, spécifiez le répertoire destination, l'URL du répertoire cible et le répertoire HTML. Habituellement, le répertoire destination et le répertoire HTML sont les mêmes que les répertoires de projets de votre serveur Web. L'URL cible est habituellement le nom de la machine serveur spécifiée dans les paramètres Windows Network | DNS.
   2. Sur la page Fichiers supplémentaires, incluez midas.dll dans votre application client.
5. Enfin, sélectionnez Projet | Déployer pour le Web, pour déployer l'application client en tant que fiche active.

N'importe quel navigateur Web pouvant exécuter des fiches actives peut exécuter votre application client. Il suffit que soit spécifié le fichier .HTM créé lorsque vous avez déployé l'application client. Ce fichier .HTM porte le même nom que votre projet d'application client et apparaît dans le répertoire spécifié comme répertoire destination.

Construction des applications Web avec InternetExpress

Une application client peut exiger du serveur d'applications la fourniture de données codées en XML à la place d'OleVariants. En combinant les paquets de données codées XML, bibliothèques javascript spéciales de fonctions pour bases de données, et le support des applications serveur Web de Delphi, vous pouvez créer des applications client simples accessibles via un navigateur Web supportant javascript. Ces applications constituent le support InternetExpress de Delphi.

Avant d'entreprendre la construction d'une application InternetExpress, vous devez comprendre l'architecture des applications serveur Web de Delphi. Elle est décrite dans le chapitre 27, "Création d'applications Internet".

Une application InternetExpress étend l'architecture d'application serveur Web de base pour faire office de client d'un serveur d'applications. Les applications InternetExpress génèrent des pages HTML qui associent HTML, XML et javascript. HTML régit la disposition et l'aspect des pages vues par les utilisateurs finals dans leur navigateur. XML code les paquets de données et les paquets delta qui représentent les informations base de données. Javascript permet aux contrôles HTML d'interpréter et de manipuler les données des paquets XML sur la machine client.

Si l'application InternetExpress utilise DCOM pour se connecter au serveur d'applications, vous devez suivre des étapes supplémentaires afin de garantir que le serveur d'applications accorde les droits d'accès et de lancement à ses clients. Voir "Droits d'accès au serveur d'applications et à son lancement" à la page 25-41, pour plus de détails.

Astuce Vous pouvez créer une application InternetExpress pour fournir des navigateurs Web avec des données "réelles" même si vous n'avez pas d'application serveur. Ajoutez simplement un fournisseur et son ensemble de données au module Web.

Construction d'une application InternetExpress

Les étapes suivantes décrivent une façon de construire une application Web avec InternetExpress. Il en résulte une application qui crée des pages HTML pour permettre aux utilisateurs d'interagir sur les données depuis un serveur d'applications via un navigateur Web javascript. Vous pouvez également construire une application InternetExpress à l'aide de l'architecture Site Express et du producteur de page d'InternetExpress (*TInetXPageProducer*).

1. Choisissez Fichier|Nouveau pour afficher la boîte de dialogue Nouveaux éléments. Dans la page Nouveau, sélectionnez Application serveur Web. Cette procédure est décrite par "Création d'applications serveur Web avec l'agent Web" à la page 28-1.
2. Depuis la page DataSnap de la palette de composants, ajoutez un composant connexion au module Web qui apparaît lorsque vous créez une nouvelle application serveur Web. Le type du composant connexion dépendra du protocole de communication utilisé. Voir "Sélection d'un protocole de connexion" à la page 25-10, pour plus de détails.
3. Définissez les propriétés de votre composant connexion pour spécifier le serveur d'applications avec lequel il doit établir la connexion. Pour en savoir plus sur la définition d'un composant connexion, voir "Connexion au serveur d'applications" à la page 25-27.
4. Depuis la page InternetExpress de la palette de composants, ajoutez au module Web un courtierXML au lieu d'un ensemble de données client. Comme *TClientDataSet*, *TXMLBroker* représente les données venant d'un fournisseur sur le serveur d'applications et interagit avec le serveur d'applications via son interface *IAppServer*. Cependant, au contraire des ensembles de données, les courtiers XML demandent des paquets de données XML et non OleVariants, et interagissent avec les composants InternetExpress et non les contrôles de données.

5 Définissez la propriété *RemoteServer* du courtier XML pour qu'elle pointe sur le composant connexion ajouté à l'étape 2. Définissez la propriété *ProviderName* pour qu'elle indique le fournisseur sur le serveur d'applications qui fournit les données et applique les mises à jour. Pour plus d'informations sur la configuration d'un courtier XML, voir "Utilisation d'un courtier XML" à la page 25-42.

6 Ajoutez au module Web un producteur de page InternetExpress (*TInetXPageProducer*) pour chaque page qui sera vue par les utilisateurs dans leur navigateur. Pour chaque producteur de page, vous devez définir une propriété *IncludePathURL*. Elle indique où se trouvent les bibliothèques javascript. Ces bibliothèques ajoutent aux contrôles HTML des capacités de gestion des données.

7 Cliquez avec le bouton droit sur une page Web et choisissez Editeur d'actions pour afficher l'éditeur d'actions. Ajoutez l'élément d'action correspondant à chaque message que vous voulez gérer depuis les navigateurs. Associez les producteurs de page que vous avez ajouté à l'étape 6 avec ces actions en définissant leur propriété *Producer* ou en écrivant du code dans un gestionnaire de l'événement *OnAction*. Pour plus d'informations sur l'ajout des éléments d'action dans l'éditeur d'actions, voir "Ajout d'actions au répartiteur" à la page 28-5.

8 Double-cliquez sur chaque page Web pour afficher l'éditeur de pages Web (vous pouvez également afficher cet éditeur en cliquant dans l'inspecteur d'objets sur le bouton points de suspension situé à côté de la propriété *WebPageItems*). Dans l'éditeur, vous pouvez ajouter des éléments Web pour concevoir les pages que verront les utilisateurs dans leur navigateur. Pour plus d'informations sur la conception de pages Web pour votre application InternetExpress, voir "Création des pages Web avec un producteur de page InternetExpress" à la page 25-44.

9 Construisez votre application Web. Une fois l'application installée avec votre serveur Web, les navigateurs pourront l'appeler en spécifiant le nom de l'application dans la portion de l'URL correspondant au nom de script et le nom du composant page Web dans la partie correspondant au chemin d'accès.

Utilisation des bibliothèques javascript

Les pages HTML générées par les composants InternetExpress et les éléments Web qu'elles contiennent utilisent plusieurs bibliothèques javascript livrées avec Delphi :

Tableau 25.3 Les bibliothèques javascript

Bibliothèque	fonctions
xmldom.js	Cette bibliothèque est un analyseur XML compatible DOM écrit en javascript. Il permet aux analyseurs ne supportant pas XML d'utiliser les paquets de données XML.
xmldb.js	Cette bibliothèque définit des classes d'accès aux données analogues à TClientDataSet et TField.

Tableau 25.3 Les bibliothèques javascript (suite)

Bibliothèque	fonctions
xmldisp.js	Cette bibliothèque définit des classes associant les classes d'accès aux données de xmldb aux contrôles HTML de la page HTML.
xmlerrdisp.js	Cette bibliothèque définit des classes qui peuvent être utilisées lors de la réconciliation des erreurs de mise à jour. Ces classes ne sont utilisées par aucun des composants InternetExpress intégrés, mais sont utiles pour écrire un générateur Reconcile.
xmlshow.js	Cette bibliothèque inclut des fonctions pour afficher des paquets de données formatés XML et des paquets delta XML. Cette bibliothèque n'est utilisée par aucun des composants InternetExpress intégrés, mais est utile pour le débogage.

Ces bibliothèques sont dans le répertoire Source/Webmidas. Lorsque vous les aurez installées, vous devrez définir la propriété *IncludePathURL* de tous les producteurs de page InternetExpress pour indiquer où elles se trouvent.

Il est possible d'écrire vos propres pages HTML en utilisant les classes javascript fournies dans ces bibliothèques au lieu d'utiliser les éléments Web pour générer vos pages Web. Toutefois, vous devez vérifier que votre code ne fait rien d'illégal car ces classes comportent une vérification d'erreurs minimale (afin de réduire la taille des pages Web générées).

Les classes des bibliothèques javascript sont un standard en cours de développement, elles sont régulièrement mises à jour. Si vous voulez les utiliser directement plutôt que de laisser les éléments Web générer le code javascript, vous pouvez obtenir leurs dernières versions et une documentation sur leur utilisation à partir de CodeCentral disponible via community.borland.com.

Droits d'accès au serveur d'applications et à son lancement

Les demandes issues de l'application InternetExpress apparaissent au serveur d'applications comme provenant d'un compte invité dont le nom est IUSR_computername, où computername est le nom du système qui exécute l'application Web. Par défaut, ce compte n'a pas le droit d'accéder au serveur d'applications ni de le lancer. Si vous essayez d'utiliser l'application Web sans accorder ces droits, lorsque le navigateur Web tente de charger la page demandée, un dépassement de délai se produit avec l'erreur `EOLE_ACCESS_ERROR`.

Remarque Le serveur d'applications s'exécutant sous le compte invité, il ne peut être arrêté par aucun autre compte.

Pour accorder à l'application Web le droit d'accéder au serveur d'applications et de le lancer, exécutez DCOMCnfg.exe, qui se trouve dans le répertoire System32 de la machine exécutant le serveur d'applications. Les étapes suivantes décrivent comment configurer votre serveur d'applications :

1. Lorsque vous exécutez DCOMCnfg, sélectionnez votre serveur dans la liste des applications de la page Applications.
2. Cliquez sur le bouton Propriétés. Lorsque le dialogue change, sélectionnez la page Sécurité.

3 Sélectionnez Permissions d'accès personnalisées, et appuyez sur le bouton Modifier. Ajoutez le nom IUSR_computername à la liste de comptes ayant un droit d'accès, où computername est le nom de la machine exécutant l'application Web.

4 Sélectionnez Permissions de lancement personnalisées, et appuyez sur le bouton Modifier. Ajoutez IUSR_computername à la liste.

5 Cliquez sur le bouton Appliquer.

Utilisation d'un courtier XML

Un courtier XML a deux fonctions principales :

- Il lit à partir du serveur d'applications les paquets de données XML et les rend disponibles aux éléments Web générant le HTML pour l'application InternetExpress.
- Il reçoit des navigateurs les mises à jour au format des paquets delta XML et les applique au serveur d'applications.

Lecture des paquets de données XML

Avant que le courtier XML fournisse les paquets de données XML aux composants générant les pages HTML, il doit les lire depuis le serveur d'applications. Pour cela, il utilise l'interface *IAppServer* du serveur d'applications, qu'il acquiert via un composant connexion. Vous devez définir les propriétés suivantes pour que le producteur XML utilise l'interface *IAppServer* du serveur d'applications :

- Définissez la propriété *RemoteServer* par le composant connexion qui établit la connexion au serveur d'applications et obtient son interface *IAppServer*. Au moment de la conception, vous pouvez sélectionner cette valeur dans l'inspecteur d'objets à partir d'une liste déroulante.
- Définissez la propriété *ProviderName* par le nom du composant fournisseur sur le serveur d'applications qui représente l'ensemble de données pour lequel vous voulez les paquets XML. Ce fournisseur fournit les paquets de données XML et applique les mises à jour à partir des paquets delta XML. Au moment de la conception, si la propriété *RemoteServer* est définie et si le composant connexion a une connexion active, l'inspecteur d'objets affiche la liste des fournisseurs disponibles. (Si vous utilisez une connexion DCOM, le serveur d'applications doit être recensé sur la machine client).

Deux propriétés vous permettent d'indiquer les informations à inclure dans les paquets de données.

- Vous pouvez limiter le nombre d'enregistrements ajoutés au paquet de données en définissant la propriété *MaxRecords*. Cela est particulièrement important pour les gros ensembles de données car les applications InternetExpress envoient tout le paquet de données aux navigateurs Web client. Si le paquet de données est trop gros, le temps de téléchargement peut devenir excessivement long.

- Si le fournisseur sur le serveur d'applications représente une requête ou une procédure stockée, vous pouvez souhaiter transmettre les valeurs des paramètres avant d'obtenir le paquet de données XML. Vous pouvez fournir ces valeurs à l'aide de la propriété *Params*.

Les composants qui génèrent le code HTML et javascript pour l'application InternetExpress utilisent automatiquement le paquet de données XML du courtier XML lorsque vous avez défini leur propriété *XMLBroker*. Pour obtenir le paquet de données XML directement dans le code, utilisez la méthode *RequestRecords*.

Remarque Quand le courtier XML fournit un paquet de données à un autre composant (ou que vous appelez *RequestRecords*), il reçoit un événement *OnRequestRecords*. Vous pouvez utiliser cet événement pour fournir votre propre chaîne XML au lieu du paquet de données du serveur d'applications. Par exemple, vous pouvez lire le paquet de données XML à partir du serveur d'applications à l'aide de *GetXMLRecords*, puis le modifier avant de le fournir à la page Web.

Application des mises à jour à partir des paquets delta XML

Lorsque vous ajoutez le courtier XML au module Web (ou à un module de données contenant un *TWebDispatcher*), il se recense lui-même automatiquement avec le répartiteur Web en tant qu'objet auto-réparti. Cela veut dire qu'au contraire des autres composants, il n'est pas nécessaire de créer un élément d'action pour le courtier XML afin qu'il réponde aux messages de mise à jour provenant du navigateur Web. Ces messages contiennent les paquets delta XML à appliquer au serveur d'applications. Habituellement, ils proviennent d'un bouton que vous avez créé dans une des pages HTML produites par l'application client Web.

Afin que le répartiteur reconnaisse les messages pour le courtier XML, vous devez les décrire à l'aide de la propriété *WebDispatch*. Définissez la propriété *PathInfo* par la partie chemin d'accès de l'URL à laquelle sont envoyés les messages du courtier XML. Définissez *MethodType* par la valeur d'en-tête de la méthode des messages de mise à jour adressés à cette URL (classiquement *mtPost*). Si vous souhaitez répondre à tous les messages avec le chemin d'accès spécifié, définissez *MethodType* par *mtAny*. Si vous ne voulez pas que le courtier XML réponde directement aux messages de mise à jour (par exemple, si vous souhaitez les gérer explicitement en utilisant un élément d'action), définissez la propriété *Enabled* par *False*. Pour plus d'informations sur la façon dont le répartiteur Web détermine les composants gérant les messages du navigateur Web, voir "Répartition des messages de requête" à la page 28-5.

Lorsque le répartiteur passe un message de mise à jour au courtier XML, il passe les mises à jour sur le serveur d'applications et il reçoit éventuellement un paquet delta XML décrivant toutes les erreurs de mise à jour qui se sont produites. Finalement, il envoie un message de réponse au navigateur, qui redirige le navigateur sur la page ayant généré le paquet delta XML delta ou lui envoie un nouveau contenu.

Certains événements vous permettent d'intégrer des traitements personnalisés au niveau de chacune des étapes de ce processus de mise à jour :

1. La première fois que le répartiteur passe le message de mise à jour au courtier XML, il reçoit un événement *BeforeDispatch*, où vous pouvez pré-traiter la demande ou même la gérer entièrement. Cet événement permet au courtier XML de gérer d'autres messages que les messages de mise à jour.
2. Si le gestionnaire de l'événement *BeforeDispatch* ne gère pas le message, le courtier XML reçoit un événement *OnRequestUpdate*, où vous pouvez appliquer les mises à jour vous-même plutôt que de suivre le processus par défaut.
3. Si le gestionnaire de l'événement *OnRequestUpdate* ne gère pas la demande, le courtier XML applique les mises à jour et reçoit un paquet delta contenant les erreurs s'y rapportant.
4. S'il n'y a pas d'erreur de mise à jour, le courtier XML reçoit un événement *OnGetResponse*, où vous pouvez créer un message de réponse indiquant que les mises à jour ont été appliquées avec succès ou envoyer des données rafraîchies au navigateur. Si le gestionnaire de l'événement *OnGetResponse* n'achève pas la réponse (ne définit pas le paramètre *Handled* par *True*), le courtier XML envoie une réponse qui redirige le navigateur sur le document ayant généré le paquet delta.
5. S'il y a des erreurs de mise à jour, le courtier XML reçoit un événement *OnGetErrorResponse*. Vous pouvez utiliser cet événement pour tenter de résoudre les erreurs ou pour générer une page Web qui les décrit à l'utilisateur final. Si le gestionnaire de l'événement *OnGetErrorResponse* n'achève pas la réponse (ne définit pas le paramètre *Handled* par *True*), le courtier XML appelle un générateur de contenu particulier, appelé le *ReconcileProducer*, pour générer le contenu du message de réponse.
6. Enfin, le courtier XML reçoit un événement *AfterDispatch*, où vous pouvez effectuer toutes les actions voulues avant de renvoyer une réponse au navigateur Web.

Création des pages Web avec un producteur de page InternetExpress

Chaque producteur de page InternetExpress génère un document HTML qui apparaît dans les navigateurs des clients de votre application. Si votre application comprend plusieurs documents Web séparés, utilisez un producteur de page différent pour chacun d'entre eux.

Le producteur de page InternetExpress (*TInetXPageProducer*) est un composant producteur de page spécial. Comme les autres producteurs de page, vous pouvez l'assigner à la propriété *Producer* d'un élément d'action ou l'appeler explicitement à partir du gestionnaire de l'événement *OnAction*. Pour plus d'informations sur l'utilisation des producteurs de contenu avec les éléments d'action, voir "Réponse aux messages de requête avec des éléments d'action" à la page 28-8.

Pour plus d'informations sur les producteurs de page, voir "Utilisation du composant générateur de page" à la page 28-14.

Contrairement à la majorité des producteurs de page, le producteur de page InternetExpress dispose d'un modèle par défaut, valeur de sa propriété *HTMLDoc*. Ce modèle contient un jeu de balises transparentes pour HTML que le producteur de page InternetExpress utilise pour assembler le document HTML (avec javascript et XML imbriqués) en incluant le contenu produit par les autres composants. Avant qu'il soit possible de traduire toutes les balises transparentes pour HTML et d'assembler le document, vous devez indiquer l'emplacement des bibliothèques javascript utilisées par le code javascript imbriqué dans la page. Cet emplacement est spécifié en définissant la propriété *IncludePathURL*.

Vous pouvez spécifier les composants générant chaque partie de la page Web avec l'éditeur de pages Web. Affichez l'éditeur de pages Web en double-cliquant sur le composant page Web ou, dans l'inspecteur d'objets, en cliquant sur le bouton points de suspension situé à côté de la propriété *WebPageItems*.

Les composants que vous ajoutez dans l'éditeur de pages Web génèrent le code HTML qui remplace une des balises transparentes pour HTML du modèle par défaut du producteur de page InternetExpress. Ces composants constituent la valeur de la propriété *WebPageItems*. Après avoir inséré les composants dans l'ordre qui vous convient, vous pouvez personnaliser le modèle pour ajouter votre propre code HTML ou modifier les balises par défaut.

Utilisation de l'éditeur de pages Web

L'éditeur de pages Web vous permet d'ajouter des éléments Web à votre producteur de page InternetExpress et de voir la page HTML qui en résulte. Affichez l'éditeur de pages Web en double-cliquant sur un composant producteur de page InternetExpress.

Remarque Vous devez avoir installé Internet Explorer 4, ou une version supérieure, pour utiliser l'éditeur de pages Web.

Le haut de l'éditeur de pages Web affiche les éléments Web qui génèrent le document HTML. Ces éléments Web sont imbriqués : chaque type d'élément assemble le HTML généré par ses sous-éléments. Différents types d'éléments peuvent contenir différents sous-éléments. A gauche, une arborescence affiche tous les éléments Web en indiquant la façon dont ils s'imbriquent. A droite, vous pouvez voir les éléments Web inclus dans l'élément en cours de sélection. Lorsque vous sélectionnez un composant en haut de l'éditeur de pages Web, vous pouvez définir ses propriétés dans l'inspecteur d'objets.

Cliquez sur le bouton Nouveau pour ajouter un sous-élément à l'élément en cours de sélection. Le dialogue Ajout de composant Web montre uniquement les éléments pouvant être ajoutés à l'élément sélectionné.

Le producteur de page InternetExpress peut contenir un des deux types d'élément suivants, chacun d'entre eux générant une fiche HTML :

- *TDataForm* génère une fiche HTML pour afficher les données et les contrôles qui manipulent les données ou soumettent les mises à jour.

Les éléments que vous ajoutez à *TDataForm* affichent les données dans une grille multi-enregistrement (*TDataGrid*) ou dans un jeu de contrôles, chacun représentant un seul champ et un seul enregistrement (*TFieldGroup*). En outre, vous pouvez ajouter un jeu de boutons pour naviguer dans les données ou pour poster les mises à jour (*TDataNavigator*). Vous pouvez également ajouter un bouton qui applique les mises à jour en retour sur le client Web (*TApplyUpdatesButton*). Chacun de ces éléments contient des sous-éléments qui représentent une valeur ou un bouton individuel. Enfin, avec la plupart des éléments Web, vous pouvez ajouter une grille de disposition (*TLayoutGroup*) permettant de personnaliser la disposition des éléments qu'ils contiennent.

- *TQueryForm* génère une fiche HTML pour afficher ou lire les valeurs définies par l'application. Par exemple, vous pouvez utiliser cette fiche pour afficher et soumettre des valeurs de paramètres.

  Les éléments que vous ajoutez à *TQueryForm* affichent les valeurs définies par l'application (*TQueryFieldGroup*). Ils peuvent également constituer un jeu de boutons soumettant ou réinitialisant ces valeurs (*TQueryButtons*). Chacun de ces éléments contient des sous-éléments qui représentent une valeur ou un bouton individuel. Vous pouvez également ajouter une grille de disposition à une fiche de requête comme vous le faites à une fiche de données.

La partie inférieure de l'éditeur de pages Web affiche le code HTML généré et vous permet de voir ce qu'il donne dans un navigateur (IE4).

Définition des propriétés des éléments Web

Les éléments Web ajoutés avec l'éditeur de pages Web sont des composants spécialisés qui génèrent le HTML. Chaque classe d'élément Web a été conçue pour produire un contrôle spécial ou une section du document HTML final. Un ensemble commun de propriétés influence l'aspect du document HTML final.

Lorsqu'un élément Web représente des informations provenant du paquet de données XML (par exemple, lorsqu'il génère un ensemble de contrôles affichant des champs ou des paramètres, ou un bouton de manipulation des données), la propriété *XMLBroker* associe l'élément Web au courtier XML gérant le paquet de données. Vous pouvez ensuite spécifier l'ensemble contenu dans un champ ensemble de données du paquet en utilisant la propriété *XMLDataSetField*. Si l'élément Web représente un champ ou une valeur de paramètre spécifique, il possède une propriété *FieldName* ou une propriété *ParamName*.

Vous pouvez appliquer un attribut de style à tout élément Web, jouant ainsi sur l'aspect global de tout le document HTML qu'il génère. Styles et feuilles de style font partie du standard HTML 4. Ils permettent à un document HTML de définir un ensemble d'attributs d'affichage à appliquer à une balise et à tout ce qui est de sa portée. Les éléments Web les utilisent de plusieurs façons :

- La façon la plus simple d'utiliser les styles est de définir un attribut de style sur l'élément Web directement. Pour cela, utilisez la propriété *Style*. La valeur de *Style* n'est rien d'autre que la partie définition d'attribut d'une définition de style HTML standard, par exemple :

  ```
  color: red.
  ```

- Vous pouvez aussi définir une feuille de style, qui détermine un ensemble de définitions. Chaque définition inclut un sélecteur de style (soit le nom d'une balise à laquelle le style s'applique constamment, soit le nom d'un style défini par l'utilisateur), plus la définition entre accolades de l'attribut :

  ```
  H2 B  {color: red}
  .MyStyle  {font-family: arial; font-weight: bold; font-size: 18px }
  ```

  L'ensemble global des définitions est maintenu par le producteur de page InternetExpress dans sa propriété *Styles*. Chaque élément Web peut alors faire référence aux styles par les noms définis par l'utilisateur via la propriété *StyleRule*.

- Si vous partagez une feuille de style avec d'autres applications, vous pouvez fournir les définitions de style dans la valeur de la propriété *StylesFile* du producteur de page InternetExpress et non dans la propriété *Styles*. Les éléments Web individuels peuvent continuer à faire référence aux styles via la propriété *StyleRule*.

Autre propriété commune des éléments Web : la propriété *Custom*. *Custom* est un jeu d'options que vous ajoutez à la balise HTML générée. HTML définit un jeu d'options différent pour chaque type de balise. La référence de la VCL donne un exemple des options possibles pour la propriété *Custom* de la plupart des éléments Web. Pour plus d'informations sur ces options, reportez-vous à un guide de référence HTML.

Personnalisation du modèle d'un producteur de page InternetExpress

Le modèle d'un producteur de page InternetExpress est un document HTML contenant des balises imbriquées supplémentaires que votre application traduit de manière dynamique. Au départ, le producteur de page génère un modèle à partir de la valeur de la propriété *HTMLDoc*. Ce modèle par défaut a la forme suivante :

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<#INCLUDES> <#STYLES> <#WARNINGS> <#FORMS> <#SCRIPT>
</BODY>
</HTML>
```

Les balises transparentes pour HTML du modèle par défaut sont traduites comme suit :

<#INCLUDES> génère les instructions qui incluent les bibliothèques javascript. Ces instructions ont la forme suivante :

```
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldom.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldb.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmlbind.js"> </SCRIPT>
```

<#STYLES> génère les instructions qui définissent une feuille de style à partir des définitions listées dans la propriété *Styles* ou *StylesFile* du producteur de page InternetExpress.

<#WARNINGS> ne génère rien pendant l'exécution. Au moment de la conception, elle ajoute des messages d'avertissement concernant les problèmes détectés lors de la génération du document HTML. Vous pouvez voir ces messages dans l'éditeur de pages Web.

<#FORMS> génère le HTML produit par les composants que vous avez ajoutés avec l'éditeur de pages Web. Le HTML correspondant à chaque composant est généré dans l'ordre de *WebPageItems*.

<#SCRIPT> génère le bloc des déclarations javascript utilisées dans le HTML généré par les composants ajoutés avec l'éditeur de pages Web.

Vous pouvez remplacer le modèle par défaut en modifiant la valeur de *HTMLDoc* ou en définissant la propriété *HTMLFile*. Le modèle HTML personnalisé peut comprendre une ou plusieurs des balises transparentes pour HTML appartenant au modèle par défaut. Le producteur de page InternetExpress traduit automatiquement ces balises lorsque vous appelez la méthode *Content*. En outre, le producteur de page InternetExpress traduit automatiquement les trois balises suivantes :

<#BODYELEMENTS> est remplacé par le même HTML que celui résultant des 5 balises du modèle par défaut. Cela peut servir à générer un modèle dans un éditeur HTML lorsque vous voulez utiliser la disposition par défaut mais ajouter des éléments supplémentaires à l'aide de l'éditeur.

<#COMPONENT Name=WebComponentName> est remplacé par le HTML généré par le composant nommé *WebComponentName*. Ce composant peut être l'un de ceux ajoutés dans l'éditeur de pages Web, ou tout autre composant supportant l'interface *IWebContent*, il a le même propriétaire que le producteur de page InternetExpress.

<#DATAPACKET XMLBroker=BrokerName> est remplacé par le paquet de données XML obtenu à partir du courtier XML spécifié par *BrokerName*. Lorsque, dans l'éditeur de pages Web, vous examinez le HTML généré par le producteur de page InternetExpress, vous voyez cette balise à la place du paquet de données lui-même.

De plus, le modèle personnalisé peut inclure toute autre balise transparente pour HTML que vous auriez définie. Lorsque le producteur de page InternetExpress rencontre une balise appartenant à un des sept types qu'il traduit automatiquement, il génère un événement *OnHTMLTag*, dans lequel vous pouvez écrire du code pour effectuer vos propres traductions. Pour plus d'informations sur les modèles HTML, voir "Modèles HTML" à la page 28-15.

Astuce Les composants qui apparaissent dans l'éditeur de pages Web génèrent du code statique. C'est dire que, sauf si le serveur d'applications modifie les métadonnées figurant dans les paquets de données, le code HTML est toujours le même quelque soit le moment où il est généré. Pour éviter, à l'exécution, la génération dynamique de ce code, en réponse à chaque message de demande, vous pouvez copier le HTML généré dans l'éditeur de pages Web et l'utiliser comme modèle. L'éditeur de pages Web affichant une balise <#DATAPACKET> au lieu de l'XML réel, utiliser ce genre de modèle n'empêche pas votre application de lire dynamiquement les paquets de données provenant du serveur d'applications.

Chapitre

26

Utilisation de XML dans les applications de bases de données

En plus de la prise en charge de la connexion à des serveurs de bases de données, Delphi vous permet de travailler avec des documents XML comme s'il s'agissait de serveurs de bases de données. XML (Extensible Markup Language) est un langage de balisage pour la description de données structurées. Les documents XML fournissent un format standard transportable pour les données, qui est utilisé dans les applications Web, la communication inter-entreprises (business-to-business), etc. Pour plus d'informations sur la manipulation directe de documents XML dans Delphi, consultez le chapitre 30, "Utilisation de documents XML".

La prise en charge des documents XML dans Delphi pour les applications de bases de données repose sur un ensemble de composants capables de convertir des paquets de données (la propriété *Data* d'un ensemble de données client) en documents XML et des documents XML en paquets de données. Pour utiliser ces composants, vous devez tout d'abord définir la transformation entre le document XML et le paquet de données. Après avoir défini la transformation, vous pouvez utiliser des composants spéciaux pour

- convertir des documents XML en paquets de données.
- fournir des données d'un document XML et traiter les mises à jour d'un document XML.
- utiliser un document XML comme client d'un fournisseur.

Définition des transformations

Pour pouvoir effectuer des conversions entre des paquets de données et des documents XML, vous devez au préalable définir la relation entre les métadonnées d'un paquet de données et les nœuds du document XML

correspondant. Une description de cette relation est stockée dans un document XML spécial appelé une transformation.

Chaque fichier de transformation contient deux éléments : la correspondance entre les nœuds d'un schéma XML et les champs d'un paquet de données, et un document de squelette XML qui représente la structure des résultats de la transformation. Une transformation est une correspondance unidirectionnelle : à partir d'un schéma ou d'un document XML vers un paquet de données ou à partir des métadonnées d'un paquet de données vers un schéma XML. Les fichiers de transformation se créent souvent par paire : l'un définit la correspondance de XML vers le paquet de données, et l'autre la correspondance du paquet de données vers XML.

Pour créer les fichiers de transformation pour une correspondance, employez l'utilitaire XMLMapper présent dans le répertoire bin.

Correspondance entre les nœuds XML et les champs du paquet de données

XML fournit une méthode textuelle pour stocker ou décrire des données structurées. Les ensembles de données fournissent une autre méthode pour stocker et décrire des données structurées. Pour convertir un document XML en ensemble de données, vous devez donc identifier les correspondances entre les nœuds d'un document XML et les champs d'un ensemble de données.

Considérons, par exemple, un document qui représente un ensemble de messages électroniques. Ce document peut avoir l'apparence suivante (avec un seul message) :

```
<?xml version="1.0" standalone='yes' ?>
<email>
   <head>
      <from>
         <name>Dave Boss</name>
         <address>dboss@MyCo.com</address>
      </from>
      <to>
         <name>Joe Engineer</name>
         <address>jengineer@MyCo.com</address>
      </to>
      <cc>
         <name>Robin Smith/name>
         <address>rsmith@MyCo.com</address>
      </cc>
      <cc>
         <name>Leonard Devon</name>
         <address>ldevon@MyCo.com</address>
      </cc>
   </head>
   <body>
      <subject>XML components</subject>
      <content>
```

```
        Joe,
        Ci-joint la spécification de la gestion du nouveau composant XML dans
        Delphi. C'est une bonne solution à notre application inter-entreprises.
        Le schedule du projet est également joint. Te semble-t'il raisonnable ?
           Dave.
      </content>
      <attachment attachfile="XMLSpec.txt"/>
      <attachment attachfile="Schedule.txt"/>
    </body>
  </email>
```

Une correspondance naturelle entre ce document et un ensemble de données consisterait à mapper chaque message sur un enregistrement unique. L'enregistrement contiendrait des champs pour le nom et l'adresse de l'expéditeur. Comme un message électronique peut avoir plusieurs destinataires, le destinataire (<to>) serait mappé sur un ensemble de données imbriqué. De même, la liste cc serait mappée sur un ensemble de données imbriqué. La ligne de sujet serait mappée sur un champ de type chaîne de caractères tandis que le message lui-même (<content>) serait probablement mappé sur un champ mémo. Les noms des fichiers joints seraient mappés sur un ensemble de données imbriqué, car un message peut avoir plusieurs fichiers joints. Par conséquent, le message électronique ci-dessus serait mappé sur un ensemble de données comme suit :

SenderName	SenderAddress	To	CC	Subject	Content	Attach
Dave Boss	dboss@MyCo.Com	(DataSet)	(DataSet)	XML components	(MEMO)	(DataSet)

où l'ensemble de données imbriqué dans le champ "To" est

Name	Address
Joe Engineer	jengineer@MyCo.Com

l'ensemble de données imbriqué dans le champ "CC" est

Name	Address
Robin Smith	rsmith@MyCo.Com
Leonard Devon	ldevon@MyCo.Com

et l'ensemble de données imbriqué dans le champ "Attach" est

Attachfile
XMLSpec.txt
Schedule.txt

La définition de telles correspondances implique l'identification dans le document XML des nœuds qui peuvent être répétés, et leur correspondance avec des ensembles de données imbriqués. Les éléments balisés qui ont des valeurs et apparaissent une fois seulement (comme <content>...</content>) sont mappés

sur des champs dont le type de données reflète le type des données pouvant apparaître comme valeurs. Les attributs d'une balise (comme l'attribut AttachFile de la balise de pièce jointe) sont également mappés sur des champs.

Notez que les balises du document XML n'apparaissent pas toutes dans l'ensemble de données correspondant. Par exemple, l'élément <head>...<head/> ne possède pas d'élément correspondant dans l'ensemble de données résultant. Généralement, seuls les éléments qui ont des valeurs, les éléments qui peuvent être répétés ou les attributs d'une balise sont mappés sur les champs (y compris les champs d'un ensemble de données imbriqué) d'un ensemble de données. Un nœud père du document XML qui est mappé sur un champ dont la valeur est composée à partir des valeurs des nœuds fils constituerait toutefois une exception à cette règle. Par exemple, un document XML peut contenir un ensemble de balises de la forme

```
<FullName>
   <Title> Mr. </Title>
   <FirstName> John </FirstName>
   <LastName> Smith </LastName>
</FullName>
```

qui peuvent être mappées sur un champ d'ensemble de données unique avec la valeur

```
Mr. John Smith
```

Utilisation de XMLMapper

L'utilitaire mappeur XML, xmlmapper.exe, vous permet de définir des mappages (ou correspondances) de trois manières :

- A partir d'un schéma (ou d'un document) XML existant vers un ensemble de données client que vous définissez. Cette méthode est utile lorsque vous souhaitez créer une application de base de données pour manipuler des données pour lesquelles vous disposez déjà d'un schéma XML.
- A partir d'un paquet de données existant vers un nouveau schéma XML que vous définissez. Cette méthode est utile lorsque vous souhaitez présenter des informations de bases de données existantes dans XML, par exemple pour créer un nouveau système de communication inter-entreprise.
- Entre un schéma XML existant et un paquet de données existant. Cette méthode est utile lorsque vous disposez d'un schéma XML et d'une base de données qui décrivent tous deux les mêmes informations et que vous souhaitez les faire travailler ensemble.

Après avoir défini le mappage, vous pouvez générer les fichiers de transformation pour la conversion des documents XML en paquets de données et vice-versa. Notez que seul le fichier de transformation est directionnel : un même mappage peut être utilisé pour générer à la fois la transformation de XML vers le paquet de données et du paquet de données vers XML.

Remarque Le mappeur XML se base sur deux .DLL (midas.dll et msxml.dll). Assurez-vous que ces .DLL sont toutes deux installées avant d'essayer d'utiliser

xmlmapper.exe. De plus, msxml.dll doit être enregistré en tant que serveur COM. Vous pouvez effectuer cet enregistrement au moyen de Regsvr32.exe.

Chargement d'un schéma XML ou d'un paquet de données

Avant de pouvoir définir un mappage et générer un fichier de transformation, vous devez charger les descriptions du document XML et du paquet de données entre lesquels vous êtes en train de définir un mappage.

Vous pouvez charger un document ou un schéma XML en choisissant Fichier | Ouvrir et en sélectionnant le document ou le schéma dans la boîte de dialogue résultante.

Vous pouvez charger un paquet de données en choisissant Fichier | Ouvrir et en sélectionnant un fichier de paquet de données dans la boîte de dialogue résultante. Le paquet de données est simplement le fichier généré lorsque vous appelez la méthode *SaveToFile* d'un ensemble de données client. Si vous n'avez pas enregistré le paquet de données sur disque, vous pouvez directement accéder au paquet de données à partir du serveur d'applications d'une application à plusieurs niveaux en cliquant avec le bouton droit de la souris dans la vue Paquet de données et en choisissant Connexion à un serveur distant.

Vous pouvez charger seulement un document ou schéma XML, seulement un paquet de données, ou les deux. Si vous ne chargez qu'un côté du mappage, le mappeur XML peut générer un mappage naturel pour l'autre côté.

Définition des mappages

Le mappage entre un document XML et un paquet de données n'inclut pas nécessairement tous les champs du paquet de données ou tous les éléments balisés du document XML. Vous devez donc spécifier les éléments à mapper. Pour cela, sélectionnez d'abord la page Affectations du volet central de la boîte de dialogue.

Pour spécifier les éléments d'un document ou d'un schéma XML à mapper sur les champs d'un paquet de données, sélectionnez l'onglet Exemple ou Structure du volet Document XML et double-cliquez sur les nœuds des éléments à mapper sur des champs du paquet de données.

Pour spécifier les champs du paquet de données à mapper sur des éléments balisés ou des attributs du document XML, double-cliquez sur les nœuds correspondant à ces champs dans la vue Paquet de données.

Si vous n'avez chargé qu'un seul côté du mappage (le document XML ou le paquet de données), vous pouvez générer l'autre côté après avoir sélectionné les nœuds à mapper.

- Si vous générez un paquet de données à partir d'un document XML, vous devez définir tout d'abord des attributs pour les nœuds sélectionnés déterminant les types de champs auxquels ils correspondent dans le paquet de données. Dans le volet central, sélectionnez la page Propriétés du nœud. Sélectionnez chaque nœud qui participe au mappage et indiquez les attributs du champ correspondant. Si le mappage n'est pas simple (comme dans le cas d'un nœud avec des sous-nœuds correspondant à un champ dont la valeur est

construite à partir de ces sous-nœuds), activez la case à cocher Traduction définie par l'utilisateur. Vous devrez écrire par la suite un gestionnaire d'événement pour effectuer la transformation sur des nœuds définis par l'utilisateur.

Après avoir spécifié la manière dont les nœuds doivent être mappés, choisissez Créer | Paquet de données à partir de données XML. Le paquet de données correspondant est automatiquement généré et affiché dans la vue Paquet de données.

- Si vous générez un document XML à partir d'un paquet de données, choisissez Créer | Données XML à partir d'un paquet de données. Une boîte de dialogue apparaît dans laquelle vous pouvez spécifier les noms des balises et des attributs dans le document XML qui correspondent aux champs, enregistrements et ensembles de données du paquet de données. Pour les valeurs des champs, vous indiquez si elles sont mappées sur un élément balisé avec une valeur ou sur un attribut en les nommant de façon différente. Les noms commençant par un symbole @ mappent sur des attributs de la balise qui correspond à l'enregistrement, tandis que les noms ne commençant pas par un symbole @ mappent sur des éléments balisés qui possèdent des valeurs et sont imbriqués dans l'élément de l'enregistrement.
- Si vous avez chargé à la fois un document XML et un paquet de données (fichier d'ensemble de données client), assurez-vous de sélectionner les nœuds correspondants dans le même ordre. Les nœuds correspondants devraient apparaître les uns à côté des autres dans le tableau situé en haut de la page Affectations.

Lorsque vous avez chargé ou généré le document XML et le paquet de données puis sélectionné les nœuds apparaissant dans le mappage, le tableau en haut de la page Affectations devrait refléter le mappage que vous avez défini.

Génération de fichiers de transformation

Pour générer un fichier de transformation, utilisez les étapes suivantes :

1. Sélectionnez d'abord le bouton radio indiquant la transformation à créer :
   - Choisissez le bouton Paquet de données en XML si le mappage s'effectue d'un paquet de données vers un document XML.
   - Choisissez le bouton XML en paquet de données si le mappage s'effectue d'un document XML vers un paquet de données.
2. Si vous générez un paquet de données, vous pouvez également utiliser les boutons radio de la section Format du paquet de données créé. Ces boutons vous permettent d'indiquer comment le paquet de données sera utilisé : comme ensemble de données, comme paquet delta pour l'application de mises à jour, ou comme paramètres à transmettre à un fournisseur avant de lire des données.
3. Cliquez sur Créer et tester le fichier de transformation pour générer une version en mémoire de la transformation. Le mappeur XML affiche le document XML qui serait généré pour le paquet de données dans la vue

Paquet de données ou le paquet de données qui serait généré pour le document XML dans le volet Document XML.

4 Enfin, choisissez Fichier | Enregistrer | Transformation pour enregistrer le fichier de transformation. Le fichier de transformation est un fichier XML spécial (avec l'extension .xtr) qui décrit la transformation que vous avez définie.

Conversion de documents XML en paquets de données

Après avoir créé un fichier de transformation indiquant la manière de transformer un document XML en paquet de données, vous pouvez créer des paquets de données pour tout document XML conforme au schéma utilisé dans la transformation. Ces paquets de données peuvent ensuite être affectés à un ensemble de données client et enregistrés dans un fichier pour constituer la base d'une application de base de données basée sur un fichier.

Le composant *TXMLTransform* transforme un document XML en paquet de données selon le mappage défini dans un fichier de transformation.

Remarque Vous pouvez aussi utiliser *TXMLTransform* pour convertir un paquet de données qui apparaît au format XML en un document XML arbitraire.

Spécification du document XML source

Il existe trois méthodes pour spécifier le document XML source :

- Si le document source est un fichier .xml sur disque, vous pouvez utiliser la propriété *SourceXmlFile*.
- Si le document source est une chaîne XML en mémoire, vous pouvez utiliser la propriété *SourceXml*.
- Si vous disposez d'une interface IDOMDocument pour le document source, vous pouvez utiliser la propriété *SourceXmlDocument*.

TXMLTransform vérifie ces propriétés dans l'ordre indiqué ci-dessus. Ainsi, il recherche d'abord un nom de fichier dans la propriété *SourceXmlFile*. Si *SourceXmlFile* est une chaîne vide, il vérifie la propriété *SourceXml*. Si *SourceXml* est une chaîne vide, il vérifie alors la propriété *SourceXmlDocument*.

Spécification de la transformation

Il existe deux méthodes pour spécifier la transformation qui convertit le document XML en paquet de données :

- Utilisez la propriété *TransformationFile* pour indiquer un fichier de transformation créé à l'aide de xmlmapper.exe.
- Utilisez la propriété *TransformationDocument* si vous disposez d'une interface *IDOMDocument* pour la transformation.

TXMLTransform vérifie ces propriétés dans l'ordre indiqué ci-dessus. Ainsi, il recherche d'abord un nom de fichier dans la propriété *TransformationFile*. Si *TransformationFile* est une chaîne vide, il vérifie la propriété *TransformationDocument*.

Obtention du paquet de données résultant

Pour que *TXMLTranform* effectue sa transformation et génère un paquet de données, il vous suffit de lire la propriété *Data*. Par exemple, le code suivant utilise un document XML et un fichier de transformation pour générer un paquet de données, qui est ensuite affecté à un ensemble de données client :

```
XMLTransform1.SourceXMLFile := 'CustomerDocument.xml';
XMLTransform1.TransformationFile := 'CustXMLToCustTable.xtr';
ClientDataSet1.XMLData := XMLTransform1.Data;
```

Conversion de nœuds définis par l'utilisateur

Lorsque vous définissez une transformation à l'aide de xmlmapper.exe, vous pouvez spécifier que certains nœuds du document XML sont "définis par l'utilisateur". Les nœuds définis par l'utilisateur sont les nœuds pour lesquels vous souhaitez fournir la transformation sous forme de code plutôt qu'en vous basant sur une traduction directe d'une valeur de nœud vers une valeur de champ.

Vous pouvez fournir le code de traduction des nœuds définis par l'utilisateur en utilisant l'événement *OnTranslate*. *OnTranslate* est appelé chaque fois que le composant *TXMLTransform* rencontre un nœud défini par l'utilisateur dans le document XML. Dans le gestionnaire d'événement OnTranslate, vous pouvez lire le document source et spécifier la valeur résultante pour le champ dans le paquet de données.

Par exemple, le gestionnaire d'événement *OnTranslate* suivant convertit un nœud du document XML de la forme suivante

```
<FullName>
   <Title> </Title>
   <FirstName> </FirstName>
   <LastName> </LastName>
</FullName>
```

en valeur de champ unique :

```
procedure TForm1.XMLTransform1Translate(Sender: TObject; Id: String; SrcNode: IDOMNode;
  var Value: String; DestNode: IDOMNode);
var
  CurNode: IDOMNode;
begin
  if Id = 'FullName' then
  begin
    Value = '';
    if SrcNode.hasChildNodes then
```

```
    begin
      CurNode := SrcNode.firstChild;
      Value := Value + CurNode.nodeValue;
      while CurNode <> SrcNode.lastChild do
      begin
        CurNode := CurNode.nextSibling;
        Value := Value + ' ';
        Value := Value + CurNode.nodeValue;
      end;
    end;
  end;
end;
```

Utilisation d'un document XML comme source pour un fournisseur

Le composant *TXMLTransformProvider* vous permet d'utiliser un document XML comme s'il s'agissait d'une table de base de données. *TXMLTransformProvider* prépare les données d'un document XML et applique en retour les mises à jour des clients à ce document XML. Il apparaît aux clients tels que les ensembles de données client ou les courtiers XML comme n'importe quel autre composant fournisseur. Pour des informations sur des composants fournisseur, consultez le chapitre 24, "Utilisation des composants fournisseur". Pour des informations sur l'utilisation de composants fournisseur avec des ensembles de données clients, consultez "Utilisation d'un ensemble de données client avec un fournisseur" à la page 23-29.

Vous pouvez spécifier le document XML à partir duquel le fournisseur XML fournit les données et auquel il applique les mises à jour au moyen de la propriété *XMLDataFile*.

Les composants *TXMLTransformProvider* utilisent des composants *TXMLTransform* internes pour effectuer les traductions entre les paquets de données et le document XML source : un composant pour traduire le document XML en paquets de données et un autre pour traduire en retour les paquets de données dans le format XML du document source après l'application des mises à jour. Ces deux composants *TXMLTransform* sont respectivement accessibles à l'aide des propriétés *TransformRead* et *TransformWrite*.

En utilisant *TXMLTransformProvider*, vous devez spécifier les transformations utilisées par ces deux composants *TXMLTransform* pour la traduction entre les paquets de données et le document XML source. Utilisez pour cela la propriété *TransformationFile* ou *TransformationDocument* du composant *TXMLTransform*, exactement comme pour utiliser un composant *TXMLTransform* autonome.

En outre, si la transformation inclut des nœuds définis par l'utilisateur, vous devez fournir un gestionnaire d'événement *OnTranslate* aux composants *TXMLTransform* internes.

Il n'est pas nécessaire de spécifier le document source pour les composants *TXMLTransform* qui constituent les valeurs de *TransformRead* et *TransformWrite*.

Pour *TransformRead*, la source est le fichier spécifié par la propriété *XMLDataFile* du fournisseur (toutefois, si vous affectez à *XMLDataFile* une chaîne vide, vous pouvez spécifier le document source en utilisant *TransformRead.XmlSource* ou *TransformRead.XmlSourceDocument*). Pour *TransformWrite*, le source est généré au niveau interne par le fournisseur lorsqu'il applique des mises à jour.

Utilisation d'un document XML comme client d'un fournisseur

Le composant *TXMLTransformClient* se comporte comme adaptateur pour vous permettre d'utiliser un document (ou un ensemble de documents) XML comme client pour un serveur d'applications (ou simplement comme client d'un ensemble de données auquel il se connecte par l'intermédiaire d'un composant *TDataSetProvider*). Cela veut dire que le client *TXMLTransform* vous permet de publier des données d'une base de données en tant que document XML et d'utiliser des requêtes de mise à jour (insertions ou suppressions) à partir d'une application externe qui les fournit sous la forme de documents XML.

Pour spécifier le fournisseur à partir duquel l'objet *TXMLTransformClient* lit des données et auquel il applique des mises à jour, utilisez la propriété *ProviderName*. Comme avec la propriété *ProviderName* d'un ensemble de données client, *ProviderName* peut être le nom d'un fournisseur sur un serveur d'applications distant ou un fournisseur local dans la même fiche ou le même module de données que l'objet *TXMLTransformClient*. Pour des informations sur les fournisseurs, consultez le chapitre 24, "Utilisation des composants fournisseur".

Si le fournisseur se trouve sur un serveur d'applications distant, vous devez utiliser un composant de connexion DataSnap pour vous connecter à ce serveur d'applications. Spécifiez le composant de connexion au moyen de la propriété *RemoteServer*. Pour des informations sur les composants de connexion DataSnap, consultez "Connexion au serveur d'applications" à la page 25-27.

Lecture d'un document XML à partir d'un fournisseur

TXMLTransformClient utilise un composant *TXMLTransform* interne pour traduire les paquets de données du fournisseur en document XML. Vous pouvez accéder à ce composant TXMLTransform avec la valeur de la propriété *TransformGetData*.

Pour pouvoir créer un document XML représentant les données d'un fournisseur, vous devez spécifier le fichier de transformation utilisé par *TransformGetData* pour traduire le paquet de données au format XML approprié. Pour cela, initialisez la propriété *TransformationFile* ou *TransformationDocument* du composant *TXMLTransform*, exactement comme pour l'utilisation d'un composant *TXMLTransform* autonome. Si cette transformation inclut des nœuds définis par l'utilisateur, vous pouvez également fournir à *TransformGetData* un gestionnaire d'événement *OnTranslate*.

Il n'est pas nécessaire de spécifier le document source pour *TransformGetData*, *car TXMLTransformClient* le récupère à partir du fournisseur. Toutefois, si le fournisseur attend des paramètres d'entrée, vous pouvez les définir avant de lire

les données. Utilisez la méthode *SetParams* pour fournir ces paramètres d'entrée avant de lire les données à partir du fournisseur. *SetParams* accepte deux arguments : une chaîne XML à partir de laquelle extraire les valeurs des paramètres, et le nom d'un fichier de transformation pour traduire ce code XML en paquet de données. *SetParams* utilise le fichier de transformation pour convertir la chaîne XML en paquet de données, puis extrait les valeurs des paramètres de ce paquet de données.

Remarque Vous pouvez outrepasser l'un quelconque de ces arguments si vous souhaitez spécifier le document ou la transformation d'une autre manière. Il vous suffit de définir l'une des propriétés avec *TransformSetParams* pour indiquer les paramètres ou la transformation à utiliser pour les convertir, puis d'affecter à l'argument que vous souhaitez outrepasser une chaîne vide lorsque vous appelez *SetParams*. Pour des détails sur les propriétés que vous pouvez utiliser, consultez "Conversion de documents XML en paquets de données" à la page 26-7.

Après avoir configuré *TransformGetData* et spécifié les éventuels paramètres d'entrée, vous pouvez appeler la méthode *GetDataAsXml* pour récupérer le code XML. *GetDataAsXml* envoie les valeurs de paramètres en cours au fournisseur, récupère un paquet de données, le convertit en document XML puis renvoie ce document sous forme de chaîne de caractères. Vous pouvez enregistrer cette chaîne dans un fichier :

```
var
  XMLDoc: TFileStream;
  XML: string;
begin
  XMLTransformClient1.ProviderName := 'Provider1';
  XMLTransformClient1.TransformGetData.TransformationFile := 'CustTableToCustXML.xtr';
  XMLTransformClient1.TransFormSetParams.SourceXmlFile := 'InputParams.xml';
  XMLTransformClient1.SetParams('', 'InputParamsToDP.xtr');
  XML := XMLTransformClient1.GetDataAsXml;
  XMLDoc := TFileStream.Create('Customers.xml', fmCreate or fmOpenWrite);
  try
    XMLDoc.Write(XML, Length(XML));
  finally
    XMLDoc.Free;
  end;
end;
```

Application de mises à jour d'un document XML à un fournisseur

TXMLTransformClient vous permet aussi d'insérer toutes les données d'un document XML dans l'ensemble de données du fournisseur ou de supprimer tous les enregistrements d'un document XML de l'ensemble de données du fournisseur. Pour effectuer ces mises à jour, appelez la méthode *ApplyUpdates* en transmettant

- Une chaîne dont la valeur est le contenu du document XML avec les données à insérer ou à supprimer.

- Le nom d'un fichier de transformation capable de convertir ces données XML en paquet delta d'insertion ou de suppression. En définissant le fichier de transformation à l'aide de l'utilitaire mappeur XML, vous indiquez si la transformation est destinée à un paquet delta d'insertion ou de suppression.
- Le nombre d'erreurs de mise à jour qui peuvent être tolérées sans que l'opération de mise à jour ne soit abandonnée. Si le nombre d'enregistrements qui ne peuvent pas être insérés ou supprimés est inférieur à ce paramètre, *ApplyUpdates* renvoie le nombre d'erreurs réelles. Si le nombre d'enregistrements qui ne peuvent pas être insérés ou supprimés est égal à ce paramètre, l'opération de mise à jour tout entière est annulée et aucune mise à jour n'est effectuée.

L'appel suivant transforme le document XML Customers.xml en paquet delta et applique toutes les mises à jour quel que soit le nombre d'erreurs :

```
StringList1.LoadFromFile('Customers.xml');
nErrors := ApplyUpdates(StringList1.Text, 'CustXMLToInsert.xtr', -1);
```

Partie

III

Ecriture d'applications Internet

Les chapitres de cette section présentent les concepts et les connaissances nécessaires à la construction d'applications distribuées sur Internet.

Remarque Les composants décrits dans cette section ne sont pas disponibles dans toutes les éditions de Delphi.

Chapitre

27

Création d'applications Internet

Les applications serveur Web étendent les fonctions et les possibilités des serveurs Web existants. Une application serveur Web reçoit des messages de requête HTTP du serveur Web, effectue les actions demandées par les messages et formule des réponses qui sont renvoyées au serveur Web. Toute opération qui peut s'effectuer dans une application Delphi peut être incorporée dans une application serveur Web.

Delphi propose deux architectures différentes pour développer des applications serveur Web : l'agent Web et WebSnap. Bien que ces deux architectures soient différentes, WebSnap et l'agent Web ont de nombreux éléments en commun. L'architecture WebSnap se comporte comme un sur-ensemble de l'agent Web. Elle propose des composants supplémentaires et de nouvelles caractéristiques comme le concepteur de surface WebSnap (qui permet au développeur d'afficher le contenu d'une page sans avoir à exécuter l'application). Les applications développées avec WebSnap peuvent contenir des composants agent Web alors que l'inverse n'est pas vrai.

Ce chapitre décrit les caractéristiques des architectures agent Web et WebSnap et donne des informations générales sur les applications client-serveur Internet.

A propos de l'agent Web et de WebSnap

La première étape de la conception d'une application serveur Web consiste à choisir l'architecture que vous souhaitez utiliser. Ces deux architectures ont de nombreuses caractéristiques en commun :

- Gestion de plusieurs types d'applications serveur Web, dont ISAPI, NSAPI, CGI, Win CGI et Apache. Elles sont décrites dans la section "Types d'applications serveur Web" à la page 27-6.
- Gestion multithread permettant aux requêtes client entrantes d'être traitées dans des threads distincts.

- Mise en cache des modules Web afin d'obtenir des meilleurs temps de réponse.

Néanmoins, ces deux approches présentent des avantages et inconvénients. Les principales différences entre ces deux technologies sont présentées dans le tableau suivant :

Tableau 27.1 Comparaison de l'agent Web et de WebSnap

Agent Web	WebSnap
Compatibilité ascendante.	Même si les applications WebSnap peuvent utiliser tous les composants agent Web produisant un contenu, les modules Web et le répartiteur qui les contiennent sont nouveaux.
Disponibles dans des applications multiplates-formes (CLX).	Actuellement, WebSnap est disponible uniquement sous Windows.
Un seul module Web par application.	Plusieurs modules Web peuvent décomposer l'application en unités, ce qui permet à plusieurs développeurs de travailler sur le même projet en minimisant les conflits.
Un seul répartiteur Web par application.	Plusieurs répartiteurs spécifiques peuvent traiter des types différents de requête.
Des composants spécialisés pour créer le contenu (générateurs de page, composants InternetExpress et les composants service Web).	Gère tous les générateurs de contenu apparaissant dans des applications agent Web, plus d'autres composants conçus pour permettre la conception rapide de pages Web orientées données.
Pas de gestion des scripts.	Gestion des scripts côté serveur (JScript ou VBscript) ce qui permet de séparer la logique de génération du HTML de la logique de métier.
Pas de gestion intégrée des pages nommées.	Les pages nommées peuvent être obtenues automatiquement par un répartiteur de page et désignées par des scripts côté serveur.
Pas de gestion des sessions.	Les sessions stockent les informations nécessaires pour un court moment sur un utilisateur final. Cela peut être utilisé pour des opérations comme la gestion de la connexion/déconnexion.
Chaque requête doit être gérée explicitement en utilisant un élément action ou un composant d'auto-répartition.	Les composants répartiteur peuvent répondre automatiquement à diverses requêtes.
Seuls des composants spécialisés permettent de prévisualiser le contenu qu'ils génèrent. L'essentiel du développement se fait de manière non visuelle.	Le concepteur de surface WebSnap vous permet de concevoir visuellement les pages Web et de visualiser le résultat à la conception. La prévisualisation est disponible pour tous les composants.

Pour davantage d'informations sur l'agent Web, voir chapitre 28, "Utilisation de l'agent Web". Pour davantage d'informations sur WebSnap, voir chapitre 29, "Utilisation de WebSnap".

Terminologie et standard

La majorité des protocoles contrôlant l'activité Internet sont définis dans des documents Request for Comment (RFC) gérés par le comité IETF (Internet Engineering Task Force), comité chargé de l'ingénierie et du développement des protocoles Internet. Plusieurs documents RFC vous apporteront d'utiles précisions pour le développement d'applications Internet :

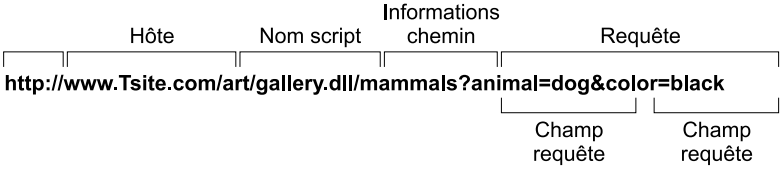
- Le RFC822, "Standard for the format of ARPA Internet text messages," décrit la structure et le contenu des en-têtes de messages.
- Le RFC1521, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies," décrit la méthode utilisée pour encapsuler et transporter des messages multiparties et multiformats.
- Le RFC1945, "Hypertext Transfer Protocol — HTTP/1.0," décrit une méthode de transfert utilisée pour distribuer des documents hypermédia collaboratifs.

Le comité IETF propose une bibliothèque des documents RFC sur le site Web `www.ietf.cnri.reston.va.us`

Composition d'une URL (Uniform Resource Locator)

L'URL est une description complète de l'emplacement d'une ressource sur Internet. Une URL se compose de plusieurs parties auxquelles une application peut accéder. Ces différentes parties sont décrites dans la figure suivante :

Figure 27.1 Composants d'une URL



La première partie (qui n'appartient pas techniquement à l'URL) identifie le protocole (http). Cette partie peut spécifier d'autres protocoles comme https (secure http), ftp, etc.

La partie Hôte identifie la machine qui exécute le serveur Web et l'application serveur Web. Bien que cela n'apparaisse pas dans l'illustration précédente, cette partie peut se substituer au port qui reçoit les messages. Généralement, il n'est pas nécessaire de spécifier un port, car le protocole implique le numéro de port.

La partie Nom script spécifie le nom de l'application serveur Web. Il s'agit de l'application à laquelle le serveur Web transmet les messages.

A la suite du nom de script apparaît le chemin. Ce dernier identifie la destination du message dans l'application serveur Web. Les valeurs du chemin peuvent faire référence à des répertoires de la machine hôte, aux noms de composants qui répondent à des messages spécifiques ou à tout autre mécanisme

utilisé par l'application serveur Web pour diviser le traitement des messages entrants.

La partie Requête contient un ensemble de valeurs nommées. Ces valeurs et leurs noms sont définis par l'application serveur Web.

URI et URL

L'URL est un sous-ensemble d'une URI (Uniform Resource Identifier) défini dans le document RFC1945. Les applications serveur Web génèrent fréquemment un contenu à partir de plusieurs sources ; le résultat final ne réside pas à un emplacement précis, mais est créé si nécessaire. Les URI peuvent décrire des ressources n'ayant pas d'emplacement défini.

En-tête de message de requête HTTP

Les messages de requête HTTP contiennent plusieurs en-têtes donnant des informations sur le client, la cible de la requête, la manière dont la requête doit être traitée et ce qui a été expédié avec la requête. Chaque en-tête est identifié par un nom, comme "Host" suivi d'une valeur chaîne. Soit par exemple, la requête HTTP suivante :

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

La première ligne identifie la requête comme un GET. Un message de requête GET demande à l'application serveur Web de renvoyer le contenu associé à l'URI suivant le mot GET (ici /art/gallery.dll/animals?animal=doc&color=black). La dernière partie de la première ligne indique que le client utilise le standard HTTP 1.0.

La deuxième ligne est l'en-tête Connection qui indique que la connexion ne doit pas être fermée tant que la requête n'a pas été traitée. La troisième ligne est l'en-tête User-Agent qui donne des informations sur le programme qui a généré la demande. La ligne suivante définit l'en-tête Host et indique le nom et le port de l'hôte sur le serveur qui est contacté pour constituer la connexion. La dernière ligne est un en-tête Accept qui énumère les types de données que le client accepte en réponse.

Activité d'un serveur HTTP

La nature client/serveur des navigateurs Web peut sembler, à tort, simple à implémenter. Pour la majorité des utilisateurs, la récupération d'informations sur le Web est une procédure ne nécessitant que quelques clics de souris. Certains utilisateurs ont quelques notions de la syntaxe HTML et de la nature client/serveur des protocoles utilisés. Ceci est généralement suffisant pour la création de sites Web simples en mode page. Les créateurs de pages Web plus complexes

ont accès à de nombreuses options pour automatiser la recherche et la présentation des informations au format HTML.

Avant de construire une application serveur Web, il est utile de comprendre comment le client effectue une requête et comment le serveur y répond.

Composition des requêtes client

Lorsqu'un lien hypertexte HTML est sélectionné (ou lorsque l'utilisateur indique une URL), le navigateur lit les informations sur, entre autres, le protocole, le domaine spécifié, le chemin d'accès aux informations, la date et l'heure, l'environnement système et le navigateur lui-même. Il compose ensuite une requête.

Par exemple, pour afficher une page d'images basée sur des critères représentés par des boutons à l'écran, le client peut générer cette URL :

```
http://www.TSite.com/art/gallery.dll/animals?animal=dog&color=black
```

qui décrit un serveur HTTP dans le domaine www.TSite.com. Le client contacte www.TSite.com, se connecte au serveur HTTP et lui transmet une requête. La requête est semblable à celle-ci :

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

Traitement des requêtes client par le serveur

Le serveur Web reçoit une requête du client et effectue diverses opérations en fonction de sa configuration. Si le serveur reconnaît la partie /gallery.dll de la requête comme un programme, il transfère à ce programme des informations sur la requête. La façon dont les informations sur la requête sont passées au programme dépend du type d'application serveur Web :

- Si le programme est un programme CGI (Common Gateway Interface), le serveur passe directement les informations contenues dans la requête au programme CGI. Le serveur attend l'exécution du programme. Lorsque le programme CGI se termine, il retransmet le contenu au serveur.
- Si le programme est de type WinCGI, le serveur ouvre un fichier et y écrit les informations de requête. Il exécute ensuite le programme Win-CGI en lui transmettant l'emplacement du fichier contenant la description de la demande du client et l'emplacement du fichier que le programme Win-CGI doit utiliser pour créer le contenu. Le serveur attend l'exécution du programme. Lorsque le programme se termine, le serveur lit le fichier de contenu créé par le programme Win-CGI.
- Si le programme est une DLL, le serveur la charge si nécessaire et passe les informations contenues dans la requête à la DLL, sous forme de structure. Le

serveur attend l'exécution du programme. Lorsque la DLL se referme, elle retransmet le contenu au serveur.

Dans ces trois cas, le programme agit sur la requête et effectue les actions demandées par le programmeur : accéder aux bases de données, faire des recherches ou des calculs simples, créer ou sélectionner des documents HTML, etc.

Réponses aux requêtes client

Lorsqu'une application serveur Web a fini de traiter une requête client, elle construit une page de code HTML (ou autre contenu MIME) et la transmet au client (via le serveur) qui l'affiche.

La méthode retenue pour l'envoi de la réponse dépend aussi du type du programme :

- Lorsqu'un script Win-CGI se termine, il construit une page HTML, la place dans un fichier, stocke les informations de réponse dans un autre fichier et transmet l'emplacement de ces deux fichiers au serveur. Le serveur ouvre ensuite ces fichiers et envoie la page HTML au client.
- Lorsqu'une DLL se termine, elle passe la page HTML et les informations de réponse au serveur, qui les transmet au client.

Le fait de créer une application serveur Web en tant que DLL réduit la charge de travail du système en diminuant le nombre de processus et d'accès disque requis pour répondre à une requête.

Types d'applications serveur Web

Que vous utilisiez l'agent Web ou WebSnap, vous pouvez créer cinq types d'applications serveur Web. De plus, vous pouvez créer un exécutable débogueur d'application Web, qui intègre le serveur Web à votre application afin de pouvoir déboguer la logique de l'application. L'exécutable débogueur d'application Web ne sert qu'au débogage. Lors du déploiement de votre application, vous devez la faire migrer vers l'un des cinq autres types.

Chacun des cinq types d'application serveur Web utilise un type particulier de descendant de *TWebApplication*, *TWebRequest* et de *TWebResponse* :

Tableau 27.2 Composants des applications serveur Web

Type d'application	Objet application	Objet requête	Objet réponse
DLL Microsoft Server (ISAPI)	*TISAPIApplication*	*TISAPIRequest*	*TISAPIResponse*
DLL Netscape Server (NSAPI)	*TISAPIApplication*	*TISAPIRequest*	*TISAPIResponse*
DLL Serveur Apache	TApacheApplication	TApacheRequest	TApacheResponse
Application console CGI	*TCGIApplication*	*TCGIRequest*	*TCGIResponse*
Application Windows CGI	*TCGIApplication*	*TWinCGIRequest*	*TWinCGIResponse*

ISAPI et NSAPI

Une application serveur Web ISAPI ou NSAPI est une DLL qui est chargée par le serveur Web. Les informations de requête client sont transmises à la DLL sous forme de structure et évaluées par les objets *TISAPIApplication*, qui créent les objets *TISAPIRequest* et *TISAPIResponse*. Chaque message de requête est automatiquement traité dans un thread d'exécution distinct.

Apache

Une application serveur Web Apache est une DLL qui est chargée par le serveur Web. Les informations de requête client sont transmises à la DLL sous forme de structure et évaluées par les objets *TApacheApplication*, qui créent les objets *TApacheRequest* et *TApacheResponse*. Chaque message de requête est automatiquement traité dans un thread distinct.

CGI autonome

Une application serveur Web CGI autonome est une application console qui reçoit les informations de requête client sur l'entrée standard et transmet les résultats au serveur sur la sortie standard. Les données sont évaluées par l'objet *TCGIApplication*, qui crée le répartiteur, et par les objets *TCGIRequest* et *TCGIResponse*. Chaque message de requête est traité dans une instance distincte de l'application.

Win-CGI autonome

Une application serveur Web Win-CGI autonome est une application Windows qui reçoit les informations de requête client depuis un fichier de configuration (INI) créé par le serveur et écrit les résultats dans un fichier que le serveur transmet au client. Le fichier INI est évalué par l'objet *TCGIApplication*, qui crée le répartiteur, et par les objets *TWinCGIRequest* et *TWinCGIResponse*. Chaque message de requête est traité dans une instance distincte de l'application.

Débogage d'applications serveur

Le débogage des applications serveur Web présente des problèmes spécifiques car l'exécution de ces applications est influencée par les messages qu'elles reçoivent du serveur Web. Lancer votre application depuis l'EDI n'est pas suffisant, car cela ne tiendrait pas compte du serveur Web et votre application ne trouverait pas les messages de requêtes qu'elle attend.

Les sections suivantes décrivent les techniques utilisables pour déboguer des applications serveur Web .

Utilisation du débogueur d'application Web

Le débogueur d'application Web permet de suivre simplement les requêtes et réponses HTTP ainsi que les temps de réponse. Le débogueur d'application Web se substitue au serveur Web. Une fois l'application déboguée, vous pouvez la convertir en l'un des types d'application Web supporté et l'installer sur un serveur Web commercial.

Pour utiliser le débogueur d'application Web, vous devez commencer par créer votre application Web comme exécutable débogueur d'application Web. Que vous utilisiez l'agent Web ou WebSnap, l'expert qui crée votre application serveur Web le propose comme option lors du démarrage de l'application. Cela crée une application serveur Web qui est également un serveur COM.

Pour davantage d'informations sur la conception d'applications serveur Web en utilisant l'agent Web, voir chapitre 28, "Utilisation de l'agent Web". Pour davantage d'informations sur l'utilisation de WebSnap, voir chapitre 29, "Utilisation de WebSnap".

Démarrage de l'application avec le débogueur d'application Web

Une fois votre application serveur Web conçue, vous pouvez l'exécuter et la déboguer de la manière suivante :

1. Votre projet étant ouvert dans l'EDI, définissez les points d'arrêt comme pour déboguer n'importe quel exécutable.
2. Choisissez Exécuter | Exécuter. Cela affiche la fenêtre console du serveur COM qui constitue votre application serveur Web. A la première exécution de votre application, elle recense le serveur COM afin que le débogueur d'application Web puisse y accéder.
3. Sélectionnez Outils | Débogueur d'application Web.
4. Choisissez le bouton Démarrer. Cela affiche la page ServerInfo dans votre navigateur par défaut.
5. La page ServerInfo propose une liste déroulante de tous les exécutables débogueur d'application Web recensés. Sélectionnez votre application dans la liste déroulante. Si vous ne la trouvez pas dans la liste déroulante, essayez de démarrer votre application comme exécutable. Votre application doit être exécutée au moins une fois pour se recenser. Si vous ne trouvez toujours pas votre application dans la liste déroulante, essayez de réactualiser la page Web (en effet, parfois le navigateur Web cache cette page, ce qui empêche de voir les modifications récentes).
6. Une fois votre application sélectionnée dans la liste déroulante, choisissez le bouton de démarrage. Cela démarre votre application dans le débogueur d'application Web, qui vous donne des détails sur les messages de requête et de réponse échangés entre votre application et le débogueur d'application Web.

Conversion de votre application vers un autre type d'application serveur Web

Une fois achevé la mise au point de votre application serveur Web, vous devez la convertir vers un autre type d'application Web avant de pouvoir l'installer dans un serveur Web commercial. Les étapes suivantes décrivent la manière d'effectuer cette conversion :

1. Dans l'EDI choisissez Projet | Ajouter un nouveau projet. Cela ouvre un nouveau projet sans fermer le projet en cours (votre application serveur Web).
2. Exécutez l'expert permettant de créer une application agent Web ou WebSnap et choisissez le type d'application à créer.
3. Choisissez Voir | Gestionnaire de projet pour afficher le gestionnaire de projet.
4. Dans le gestionnaire de projet, faites glisser toutes les unités constituant votre application serveur Web de l'ancien projet vers celui que vous venez d'ajouter. Ne déplacez pas les unités qui implémentent la fenêtre console.

Vous disposez maintenant d'une application serveur Web du type approprié.

Débogage d'applications Web sous forme de DLL

Les applications ISAPI, NSAPI et Apache sont en fait des DLL qui contiennent des points d'entrée prédéfinis. Le serveur Web passe les messages de requête à l'application en faisant des appels à ces points d'entrée. Comme ces applications sont des DLL, vous devrez définir les paramètres d'exécution de votre application de telle façon qu'elle lance le serveur.

Pour configurer les paramètres d'exécution de votre application, choisissez Exécuter | Paramètres et initialisez les zones Application hôte et Paramètres d'exécution pour spécifier l'exécutable du serveur Web et les éventuels paramètres nécessaires à son exécution. Pour davantage d'informations sur ces valeurs, consultez la documentation de votre serveur Web.

Remarque Certains serveurs Web nécessitent d'autres modifications pour que vous ayez le droit de lancer l'application hôte de cette manière. Pour davantage d'informations, consultez la documentation de votre serveur Web.

Astuce Si vous utilisez Windows 2000 avec IIS 5, vous trouverez des détails sur toutes les modifications nécessaires pour configurer correctement vos droits à l'adresse suivante :

http://community.borland.com/article/0,1410,23024,00.html

Une fois renseignées les zones Application hôte et Paramètres d'exécution, vous pouvez définir vos points d'arrêt afin que lorsque le serveur passe un message de requête à votre DLL, un point d'arrêt soit activé et que vous puissiez effectuer le débogage normalement.

Remarque Avant de lancer le serveur Web avec les paramètres d'exécution de votre application, vérifiez qu'il n'est pas déjà ouvert.

Débogage sous Windows NT

Sous Windows NT, vous devez avoir certains droits d'utilisateur pour pouvoir déboguer une DLL. Dans le gestionnaire des utilisateurs, ajoutez votre nom aux listes accordant les droits pour :

- Ouvrir une session en tant que service
- Agir en tant que composante du système d'exploitation
- Générer des audits de sécurité

Débogage sous Windows 2000

Sous Windows 2000, vous accordez les même droits de la manière suivante :

1. Dans la zones Outils d'administration du panneau de configuration, choisissez Stratégie de sécurité locale. Développez Stratégies locales et sélectionnez Attribution des droits utilisateur. Dans le volet de droite, double-cliquez sur Agir en tant que partie du système d'exploitation.
2. Sélectionnez Ajouter pour ajouter un utilisateur à la liste, ajoutez votre compte utilisateur en cours.
3. Redémarrez pour que les modifications prennent effet.

Chapitre

28

Utilisation de l'agent Web

Les composants agent Web (placés dans l'onglet Internet de la palette des composants) vous permettent de créer des gestionnaires d'événements qui sont associés à une URL spécifique. Une fois le traitement achevé, il est possible de construire par programme des documents HTML ou XML et de les transférer au client. Vous pouvez utiliser les composants agent Web pour le développement d'applications multiplates-formes.

Souvent, le contenu des pages Web provient de bases de données. Vous pouvez utiliser des composants Internet pour gérer automatiquement les connexions aux bases de données, en permettant à une seule DLL de gérer simultanément plusieurs connexions de bases de données adaptées aux threads.

Les sections suivantes de ce chapitre expliquent comment utiliser les composants agent Web pour créer une application serveur Web.

Création d'applications serveur Web avec l'agent Web

Pour créer une nouvelle application serveur Web en utilisant l'architecture agent Web :

1 Sélectionnez Fichier | Nouveau | Autre.

2 Dans la boîte de dialogue Nouveaux éléments, choisissez l'onglet Nouveau et sélectionnez Application serveur Web.

3 Dans la boîte de dialogue qui s'affiche, vous pouvez sélectionner un type d'application serveur Web :

- ISAPI et NSAPI : la sélection de ce type d'application configure votre projet comme une DLL proposant les méthodes exportées attendues par le serveur Web. L'en-tête de la bibliothèque est également inclus dans le fichier projet et les entrées nécessaires dans les listes des clauses uses et exports du fichier projet.

- Apache : la sélection de ce type d'application configure votre projet comme une DLL proposant les méthodes exportées attendues par le serveur Web Apache. L'en-tête de la bibliothèque est également inclus dans le fichier projet et les entrées nécessaires dans les listes des clauses uses et exports du fichier projet.
- Exécutable autonome CGI : la sélection de ce type d'application configure votre projet comme une application en mode console et ajoute les entrées nécessaires à la clause uses du fichier projet.
- Exécutable autonome Win-CGI : si vous sélectionnez ce type d'application, le projet est configuré comme une application Windows et ajoute les entrées nécessaires à la clause uses du fichier projet.
- Exécutable débogueur d'application Web : la sélection de ce type d'application configure un environnement permettant de développer et de tester des applications serveur Web. Ce type d'application n'est pas conçu pour être déployé.

Choisissez le type d'application serveur Web qui communique avec le type de serveur Web que votre application utilisera. Ceci crée un nouveau projet configuré pour utiliser les composants Internet et contenant un module Web vide.

Module Web

Le module Web (*TWebModule*) est un descendant de *TDataModule* ; il s'utilise de la même façon pour offrir un contrôle centralisé pour les règles de gestion et les composants non visuels dans l'application Web.

Ajoutez tout générateur de contenu que votre application utilise pour générer les messages de réponse. Il peut s'agir de générateurs de contenu intégrés, tels que *TPageProducer*, *TDataSetPageProducer*, *TDataSetTableProducer*, *TQueryTableProducer* et *TInetXPageProducer* ou de descendants de *TCustomContentProducer* que vous avez créés vous-même. Si votre application génère des messages de réponse incluant des données extraites d'une base de données, vous pouvez ajouter des composants d'accès aux données ou des composants particuliers pour écrire un serveur Web qui fait office de client dans une application de bases de données multiniveaux.

Le module Web ne se contente pas de stocker les composants non visuels et des règles de gestion : il fait aussi office de répartiteur en associant les messages de requête HTTP reçus aux éléments d'action qui génèrent les réponses à ces requêtes.

Vous avez peut-être déjà un module de données paramétré avec les composants non visuels et les règles de gestion que vous souhaitez utiliser dans votre application Web. Vous pouvez alors remplacer le module Web par ce module de données : il suffit de supprimer le module Web généré automatiquement et de le remplacer par votre module de données. Ajoutez ensuite un composant *TWebDispatcher* à votre module de données pour qu'il puisse répartir les messages de requête vers les éléments d'action, comme le ferait un module Web.

Si vous voulez modifier la façon dont les éléments d'action sont choisis pour répondre aux messages de requête HTTP reçus, dérivez un nouveau composant répartiteur de *TCustomWebDispatcher* et ajoutez-le au module de données.

Votre projet ne peut contenir qu'un répartiteur. Il peut s'agir soit du module Web automatiquement généré lorsque vous créez le projet, soit du composant *TWebDispatcher* que vous ajoutez au module de données qui remplace le module Web. Si un second module de données contenant un répartiteur est créé lors de l'exécution, l'application serveur Web générera une erreur.

Remarque Le module Web que vous paramétrez en phase de conception est en fait un modèle. Dans les applications ISAPI et NSAPI, chaque message de requête crée un thread distinct. Une instance du module Web et de son contenu est créée dynamiquement pour chaque thread.

Attention Le module Web d'une application serveur Web à base de DLL est mis en mémoire cache pour une utilisation ultérieure afin d'améliorer les temps de réponse. L'état du répartiteur et sa liste d'actions ne sont pas réinitialisés entre deux requêtes. Si vous activez ou désactivez des éléments d'action en cours d'exécution, vous obtiendrez des résultats imprévisibles lorsque ce module sera utilisé pour les requêtes client suivantes.

Objet application Web

Le projet préparé pour votre application Web contient une variable globale du nom de *Application*. *Application* est un descendant de *TWebApplication* (soit *TISAPIApplication*, soit *TCGIApplication*) approprié pour le type d'application que vous créez. Il s'exécute en réponse aux messages de requête HTTP reçus par le serveur Web.

Attention N'incluez pas l'unité forms dans la clause **uses** du projet après l'unité CGIApp ou ISAPIApp. Forms déclare en effet une autre variable globale du nom d'*Application* et, si elle apparaît après l'unité CGIApp ou ISAPIApp, *Application* sera initialisée comme un objet de type erroné.

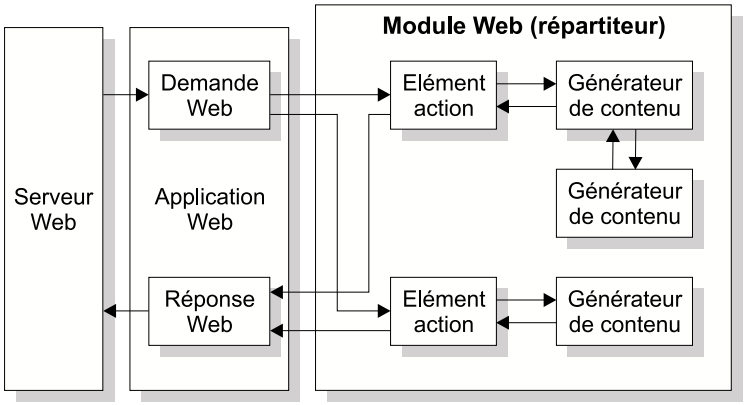
Structure d'une application agent Web

Lorsque l'application Web reçoit un message de requête HTTP, elle crée un objet *TWebRequest* pour représenter le message de requête HTTP et un objet *TWebResponse* pour représenter la réponse qui doit être retournée. L'application transmet ensuite ces objets au répartiteur Web (le module Web ou un composant *TWebDispatcher*).

Le répartiteur Web contrôle le déroulement de l'application serveur Web. Il gère un ensemble d'éléments d'action (*TWebActionItem*) qui sait comment réagir à certains messages de requête HTTP. Le répartiteur identifie les éléments d'action ou les composants à répartition automatique aptes à répondre au message de requête HTTP et transmet les objets requête et réponse au gestionnaire identifié pour qu'il lance les opérations demandées ou formule un message de réponse.

Le répartiteur est décrit plus en détail dans la section “Répartiteur Web” à la page 28-4.

Figure 28.1 Structure d’une application serveur



Les éléments d’action gèrent la lecture de la requête et l’assemblage d’un message de réponseLes composants générateur de contenu spécialisés aident les éléments d’action à générer dynamiquement le contenu des réponses de messagepouvant inclure du code HTML personnalisé ou un autre contenu MIME. Les générateurs de contenu se servent d’autres générateurs de contenu ou descendants de *THTMLTagAttributes*, pour créer le contenu du message de réponse. Pour plus d’informations sur les générateurs de contenu, voir “Génération du contenu des messages de réponse” à la page 28-14.

Si vous créez le client Web dans une application de base de données multiniveau, votre application serveur Web peut inclure d’autres composants à répartition automatique qui représentent des informations de bases de données codées en XML et des classes de manipulation de base de données codées en javascript. Si vous créez un serveur qui implémente un service Web, votre application serveur Web peut contenir un composant d’auto-répartition qui transmet les messages utilisant le protocole SOAP à un exécutant qui les interprète et les exécute. Le répartiteur fait appel à ces composants à répartition automatique pour traiter le message de requête après avoir essayé tous ses éléments d’action.

Lorsque tous les éléments d’action (ou composants à répartition automatique) ont créé leur réponse en remplissant l’objet *TWebResponse*, le répartiteur transmet le résultat à l’application Web qui le transmet en retour au client via le serveur Web.

Répartiteur Web

Si vous utilisez un module Web, il agit comme un répartiteur Web. Si vous utilisez un module de données existant, vous devez lui ajouter un composant (*TWebDispatcher*). Le répartiteur gère un ensemble d’éléments d’action qui savent

comment répondre à certains types de messages de requête. Lorsque l'application Web transmet un objet requête et un objet réponse au répartiteur, il choisit un ou plusieurs éléments d'action pour répondre à la requête.

Ajout d'actions au répartiteur

Ouvrez l'éditeur d'action depuis l'inspecteur d'objets en cliquant sur les points de suspension de la propriété *Actions* du répartiteur. Il est possible d'ajouter des éléments d'action au répartiteur en cliquant sur le bouton Ajouter dans l'éditeur d'action.

Ajoutez des actions au répartiteur pour répondre aux différentes méthodes de requête ou URI de destination. Vous pouvez paramétrer vos éléments d'action de diverses façons. Vous pouvez commencer par les éléments d'action qui font un prétraitement des requêtes et terminer par une action par défaut qui vérifie que la réponse est complète et l'envoie ou retourne un code d'erreur. Vous pouvez aussi ajouter un élément d'action pour chaque type de requête, auquel cas chaque élément d'action traitera entièrement la requête.

Les éléments d'action sont décrits en détail à la section "Eléments d'action" à la page 28-6.

Répartition des messages de requête

Lorsque le répartiteur reçoit la requête client, il génère un événement *BeforeDispatch*. Ceci permet à votre application de faire un prétraitement du message de requête avant qu'il ne soit vu par les éléments d'action.

Ensuite, le répartiteur recherche dans sa liste d'éléments d'action un élément qui corresponde à la partie chemin de l'URL de destination du message de requête et sachant fournir le service spécifié comme méthode du message de requête. Pour ce faire, il compare les propriétés *PathInfo* et *MethodType* de l'objet *TWebRequest* avec ces mêmes propriétés dans l'élément d'action.

Lorsque le répartiteur trouve l'élément d'action recherché, il déclenche cet élément d'action, qui effectue l'une des opérations suivantes :

- Il fournit le contenu de la réponse et envoie la réponse ou signale que la requête est complètement traitée.
- Il ajoute des informations à la réponse puis permet à d'autres éléments d'action de finir le travail.
- Il transmet la requête à d'autres éléments d'action.

Après avoir vérifié tous ses éléments d'action, si le message n'est pas traité, le répartiteur vérifie tous les composants à répartition automatique spécialement recensés n'utilisant pas d'élément d'action. Ces composants sont propres aux applications de bases de données multiniveaux, décrites dans la section "Construction des applications Web avec InternetExpress" à la page 25-38.

Si, après contrôle de tous les éléments d'action et des composants à répartition automatique spécialement recensés, le message de requête n'est pas traité, le répartiteur appelle l'élément d'action par défaut. Il n'est pas nécessaire que cet élément d'action corresponde à l'URL de destination ou à la méthode de requête.

Si le répartiteur atteint la fin de la liste d'actions (y compris l'éventuelle action par défaut) et qu'aucune action n'a été déclenchée, rien n'est retourné au serveur. Le serveur met alors fin à la connexion avec le client.

Si la requête est traitée par les éléments d'action, le répartiteur génère un événement *AfterDispatch*. Ceci représente la dernière possibilité pour votre application de vérifier que la réponse a été générée et de faire les modifications requises.

Eléments d'action

Chaque élément d'action (*TWebActionItem*) effectue une tâche précise en réponse à un type spécifique de message de requête.

Les éléments d'action peuvent répondre entièrement à une requête ou n'y répondre que partiellement et laisser d'autres éléments d'action finir le travail. Les éléments d'action peuvent envoyer le message de réponse HTTP pour la requête, ou simplement préparer une partie de la réponse, que d'autres éléments d'action termineront. Si une réponse est complétée par les éléments d'action mais non envoyée, l'application serveur Web envoie le message de réponse.

Choix du déclenchement des éléments d'action

La plupart des propriétés d'un élément d'action indiquent le moment auquel le répartiteur doit le sélectionner pour gérer un message de requête HTTP. Pour définir les propriétés d'un élément d'action, vous devez d'abord ouvrir l'éditeur d'action : sélectionnez la propriété Actions du répartiteur dans l'inspecteur d'objets et cliquez sur les points de suspension. Lorsqu'une action est sélectionnée dans l'éditeur, ses propriétés peuvent être modifiées dans l'inspecteur d'objets.

URL de destination

Le répartiteur compare la valeur de la propriété *PathInfo* d'un élément d'action à celle de la propriété *PathInfo* du message de requête. La valeur de cette propriété doit être le chemin d'accès à l'URL pour toutes les requêtes que l'élément d'action est prêt à gérer. Par exemple dans cette URL

```
http://www.TSite.com/art/gallery.dll/mammals?animal=dog&color=black
```

si la partie `/gallery.dll` indique l'application serveur Web, le chemin d'accès est

```
/mammals
```

Utilisez les informations de chemin d'accès pour indiquer où votre application Web doit rechercher des informations au moment de traiter des requêtes ou pour subdiviser votre serveur Web en sous-services logiques.

Type de méthode de requête

La propriété *MethodType* d'un élément d'action indique quels types de messages de requête il peut traiter. Le répartiteur compare la propriété *MethodType* d'un élément d'action à la propriété *MethodType* du message de requête. *MethodType* peut avoir les valeurs suivantes :

Tableau 28.1 Valeurs de MethodType

Valeur	Signification
mtGet	La requête demande que les informations associées à l'URI de destination soient retournées dans un message de réponse.
mtHead	La requête demande les propriétés d'en-tête d'une réponse, comme dans le cas du traitement d'une requête *mtGet*, mais omet le contenu de la réponse.
mtPost	La requête fournit les informations à émettre dans l'application Web.
mtPut	La requête demande que la ressource associée à l'URI de destination soit remplacée par le contenu du message de requête.
mtAny	L'élément d'action correspond à tout type de méthode de requête, y compris *mtGet*, *mtHead*, *mtPut* et *mtPost*.

Activation et désactivation des éléments d'action

Chaque élément d'action possède une propriété *Enabled* qui permet de l'activer et de le désactiver. En mettant *Enabled* à *False*, vous désactivez l'élément d'action, qui n'est plus pris en compte par le répartiteur lors de la recherche d'un élément d'action capable de gérer une requête.

Un gestionnaire d'événement *BeforeDispatch* peut définir quels éléments d'action doivent traiter une requête en modifiant la propriété *Enabled* des éléments d'action avant que le répartiteur ne commence sa recherche.

Attention Si vous modifiez la propriété *Enabled* d'une action lors de l'exécution, vous risquez d'obtenir des résultats imprévisibles pour les requêtes suivantes. Si l'application serveur Web est une DLL qui met en cache les modules Web, l'état initial ne sera pas réinitialisé pour la requête suivante. Utilisez l'événement *BeforeDispatch* pour vous assurer que tous les éléments d'action sont correctement initialisés.

Choix d'un élément d'action par défaut

Un seul élément d'action peut être l'élément d'action par défaut. L'élément d'action par défaut est choisi en mettant sa propriété *Default* à *True*. Lorsque vous mettez la propriété *Default* d'un élément d'action à *True*, la propriété *Default* du précédent élément d'action par défaut (s'il en existait un) passe à *False*.

Lorsque le répartiteur recherche dans sa liste d'éléments d'action celui qui peut gérer la requête, il conserve en mémoire le nom de l'élément d'action par défaut.

Si la requête n'a pas été entièrement honorée lorsque le répartiteur atteint la fin de la liste d'éléments d'action, il exécute l'élément d'action par défaut.

Le répartiteur ne vérifie pas les propriétés *PathInfo* et *MethodType* de l'élément d'action par défaut. Il ne vérifie pas non plus la propriété *Enabled* de l'élément d'action par défaut. Vous pouvez ainsi vous assurer que l'élément d'action par défaut n'est appelé qu'en dernier recours en mettant sa propriété *Enabled* à *False*.

L'élément d'action par défaut doit être prêt à gérer toute requête détectée, même si ce n'est qu'en retournant un code d'erreur signalant une URI ou une valeur de propriété *MethodType* non valide. Si l'élément d'action par défaut ne peut pas traiter la requête, aucune réponse n'est transmise au client Web.

Attention Si vous modifiez la propriété *Default* d'une action lors de l'exécution, vous risquez d'obtenir des résultats imprévisibles pour la requête active. Si la propriété *Default* d'une action ayant déjà été déclenchée passe à *True*, cette action ne sera pas réévaluée et le répartiteur ne la déclenchera pas lorsqu'il atteindra la fin de la liste d'actions.

Réponse aux messages de requête avec des éléments d'action

Le plus gros du travail de l'application serveur Web est effectué par les éléments d'action lors de leur exécution. Lorsque le répartiteur Web déclenche un élément d'action, ce dernier peut répondre au message de requête en cours de deux manières :

- Si un composant générateur est associé à l'élément d'action comme valeur de sa propriété *Producer*, ce générateur affecte automatiquement le contenu (*Content*) du message de réponse à l'aide de sa méthode *Content*. La page Internet de la palette de composants inclut une série de composants générateur de contenu qui permettent d'élaborer une page HTML pour le contenu du message de réponse.
- Une fois que le générateur a affecté un contenu de réponse (si un générateur est associé), l'élément d'action reçoit un événement *OnAction*. Le gestionnaire d'événement *OnAction* reçoit l'objet *TWebRequest* qui représente le message de requête HTTP et un objet *TWebResponse* permettant de remplir les informations de réponse.

Si le contenu de l'élément d'action peut être généré par un seul générateur de contenu, il est plus simple d'affecter le générateur de contenu comme valeur de la propriété *Producer* de l'élément d'action. Toutefois, tout générateur de contenu demeure accessible à partir du gestionnaire d'événement *OnAction*. Le gestionnaire d'événement *OnAction* offre une souplesse supplémentaire, qui vous permet d'utiliser plusieurs générateurs de contenu, d'affecter des propriétés de message de réponse, etc.

Le composant générateur de contenu et le gestionnaire d'événement *OnAction* peuvent utiliser tous les objets et toutes les méthodes de bibliothèque d'exécution pour répondre aux messages de requête. Ils peuvent accéder à des bases de données, faire des calculs, construire ou sélectionner des documents HTML, etc. Pour plus d'informations sur la génération de contenu de réponse à l'aide de

composants générateur de contenu, voir “Génération du contenu des messages de réponse” à la page 28-14.

Envoi de la réponse

Un gestionnaire d’événement *OnAction* peut renvoyer la réponse au client Web à l’aide des méthodes de l’objet *TWebResponse*. Cependant, si aucun élément d’action n’envoie la réponse au client, elle sera envoyée par l’application serveur Web si le dernier élément d’action ayant analysé la requête indique que la requête a été traitée.

Utilisation de plusieurs éléments d’action

Vous pouvez répondre à une requête depuis un seul élément d’action ou répartir le travail entre plusieurs éléments d’action. Si l’élément d’action n’élabore pas complètement le message de réponse, il doit indiquer cet état dans le gestionnaire d’évément *OnAction* en mettant le paramètre *Handled* à *False*.

Si de nombreux éléments d’action se partagent la tâche de répondre aux messages de requête et que chacun met *Handled* à *False*, pensez à vérifier que l’élément d’action par défaut laisse le paramètre *Handled* à *True*, faute de quoi aucune réponse ne serait envoyée au client Web.

Lorsque plusieurs éléments d’action se partagent la gestion du message de requête, il faut que le gestionnaire d’événement *OnAction* de l’élément d’action par défaut ou que le gestionnaire d’événement *AfterDispatch* du répartiteur vérifie que toutes les tâches ont bien été effectuées. Si ce n’est pas le cas, il devra définir le code d’erreur adéquat.

Accès aux informations de requêtes client

Lorsqu’un message de requête HTTP est reçu par l’application serveur Web, les en-têtes de la requête client sont chargés dans les propriétés d’un objet *TWebRequest*. Dans les applications NSAPI et ISAPI, le message de requête est encapsulé par un objet *TISAPIRequest*. Les applications Console CGI utilisent les objets *TCGIRequest*, alors que les applications Windows CGI utilisent les objets *TWinCGIRequest*.

Les propriétés de l’objet requête sont en lecture seule. Vous pouvez les utiliser pour recueillir toutes les informations disponibles dans la requête client.

Propriétés contenant des informations d’en-tête de requête

La plupart des propriétés d’un objet requête contiennent des informations sur la requête provenant de l’en-tête de requête HTTP. Toutes les requêtes, cependant, ne fournissent pas une valeur pour toutes les propriétés. De plus, certaines requêtes peuvent inclure des champs d’en-tête qui n’apparaissent pas dans une propriété de l’objet requête, car le standard HTTP est en évolution permanente. Pour obtenir la valeur d’un champ d’en-tête de requête qui n’apparaît pas

comme l'une des propriétés de l'objet requête, utilisez la méthode *GetFieldByName*.

Propriétés identifiant la destination

La destination complète du message de requête est fournie par la propriété *URL*. En règle générale, il s'agit d'une URL qui peut se décomposer en un protocole (HTTP), un *Host* (système serveur), un *ScriptName* (application serveur), un *PathInfo* (emplacement sur l'hôte) et un *Query*.

Chacune de ces parties est reflétée dans sa propre propriété. Le protocole est toujours HTTP, *Host* et *ScriptName* identifient l'application serveur Web. Le répartiteur utilise la partie *PathInfo* lorsqu'il associe des éléments d'action aux messages de requête. La partie *Query* est utilisée par certaines requêtes pour spécifier des détails sur les informations demandées. Sa valeur est elle aussi analysée pour vous dans la propriété *QueryFields*.

Propriétés décrivant le client Web

La requête inclut en outre plusieurs propriétés qui fournissent des informations sur son origine. Il peut s'agir de l'adresse e-mail de l'expéditeur (propriété *From*) ou de l'URI d'origine du message (propriété *Referer* ou *RemoteHost*). Si la requête a un contenu et que ce contenu ne provient pas du même URI que la requête, la source de ce contenu est indiquée par la propriété *DerivedFrom*. Vous pouvez également connaître l'adresse IP du client (propriété *RemoteAddr*) et le nom et la version de l'application ayant envoyé la requête (propriété *UserAgent*).

Propriétés identifiant le but de la requête

La propriété *Method* est une chaîne décrivant ce que le message de requête demande à l'application serveur. Le standard HTTP 1.1 définit les méthodes suivantes :

Valeur	Informations demandées
OPTIONS	Informations sur les options de communication disponibles.
GET	Informations identifiées par la propriété *URL*.
HEAD	Informations d'en-tête issues d'un message GET équivalent, sans le contenu de la réponse.
POST	L'application serveur doit émettre les données incluses dans la propriété *Content*, de la façon appropriée.
PUT	L'application serveur doit remplacer la ressource indiquée par la propriété *URL* par les données de la propriété *Content*.
DELETE	L'application serveur doit supprimer ou masquer la ressource identifiée par la propriété *URL*.
TRACE	L'application serveur doit confirmer la réception de la requête.

La propriété *Method* peut indiquer toute autre méthode que le client Web demande au serveur.

Il n'est pas nécessaire que l'application serveur Web fournisse une réponse pour toutes les valeurs possibles de la propriété *Method*. Le standard HTTP exige cependant qu'elle sache répondre aux requêtes GET et HEAD.

La propriété *MethodType* indique si la valeur de *Method* est GET (mtGet), HEAD (mtHead), POST (mtPost), PUT (mtPut) ou une autre chaîne (mtAny). Le répartiteur fait correspondre la valeur de la propriété *MethodType* avec celle de la propriété *MethodType* de tous les éléments d'action.

Propriétés décrivant la réponse attendue

La propriété *Accept* indique les types de support que le client Web accepte comme contenu du message de réponse. La propriété *IfModifiedSince* indique si le client ne souhaite que les informations modifiées récemment. La propriété *Cookie* inclut des informations d'état (généralement ajoutées par l'application) qui peuvent modifier la réponse.

Propriétés décrivant le contenu

La plupart des requêtes n'incluent aucun contenu car elles ne font que demander des informations. Certaines requêtes, cependant, telles que les requêtes POST, fournissent un contenu que l'application serveur Web doit utiliser. Le type de support du contenu est précisé dans la propriété *ContentType* et sa longueur l'est dans la propriété *ContentLength*. Si le contenu du message a été codé (par compression des données, par exemple), cette information figure dans la propriété *ContentEncoding*. Le nom et le numéro de version de l'application ayant généré le contenu sont indiqués dans la propriété *ContentVersion*. La propriété *Title* fournit elle aussi, parfois, des renseignements sur le contenu.

Contenu d'un message de requête HTTP

En plus des champs d'en-tête, certains messages de requête incluent une partie contenu que l'application serveur Web doit traiter. Par exemple, une requête POST peut comporter des informations qui doivent être ajoutées à une base de données gérée par l'application serveur Web.

La valeur non traitée du contenu est fournie par la propriété *Content*. Si le contenu peut être réparti dans des champs séparés par le caractère "&", cette version sera disponible dans la propriété *ContentFields*.

Création de messages de réponse HTTP

Lorsque l'application serveur Web crée un objet *TWebRequest* pour un message de requête HTTP reçu, elle crée aussi un objet *TWebResponse* correspondant pour représenter le message de réponse qui sera envoyé en retour. Dans les applications NSAPI et ISAPI, le message de réponse est encapsulé par un objet *TISAPIResponse*. Les applications GCI Console utilisent les objets *TCGIResponse* et les applications Windows CGI utilisent les objets *TWinCGIResponse*.

Les éléments d'action qui génèrent la réponse à la requête client renseignent les propriétés de l'objet réponse. Dans certains cas, il suffit de retourner un code d'erreur ou de passer la requête à une autre URI. Mais dans d'autres cas, il faut parfois effectuer des calculs tels que l'élément d'action doit aller chercher des informations à d'autres sources puis les regrouper avant de les présenter au format final. La plupart des messages de requête nécessitent une réponse, même s'il ne s'agit que d'indiquer au client que l'opération demandée a été effectuée.

Informations d'en-tête de réponse

La plupart des propriétés de l'objet *TWebResponse* représentent les informations d'en-tête du message de réponse HTTP retourné au client Web. Un élément d'action définit ces propriétés à partir de son gestionnaire d'événement *OnAction*.

Il n'est pas nécessaire que tous les messages de réponse contiennent une valeur pour toutes les propriétés de l'en-tête. Les propriétés qui doivent être renseignées dépendent de la nature de la requête et du statut de la réponse.

Indication du statut de la réponse

Tout message de réponse doit inclure un code indiquant le statut de la réponse. Vous pouvez spécifier ce code en définissant la propriété *StatusCode*. Le standard HTTP définit des codes de statut à la signification prédéfinie. De plus, vous pouvez définir vos propres codes de statut avec les valeurs possibles inutilisées.

Un code de statut est un numéro à trois chiffres dans lequel le chiffre le plus significatif indique la classe de la réponse, de la façon suivante :

- 1xx : Information (la requête a été reçue mais n'a pas été entièrement traitée).
- 2xx : Succès (la requête a été reçue, comprise et acceptée).
- 3xx : Redirection (le client doit intervenir pour compléter la requête).
- 4xx : Erreur du client (la requête est incompréhensible ou ne peut être traitée).
- 5xx : Erreur du serveur (la requête est valide mais le serveur n'a pas pu la traiter).

Une chaîne est associée à chaque code de statut ; elle donne la signification de ce code de statut. Elle se trouve dans la propriété *ReasonString*. Pour les codes de statut prédéfinis, il n'est pas nécessaire de définir la propriété *ReasonString*. Si vous créez vos propres codes de statut, cependant, pensez à définir la propriété *ReasonString*.

Indication d'attente d'une action du client

Lorsque le code de statut est compris entre 300 et 399, le client doit lancer une action pour que l'application serveur Web puisse traiter la requête en entier. Si vous devez rediriger le client vers une autre URI, ou indiquer qu'une nouvelle URI a été créée pour traiter cette requête, utilisez la propriété *Location*. Si le client doit fournir un mot de passe pour poursuivre, définissez la propriété *WWWAuthenticate*.

Description de l'application serveur

Certaines propriétés d'en-tête de réponse décrivent les capacités de l'application serveur Web. La propriété *Allow* indique les méthodes auxquelles elle peut répondre. La propriété *Server* contient le nom et la version de l'application servant à générer la réponse. La propriété *Cookies* peut contenir des informations d'état concernant l'utilisation par le client de l'application serveur qui sont incluses dans les messages de requête ultérieurs.

Description du contenu

Plusieurs propriétés décrivent le contenu de la réponse. *ContentType* fournit le type de support de la réponse, *ContentVersion* le numéro de version de ce support. *ContentLength* indique la longueur de la réponse. Si le contenu est codé (par compression des données, par exemple), indiquez-le dans la propriété *ContentEncoding*. Si le contenu provient d'une autre URI, indiquez-le dans la propriété *DerivedFrom*. Si la valeur du contenu tient compte de la date, utilisez les propriétés *LastModified* et *Expires* pour indiquer si le contenu est toujours valide. La propriété *Title* peut fournir des informations descriptives sur le contenu.

Définition du contenu de la réponse

Dans certains cas, la réponse au message de requête est entièrement contenue dans les propriétés d'en-tête de la réponse. La plupart du temps, cependant, un élément d'action assigne un contenu au message de réponse. Ce contenu peut être des informations statiques stockées dans un fichier ou des informations générées par l'élément d'action ou son générateur de contenu.

Vous pouvez définir le contenu du message de réponse à l'aide des propriétés *Content* et *ContentStream*.

La propriété *Content* est une chaîne. Les chaînes Delphi ne sont pas limitées à des valeurs littérales, aussi la valeur de la propriété *Content* peut-elle être une série de commandes HTML, un contenu graphique ou tout type de contenu MIME.

Utilisez la propriété *ContentStream* si le contenu du message de réponse peut être lu dans un flux. Par exemple, si le message de réponse doit envoyer le contenu d'un fichier, utilisez un objet *TFileStream* pour la propriété *ContentStream*. Comme avec la propriété *Content*, *ContentStream* peut fournir une chaîne de commandes HTML ou un autre contenu de type MIME. Si vous utilisez la propriété *ContentStream*, ne libérez pas le flux vous-même : l'objet réponse Web le libérera automatiquement.

Remarque Si la valeur de la propriété *ContentStream* n'est pas **nil**, la propriété *Content* est ignorée.

Envoi de la réponse

Si vous êtes sûr que le traitement du message de requête est terminé, vous pouvez envoyer une réponse directement depuis le gestionnaire d'événement*OnAction*. L'objet réponse offre deux méthodes d'envoi de réponses : *SendResponse* et *SendRedirect*. Appelez *SendResponse* pour envoyer une réponse avec le contenu et les propriétés d'en-tête de l'objet *TWebResponse*. Si votre action doit se limiter à orienter le client Web vers une autre URI, utilisez la méthode *SendRedirect*, plus efficace.

Si aucun des gestionnaires d'événements n'envoie de réponse, l'application Web l'envoie lorsque le répartiteur se referme. Cependant, si aucun élément d'action n'indique qu'il a traité la réponse, l'application ferme la connexion au client Web sans envoyer de réponse.

Génération du contenu des messages de réponse

Delphi met à votre disposition plusieurs objets qui aideront vos éléments d'action à générer un contenu pour les messages de réponse HTTP. Vous pouvez utiliser ces objets pour générer des chaînes de commandes HTML enregistrées dans un fichier ou transmises directement au client Web. Vous pouvez créer vos propres générateurs de contenu en les dérivant de *TCustomContentProducer* ou de l'un de ses descendants.

TCustomContentProducer offre une interface générique pour la création de contenus de type MIME dans un message de réponse HTTP. Ses descendants incluent des générateurs de page et des générateurs de tableau :

- Les générateurs de page peuvent rechercher dans les documents HTML des balises spéciales qu'ils remplacent par du code HTML spécifique. Ils sont décrits dans la section suivante.
- Les générateurs de tableau créent des commandes HTML à partir des informations contenues dans un ensemble de données. Ils sont décrits dans "Utilisation des bases de données dans les réponses" à la page 28-18.

Utilisation du composant générateur de page

Les générateurs de page (*TPageProducer* et des descendants*)* convertissent un modèle HTML en remplaçant les balises HTML transparentes par du code HTML personnalisé. Vous pouvez garder sous la main un jeu de modèles de réponses qui seront remplis par les générateurs de page lorsque vous devrez répondre à un message de requête HTTP. Vous pouvez chaîner des générateurs de page pour construire de façon itérative un document HTML par traitements successifs des balises HTML transparentes.

Modèles HTML

Un modèle HTML est une suite de commandes HTML et de balises HTML transparentes. Une balise HTML transparente est au format :

```
<#Nom_de_balise Param1=Valeur1 Param2=Valeur2 ...>
```

Les crochets (< et >) définissent la portée de la balise. Le signe "#" vient immédiatement après le crochet ouvrant, sans espace entre les deux. Il indique au générateur de page que la chaîne qui suit est une balise HTML transparente. Le nom de la balise suit immédiatement le signe dièse, sans espace entre les deux. Le nom de la balise peut être tout identificateur valide ; il identifie le type de conversion représenté par la balise.

Après un nom de balise, une balise HTML transparente peut parfois comporter des paramètres fournissant des informations sur la conversion à effectuer. Chaque paramètre est au format *Paramètre=Valeur*. Aucun espace ne doit figurer entre le nom du paramètre, le signe égal et la valeur. Les paramètres sont séparés par des espaces.

Les crochets (< et >) rendent la balise transparente pour les navigateurs HTML qui ne reconnaissent pas la syntaxe "#Nom_de_balise".

Bien que vous puissiez créer vos propres balises HTML transparentes pour représenter tout type d'informations traitées par votre générateur de page, plusieurs noms de balises associés à des valeurs du type de données *TTag* sont mis à votre disposition. Ces noms de balise prédéfinis corrrespondent aux commandes HTML susceptibles de varier d'un message de réponse à l'autre. Ils sont décrits dans le tableau suivant :

Nom de la balise	Valeur de TTag	La balise est convertie en :
Link	*tgLink*	Lien hypertexte. Le résultat est une séquence HTML commençant par la balise <A> et se terminant par </A>.
Image	*tgImage*	Graphique. Le résultat est une balise HTML <IMG>.
Table	*tgTable*	Tableau HTML. Le résultat est une séquence HTML commençant par la balise <TABLE> et se terminant par la balise </TABLE>.
ImageMap	*tgImageMap*	Graphique contenant des zones sensibles. Le résultat est une séquence HTML commençant par la balise <MAP> et se terminant par </MAP>.
Object	*tgObject*	Objet ActiveX incorporé. Le résultat est une séquence HTML commençant par la balise <OBJECT> et se terminant par la balise </OBJECT>.
Embed	*tgEmbed*	DLL compatible Netscape. Le résultat est une séquence HTML commençant par la balise <EMBED> et se terminant par la balise </EMBED>.

Tout autre nom de balise est associé à *tgCustom*. Le générateur de page n'offre aucun traitement spécifique des noms de balises prédéfinis. Ils sont simplement fournis pour aider vos applications à structurer le processus de conversion des tâches les plus fréquentes.

Remarque Les noms de balises prédéfinis tiennent compte de la différence entre majuscules et minuscules.

Choix du modèle HTML

Les générateurs de page vous proposent plusieurs façons pour désigner le modèle HTML. Vous pouvez donner à la propriété *HTMLFile* le nom du fichier contenant le modèle HTML. Vous pouvez donner à la propriété *HTMLDoc* le nom d'un objet *TStrings* contenant le modèle HTML. Si vous utilisez l'une de ces propriétés pour désigner le modèle, vous pouvez générer les commandes HTML converties en appelant la méthode *Content*.

Vous pouvez également appeler la méthode *ContentFromString* pour convertir directement un modèle HTML composé d'une seule chaîne passée dans un paramètre, ou appeler la méthode *ContentFromStream* pour lire le modèle HTML depuis un flux. Ainsi, par exemple, vous pourriez placer vos modèles HTML dans un champ mémo d'une base de données et, à l'aide de la méthode *ContentFromStream*, obtenir les commandes HTML converties en lisant le modèle depuis un objet *TBlobStream*.

Conversion des balises HTML transparentes

Le générateur de page convertit le modèle HTML lorsque vous appelez l'une de ses méthodes *Content*. Lorsque la méthode *Content* détecte une balise HTML transparente, elle déclenche un événement *OnHTMLTag*. Vous devez écrire un gestionnaire d'événement pour définir le type de balise détecté et la remplacer par un contenu personnalisé.

Si vous ne créez pas de gestionnaire d'événement *OnHTMLTag* pour le générateur de page, les balises HTML transparentes sont remplacées par des chaînes vides.

Utilisation du générateur de page depuis un élément d'action

Un exemple d'utilisation d'un composant générateur de page est d'utiliser la propriété *HTMLFile* pour spécifier un fichier contenant un modèle HTML. Le gestionnaire d'événement *OnAction* appelle la méthode *Content* pour convertir le modèle en séquence HTML finale :

```
procedure WebModule1.MyActionEventHandler(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
  PageProducer1.HTMLFile := 'Accueil.html';
  Response.Content := PageProducer1.Content;
end;
```

Accueil.html est un fichier contenant ce modèle HTML :

```
<HTML>
<HEAD><TITLE>Notre nouveau site Web</TITLE></HEAD>
<BODY>
Hello <#UserName>! Bienvenue sur notre site.
</BODY>
</HTML>
```

Le gestionnaire d'événement *OnHTMLTag* remplace la balise personnalisée (<#UserName>) dans le code HTML lors de l'exécution :

```
procedure WebModule1.PageProducer1HTMLTag(Sender : TObject;Tag: TTag;
   const TagString: string; TagParams: TStrings; var ReplaceText: string);
begin
  if CompareText(TagString,'UserName') = 0 then
    ReplaceText := TPageProducer(Sender).Dispatcher.Request.Content;
end;
```

Si le contenu du message de requête était la chaîne *Alain*, la valeur de *Response.Content* sera

```
<HTML>
<HEAD><TITLE>Notre nouveau site Web</TITLE></HEAD>
<BODY>
Bonjour Alain ! Bienvenue sur notre site.
</BODY>
</HTML>
```

Remarque Cet exemple utilise un gestionnaire d'événement *OnAction* pour appeler le générateur de contenu et assigner le contenu de message de réponse. Vous n'avez pas besoin d'écrire un gestionnaire d'événement *OnAction* si vous assignez la propriété *HTMLFile* du générateur de page lors de la conception. Dans ce cas, vous pouvez simplement assigner *PageProducer1* comme valeur de la propriété *Producer* de l'élément d'action pour obtenir le même effet que le gestionnaire d'événement *OnAction* ci-dessus.

Chaînage de générateurs de page

Le texte de substitution venant du gestionnaire d'événement *OnHTMLTag* ne doit pas nécessairement être la séquence HTML finale que vous voulez utiliser dans le message de réponse HTTP. Vous pouvez utiliser plusieurs générateurs de page, auquel cas le résultat de l'un sera passé au suivant.

La façon la plus simple de chaîner des générateurs de page consiste à associer chaque générateur de page à un élément d'action particulier, tous les éléments d'action disposant des mêmes *PathInfo* et *MethodType*. Le premier élément d'action définit le contenu du message de réponse Web à partir de son générateur de contenu, mais son gestionnaire d'événement *OnAction* veille à ce que le message ne soit pas considéré comme traité. L'élément d'action suivant utilise la méthode *ContentFromString* de son générateur associé pour manipuler le contenu du message de réponse Web, etc. Les éléments d'action qui suivent le premier élément d'action utilisent un gestionnaire d'événement comme suit :

```
procedure WebModule1.Action2Action(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageProducer2.ContentFromString(Response.Content);
end;
```

Prenons comme exemple une application qui affiche des pages de calendrier en réponse à des messages de requête indiquant le mois et l'année à afficher. Chaque page de calendrier contient une image suivie de l'année et du mois,

placés entre des images des mois précédent et suivant, puis du calendrier lui-même. L'image finale aurait l'aspect suivant :

Le format général du calendrier réside dans un fichier de modèle à l'aspect suivant :

```
<HTML>
<Head></HEAD>
<BODY>
<#MonthlyImage> <#TitleLine><#MainBody>
</BODY>
</HTML>
```

Le gestionnaire d'événement *OnHTMLTag* du premier générateur de page recherche le mois et l'année dans le message de requête. Au moyen de ces informations et du fichier de modèle, il :

- Remplace <#MonthlyImage> par <#Image Month=Janvier Year=1997>.
- Remplace <#TitleLine> par <#Calendar Month=Décembre Year=1996 Size=Small> Janvier 1997 <#Calendar Month=Février Year=1997 Size=Small>.
- Remplace <#MainBody> par <#Calendar Month=Janvier Year=1997 Size=Large>.

Le gestionnaire d'événement *OnHTMLTag* du générateur de page suivant utilise le contenu créé par le premier générateur de page et remplace la balise <#Image Month=Janvier Year=1997> par la balise HTML <IMG> appropriée. Un troisième générateur de page convertit les balises #Calendar en tableaux HTML.

Utilisation des bases de données dans les réponses

La réponse à un message de requête HTTP peut inclure des informations extraites d'une base de données. Les générateurs de contenu spécialisés de la page Internet sont capables de générer du code HTML pour représenter les enregistrements d'une base de données dans un tableau HTML.

Sous une approche différente, les composants spéciaux de la page InternetExpress de la palette de composants vous permettent d'élaborer des serveurs Web faisant partie d'une application de base de données multiniveau. Voir "Construction des applications Web avec InternetExpress" à la page 25-38, pour plus de détails.

Ajout d'une session au module Web

Les applications console CGI et les applications Win-CGI sont lancées en réponse à des messages de requête HTTP. Lorsque vous manipulez des bases de données dans des applications de ce type, vous pouvez utiliser la session par défaut pour gérer vos connexions aux bases de données car chaque message de requête possède sa propre instance de l'application. Chaque instance de l'application possède sa propre session par défaut.

Cependant, lorsque vous créez une application NSAPI ou ISAPI, chaque message de requête est géré dans un thread distinct d'une instance d'application. Pour empêcher que les connexions aux bases de données d'autres threads ne se parasitent, vous devez donner une session distincte à chaque thread.

Chaque message de requête dans une application ISAPI ou NSAPI génère un nouveau thread. Le module Web de ce thread est dynamiquement généré lors de l'exécution. Ajoutez un objet *TSession* au module Web pour gérer les connexions aux bases de données pour le thread qui contient le module Web.

A l'exécution, une instance distincte du module Web est générée pour chacun des threads. Chacun de ces modules contient l'objet session. Chaque session doit avoir un nom distinct pour que les threads qui gèrent les messages de requête ne parasitent pas leur connexion aux bases de données respectives. Pour que les objets session de chaque module s'attribuent dynamiquement un nom unique, définissez la propriété *AutoSessionName* de l'objet session. Chaque session s'attribuera dynamiquement un nom unique et définira la propriété *SessionName* de tous les ensembles de données du module de façon à ce qu'elle fasse référence à ce nom unique. Ceci permet l'interaction des threads de requête avec la base de données, sans interférences avec les autres messages de requête. Pour plus d'informations sur les sessions, voir "Gestion des sessions de bases de données" à la page 20-18.

Représentation HTML d'une base de données

Les commandes HTML offertes par les composants générateur de contenu spécialisé de la page Internet de la palette des composants dépendent des enregistrements contenus dans l'ensemble de données. Il existe deux types de générateurs de contenu orientés données :

- Le générateur de page ensemble de données, qui formate les champs d'un ensemble de données en texte d'un document HTML.
- Les générateurs de tableau, qui formatent les enregistrements d'un ensemble de données sous forme de tableau HTML.

Utilisation des générateurs de page ensemble de données

Les générateurs de page ensemble de données fonctionnent comme les autres composants générateur de page : ils convertissent un modèle incluant des balises HTML transparentes en une représentation HTML finale. Cependant, ils présentent la particularité de pouvoir convertir les balises dont le nom

correspond à celui d'un champ d'un ensemble de données en la valeur courante de ce champ. Pour plus d'informations sur l'utilisation générale des générateurs de page, voir "Utilisation du composant générateur de page" à la page 28-14.

Pour utiliser un générateur de page ensemble de données, ajoutez un composant *TDataSetPageProducer* au module Web et attribuez à sa propriété *DataSet* l'ensemble de données dont les valeurs de champ doivent être affichées dans le contenu HTML. Créez un modèle HTML qui décrit le résultat de votre générateur de page ensemble de données. Pour chaque valeur de champ à afficher, incluez une balise de la forme

```
<#NomChamp>
```

dans le modèle HTML, où *NomChamp* spécifie le nom du champ de l'ensemble de données dont la valeur doit être affichée.

Lorsque votre application appelle la méthode *Content*, *ContentFromString* ou *ContentFromStream*, le générateur de page ensemble de données remplace les balises représentant les champs par les valeurs courantes de ces derniers.

Utilisation des générateurs de tableau

La page Internet de la palette des composants offre deux composants qui permettent de créer un tableau HTML représentant les enregistrements d'un ensemble de données :

- Legénérateur de tableau ensemble de données, qui formate les champs d'un ensemble de données en texte d'un document HTML.
- Legénérateur de tableau requête, qui exécute une requête après avoir initialisé les paramètres fournis par le message de requête et formate l'ensemble de données obtenu en un tableau HTML.

A partir de l'un ou l'autre des générateurs de tableau, vous pouvez personnaliser l'aspect d'un tableau HTML obtenu en indiquant ses propriétés de couleur, de bordure, de type de séparateur, etc. Pour définir les propriétés d'un générateur de tableau à la conception, double-cliquez sur le composant générateur de tableau pour afficher la boîte de dialogue Editeur de réponses.

Choix des attributs de tableau

Les générateurs de tableau utilisent l'objet *THTMLTableAttributes* pour décrire l'aspect visuel du tableau HTML qui affiche les enregistrements de l'ensemble de données. L'objet *THTMLTableAttributes* inclut les propriétés de largeur et d'espacement du tableau dans le document HTML, ainsi que sa couleur de fond, l'épaisseur de la bordure, l'alignement du texte dans les cellules et l'espacement des cellules. Ces propriétés sont toutes converties en options de la balise HTML <TABLE> créée par le générateur de tableau.

Spécifiez ces propriétés lors de la conception dans l'inspecteur d'objets. Sélectionnez l'objet générateur de tableau et développez la propriété *TableAttributes* pour afficher les propriétés de l'objet *THTMLTableAttributes*.

Vous pouvez également spécifier ces propriétés par programmation à l'exécution.

Choix des attributs de lignes

En plus des attributs de tableau, vous pouvez spécifier l'alignement et la couleur du fond des lignes affichant des données. La propriété *RowAttributes* est un objet *THTMLTableRowAttributes*.

Spécifiez ces propriétés lors de la conception dans l'inspecteur d'objets, en développant la propriété *RowAttributes*. Vous pouvez également spécifier ces propriétés par programmation à l'exécution.

Il est également possible de modifier le nombre de lignes du tableau HTML en utilisant la propriété *MaxRows*.

Choix des attributs de colonnes

Si vous connaissez, au moment de la conception, l'ensemble de données à placer dans le tableau, vous pouvez utiliser l'éditeur de colonnes pour personnaliser le contenu et les attributs d'affichage des colonnes. Sélectionnez le composant générateur de tableau et faites un clic sur le bouton droit de la souris. Dans le menu contextuel, choisissez Editeur de colonnes. Ceci vous permet d'ajouter, de supprimer et de déplacer les colonnes du tableau. Vous pouvez définir les champs à placer et les propriétés d'affichage des colonnes dans l'inspecteur d'objets après les avoir sélectionnées dans l'éditeur de colonnes.

Si le nom de l'ensemble de données figure dans le message de requête HTTP, vous ne pouvez pas définir les champs à placer depuis l'éditeur de colonnes en phase de conception. Vous pouvez cependant personnaliser les colonnes par programmation lors de l'exécution en définissant les objets *THTMLTableColumn* correspondants et en utilisant les méthodes de la propriété *Columns* pour ajouter ces colonnes au tableau. Si vous ne vous servez pas de la propriété *Columns*, le générateur de tableau créera des colonnes par défaut adaptées aux champs de l'ensemble de données, mais ne spécifiera aucune caractéristique d'affichage.

Incorporation de tableaux dans un document HTML

Vous pouvez incorporer le tableau HTML représentant votre ensemble de données dans un document HTML à l'aide des propriétés *Header* et *Footer* du générateur de tableau. La propriété *Header* permet d'indiquer quelles informations doivent être affichées avant le tableau, alors que *Footer* spécifie ce qui vient après.

Il est possible d'utiliser un autre générateur de contenu (tel qu'un générateur de tableau) pour créer les valeurs des propriétés *Header* et *Footer*.

Lorsque vous incorporez votre tableau dans un document distinct, il est possible d'ajouter une légende au tableau à l'aide des propriétés *Caption* et *CaptionAlignment*.

Configuration d'un générateur de tableau ensemble de données

TDataSetTableProducer est un générateur de tableau qui permet de créer un tableau HTML pour un ensemble de données. Définissez la propriété *DataSet* de *TDataSetTableProducer* pour désigner l'ensemble de données contenant les enregistrements à afficher. Contrairement à la procédure normale pour la plupart

des objets orientés données d'une application de bases de données conventionnelle, la propriété *DataSource* n'a pas à être définie. Ceci s'explique par le fait que *TDataSetTableProducer* génère sa propre source de données en interne.

Vous pouvez choisir la valeur de *DataSet* en phase de conception si votre application Web affiche toujours des enregistrements issus du même ensemble de données. Vous devez définir la propriété *DataSet* lors de l'exécution si l'ensemble de données varie en fonction des informations contenues dans le message de requête HTTP.

Configuration d'un générateur de tableau requête

Vous pouvez générer un tableau HTML pour afficher les résultats d'une requête, dans lequel les paramètres de recherche viennent du message HTTP. Désignez l'objet *TQuery* qui utilise ces paramètres comme propriété *Query* d'un composant *TQueryTableProducer*.

Si le message est une requête GET, les paramètres de recherche sont situés dans les champs *Query* de l'URL fournie comme destination du message de requête HTTP. Si le message est une requête POST, ces paramètres sont dans le contenu du message de requête.

Lorsque vous appelez la méthode *Content* de *TQueryTableProducer*, elle exécute la requête à l'aide des paramètres figurant dans l'objet requête. Elle formate ensuite un tableau HTML pour représenter les enregistrements dans l'ensemble de données obtenu.

Comme avec tout générateur de tableau, vous pouvez personnaliser les propriétés d'affichage ou de contenu des colonnes du tableau HTML. Il est également possible d'incorporer le tableau dans un autre document HTML.

Chapitre

29

Utilisation de WebSnap

WebSnap étend l'agent Web avec de nouveaux composants, des experts et des vues qui simplifient la conception d'applications Web contenant des pages Web complexes orientées données. WebSnap gère plusieurs modules ainsi que les scripts côté serveur, ce qui simplifie le développement en équipe.

Les composants répartiteur gèrent automatiquement les requêtes pour le contenu des pages, la validation des formulaires HTML et les requêtes d'images dynamiques. De nouveaux composants, les adaptateurs, permettent de définir une interface sous forme de scripts avec les règles de gestion de votre application. Ainsi, l'objet *TDataSetAdapter* permet de créer des composants ensembles de données manipulables par scripts. Vous pouvez utiliser de nouveaux composants générateur pour concevoir rapidement des formulaires ou des tableaux complexes orientés données ou utiliser XSL pour générer une page. Le composant session vous permet de conserver une trace des utilisateurs finaux. Le composant liste d'utilisateurs permet de gérer des utilisateurs, des mots de passes et des droits d'accès.

L'expert application Web permet de générer rapidement une application personnalisée avec les composants dont vous avez besoin. L'expert module de page Web vous permet de créer un module définissant une nouvelle page de votre application. L'expert module de données Web vous permet de créer un conteneur pour des composants réutilisés dans votre application Web.

Les vues de module de page peuvent visualiser le résultat d'un script côté serveur sans avoir à exécuter l'application. L'onglet Prévisualisation affiche la page dans un navigateur incorporé. La vue Résultat HTML affiche le HTML généré. Les vues Arborescence XSL et Arborescence XML simplifient la manipulation du XML et du XSL.

Pour gérer une équipe de développeurs, vous pouvez utiliser la gestion multimodule de WebSnap afin de décomposer une application en unités développables séparément. Lors de la création d'un nouveau module de page, l'expert module de page peut créer un fichier modèle externe. Ce fichier modèle

externe peut être modifié hors de l'EDI et testé sans avoir à recompiler l'application.

Les sections suivantes de ce chapitre décrivent comme utiliser les composants WebSnap pour créer une application serveur Web.

Création d'applications serveur Web avec WebSnap

Pour créer une nouvelle application serveur Web utilisant l'architecture WebSnap :

1 Sélectionnez Fichier | Nouveau | Autre.

2 Dans la boîte de dialogue Nouveaux éléments, sélectionnez l'onglet WebSnap et choisissez Application WebSnap.

3 La boîte de dialogue qui s'affiche attend les informations suivantes :

- Le type de serveur
- Le type de module d'application Web
- Les options de module d'application Web
- Les composants de l'application

Type de serveur

Sélectionnez l'un des types suivants d'application serveur Web :

- ISAPI et NSAPI : la sélection de ce type d'application configure votre projet comme une DLL proposant les méthodes exportées attendues par le serveur Web. L'en-tête de la bibliothèque est également inclus dans le fichier projet, ainsi que les entrées nécessaires dans les listes des clauses uses et exports du fichier projet.
- Apache : la sélection de ce type d'application configure votre projet comme une DLL proposant les méthodes exportées attendues par le serveur Web Apache. L'en-tête de la bibliothèque est également inclus dans le fichier projet, ainsi que les entrées nécessaires dans les listes des clauses uses et exports du fichier projet.
- Exécutable autonome CGI : la sélection de ce type d'application configure votre projet comme une application en mode console et ajoute les entrées nécessaires à la clause uses du fichier projet.
- Exécutable autonome Win-CGI : si vous sélectionnez ce type d'application, le projet est configuré comme une application Windows et ajoute les entrées nécessaires à la clause uses du fichier projet.
- Exécutable débogueur d'application Web : la sélection de ce type d'application configure un environnement permettant de développer et de tester des applications serveur Web. Ce type d'application n'est pas conçu pour être déployé.

Choisissez le type d'application serveur Web qui communique avec le type de serveur Web que votre application utilisera.

Types de module d'application Web

Le module d'application Web permet un contrôle centralisé des règles de gestion et des composants non-visuels d'une application Web. Il y a deux types de modules d'application Web :

- Module de page. La sélection de ce type de module crée une page de contenu. Le module de page contient un générateur de page chargé de générer le contenu d'une page. Le générateur de page affiche sa page associée quand les informations de chemin d'accès (pathinfo) de la requête HTTP correspondent au nom de la page. Ce peut être la page par défaut si la zone pathinfo est vide.
- Module de données. La sélection de ce type de module de données ne crée pas de contenu de page. Ce module est utilisé comme conteneur pour des composants partagés par d'autres modules : par exemple, les composants de bases de données utilisés par plusieurs modules de page Web.

Options des modules d'application Web

Si vous sélectionnez un module d'application de type module de page, vous pouvez associer un nom à la page en renseignant la zone Nom de page de la boîte de dialogue. A l'exécution, l'instance de ce module peut être placée dans un cache ou retirée de la mémoire quand la requête a été traitée. Vous pouvez choisir le comportement adopté en utilisant les options de la zone Mise en mémoire cache. Vous pouvez également définir d'autres options du module de page via le bouton Options de page. Vous pouvez les choisir parmi les catégories suivantes :

- Générateur. Le type de générateur de la page peut être l'un des suivants : *AdapterPageProducer*, *DataSetPageProducer*, *InetXPageProducer*, *PageProducer* ou *XSLPageProducer*. Si le générateur de page sélectionné gère les scripts, utilisez la liste déroulante Moteur de script pour sélectionner le type de langage utilisé pour les scripts de la page.

Remarque Le type *AdapterPageProducer* ne gère que JScript.

- HTML. Quand le type de générateur sélectionné utilise un modèle HTML, ce groupe est visible.
- XSL. Quand le type de générateur sélectionné utilise un modèle XSL (comme *TXSLPageProducer*), ce groupe est visible.
- Nouveau fichier. Cochez cette option si vous voulez qu'un fichier modèle soit créé et géré comme faisant partie de l'unité. Le fichier modèle géré apparaît alors dans le gestionnaire de projet et utilise le même nom et le même emplacement que le fichier source de l'unité. Ne cochez pas cette option si

vous voulez utiliser les propriétés du composant générateur (en général, les propriétés *HTMLDoc* et *HTMLFile*).

- Modèle. Si l'option Nouveau fichier est cochée, choisissez le contenu par défaut du fichier modèle en utilisant la liste déroulante Modèle. Le modèle "Défaut" affiche le titre de l'application, le titre de la page et un lien hypertexte vers les pages publiées.
- Page. Saisissez le nom de la page et le titre du module de page. Le nom de page est utilisé pour référencer la page dans une requête HTTP ou dans la logique même de l'application. Le titre est le nom que l'utilisateur final voit quand la page est affichée dans un navigateur.
- Publié. Cochez cette option pour répondre automatiquement aux requêtes HTTP dont le paramètre pathinfo correspond au nom de la page.
- Nom de connexion requis. Cochez cette option pour obliger l'utilisateur à s'identifier avant de pouvoir accéder à la page.

Composants d'application

Les composants d'application fournissent les fonctionnalités de l'application Web. Si, par exemple, vous utilisez un composant répartiteur d'adaptateur, celui-ci gère automatiquement la validation des formulaires HTML et renvoie automatiquement les images gérées dynamiquement. Un répartiteur de page affiche automatiquement le contenu d'une page quand la zone pathinfo d'une requête HTTP contient le nom de la page.

La sélection du bouton Composants affiche une boîte de dialogue dans laquelle vous pouvez sélectionner un ou plusieurs composants d'application :

- Adaptateur d'application. Contient des informations sur l'application comme le titre. Le type par défaut est *TApplicationAdapter*.
- Adaptateur utilisateur final. Contient des informations sur l'utilisateur, comme le nom, les droits d'accès et s'il est connecté. Le type par défaut est *TEndUserAdapter*. Vous pouvez également sélectionner *TEndUserSessionAdapter*.
- Répartiteur de page. Examine la valeur de la zone pathinfo de la requête HTTP et appelle le module de page approprié pour renvoyer le contenu d'une page. Le type par défaut est *TPageDispatcher*.
- Répartiteur d'adaptateur. Gère automatiquement la validation des formulaires HTML et les requêtes d'image dynamique en appelant des composants action adaptateur ou champs. Le type par défaut est *TAdapterDispatcher*.
- Actions de répartition. Vous permet de définir une collection d'éléments action pour gérer les requêtes basées sur pathinfo et le type de méthode. Les éléments action appellent des événements définis par l'utilisateur ou demandent le contenu à des composants générateur de page. Le type par défaut est *TWebDispatcher*.

- Service de localisation de fichiers. Permet de contrôler le chargement de fichiers modèle et de fichiers include de scripts pendant l'exécution de l'application Web. Le type par défaut est *TLocateFileService*.
- Service de sessions. S'utilise pour stocker des informations sur un utilisateur final qui sont nécessaires pour un court laps de temps. Les sessions vous permettent par exemple de garder la trace des utilisateurs connectés et de déconnecter automatiquement un utilisateur après une certaine période d'inactivité. Le type par défaut est *TSessionService*.
- Service de liste d'utilisateurs. Stocke une liste des utilisateurs autorisés, de leurs mot de passe et droits d'accès. Le type par défaut est *TWebUserList*.

Pour chacun de ces composants, les types de composant listés sont les types par défaut livrés avec Delphi. Les utilisateurs peuvent créer leur propres types de composant ou utiliser des composants proposés par des tiers.

Modules Web

Il y a quatre types de modules Web :

- TWebAppPageModule
- *TWebAppDataModule*
- *TWebPageModule*
- *TWebDataModule*

Le module application Web (*TWebAppPageModule* ou *TWebAppDataModule*) sert de conteneur aux composants de l'application qui remplissent des fonctions portant sur l'application prise comme un tout : la répartition des requêtes, le gestion des sessions ou la gestion de la liste des utilisateurs. Une application ne peut contenir qu'un seul module d'application de ce type.

Le module de page Web (*TWebPageModule)* fournit le contenu d'une page alors que le module de données Web (*TWebDataModule*) sert de conteneur à des composants utilisés dans divers endroits de votre application. Vous pouvez éventuellement inclure plusieurs modules de pages Web ou modules de données Web dans le module d'application Web.

Modules de données Web

Comme les modules de données standard, un module de données Web sert de conteneur à des composants de la palette. Les modules de données fournissent une surface de conception dans laquelle ajouter, retirer ou sélectionner des composants. Lors de l'exécution de l'application, un module de données ne crée pas de fenêtre.

Le module de données Web diffère d'un module de données standard par la structure de l'unité et les interfaces qu'il implémente.

Utilisez un module de données Web comme conteneur pour les composants partagés par votre application. Vous pouvez, par exemple, placer un composant ensemble de données dans un module de données et y accéder depuis :

- un module de page affichant une grille, et
- un module de page affichant un formulaire de saisie.

Structure de l'unité d'un module de données Web

Les modules de données standard ont une variable appelée la variable de fiche qui est utilisée pour accéder à l'objet module de données. Les modules de données Web proposent à la place une fonction. Elle a le même rôle. Cependant, comme les applications WebSnap peuvent être multithreads et avoir plusieurs instances d'un module traitant simultanément plusieurs requêtes, cette fonction est implémentée de manière à renvoyer la bonne instance.

Cette unité recense également un fabricant. Le fabricant spécifie comment le module doit être géré par l'application WebSnap. Par exemple, des indicateurs spécifient s'il faut placer le module dans un cache afin de le réutiliser ou s'il faut détruire le module après le traitement d'une requête.

Interfaces implémentées par un module de données Web

Un module de données Web implémente les interfaces suivantes :

INotifyWebActivate. Cette interface est appelée quand le module est activé pour traiter une requête Web et avant la désactivation du module après traitement de la requête Web.

IWebVariablesContainer. Cette interface est appelée durant l'exécution d'un script afin de résoudre les références de variables. Le répartiteur d'adaptateur appelle également cette interface pour rechercher les actions d'adaptation et les champs référencés dans une requête HTTP.

IGetScriptObject. Cette interface est appelée pour obtenir l'implémentation par le module de IDispatch. L'objet renvoyé est l'interface entre le module et le moteur de script actif.

IIteratorObjectSupport. Cette interface est utilisée pour parcourir tous les objets du module accessibles par script.

Modules de page Web

Le module de page est associé à un composant générateur de page. A la réception d'une requête, le répartiteur de page analyse la requête et appelle le module de page approprié afin de traiter la requête et renvoyer le contenu de la page.

Comme les modules de données Web, les modules de page Web servent de conteneur à des composants. La différence étant que le module de page Web sert lui à générer une page Web.

Composant générateur de page

Les modules de page Web ont une propriété qui identifie le composant générateur de page responsable de la génération du contenu de la page. Les experts WebSnap ajoutent automatiquement un générateur lors de la création d'un module de page Web. Vous pouvez ultérieurement changer de composant générateur de page en déposant un autre générateur provenant de la palette WebSnap. Cependant, si le module de page utilise un fichier modèle, assurez-vous que le contenu du fichier est compatible avec le composant générateur.

Nom de page

Les modules de page Web ont un nom de page qui sert à référencer la page dans une requête HTTP ou dans la logique de l'application. Un fabricant dans l'unité du module de page Web spécifie le nom de page pour le module de page Web.

Modèle de générateur

La plupart des générateurs de page utilisent un modèle. Généralement, les modèles HTML contiennent du code HTML statique mélangé à des balises transparentes ou du script exécuté côté serveur. Quand les générateurs de page créent leur contenu, ils remplacent les balises transparentes par les valeurs appropriées et exécutent les scripts serveur afin de générer le HTML qui est affiché dans le navigateur du client. XSLPageProducer constitue une exception. Il utilise des modèles XSL, qui contiennent du XSL et non du HTML. Les modèles XSL ne gèrent pas les balises transparentes ou les scripts serveur.

Les modules de page Web peuvent avoir un fichier modèle associé qui est géré comme faisant partie de l'unité. Un fichier modèle géré apparaît dans le gestionnaire de projet et utilise le même nom et le même emplacement que le fichier source de l'unité. Si le module de page Web n'a pas de fichier modèle associé, ce sont les propriétés du composant générateur qui spécifient le modèle.

Interfaces implémentées par le module de page Web

Un module de page Web implémente toutes les interfaces d'un module de données Web plus les interfaces suivantes :

IDefaultPageFileName. Le concepteur de surface WebSnap utilise cette interface pour vérifier qu'un module de page utilise le bon fichier modèle.

ISetWebContentOptions. Le concepteur de surface WebSnap utilise cette interface pour contrôler le contenu généré par le module de page. Ainsi, une option permet de récupérer le contenu avant l'exécution de scripts ASP.

IGetProducerComponent. Le concepteur de surface WebSnap utilise cette interface pour obtenir le composant générateur associé au module de page.

IProducerEditorViewSupport. Le concepteur de surface WebSnap utilise cette interface pour obtenir des informations sur les vues de l'éditeur qu'il doit afficher quand le module de page est actif. Les vues de l'éditeur disponibles sont : Script HTML, Prévisualiser, Résultat HTML, Arborescence XSL et Arborescence XML.

IPageResult. Cette interface donne accès au résultat généré par un module de page. Cette interface gère trois types de résultat : Contenu HTTP, Redirection HTTP et Page Include.

IGetDefaultAction. Cette interface récupère l'éventuelle action d'adaptation associée au module de page.

Modules d'application Web

Le module d'application Web implémente des interfaces qui ne sont pas implémentées par *TWebPageModule* ou *TWebDataModule*.

Interfaces implémentées par un module de données d'application Web

Un module de données d'application Web implémente toutes les interfaces d'un module de données Web en plus des suivantes :

IGetWebAppServices. Récupère l'interface avec le gestionnaire de requêtes de l'application Web.

IGetWebAppComponents. Récupère l'interface avec les composants de niveau application (AdapterDispatcher, PageDispatcher, SessionsService et EndUserAdapter).

Interfaces implémentées par un module de page d'application Web

Un module de page d'application Web implémente toutes les interfaces d'un module de données Web et d'un module de page Web.

Adaptateurs

Les adaptateurs permettent de créer une interface avec les données de l'application. Ils vous permettent d'insérer du code en langage de script dans une page et de récupérer des informations en appelant les adaptateurs depuis le code script.

Vous pouvez, par exemple, utiliser un adaptateur pour définir les champs de données à afficher dans une page HTML. Une page HTML scriptée peut alors contenir du contenu HTML et des instructions en script qui récupère la valeur de ces champs.

Champs

Les champs sont des composants qu'utilise le générateur de page pour récupérer des données de votre application afin d'en afficher le contenu dans une page Web. Les champs peuvent également servir à récupérer une image. Dans ce cas, le champ renvoie l'adresse de l'image écrite dans la page Web. Quand une page affiche son contenu, une requête envoyée à l'application Web demande au répartiteur d'adaptateur de rechercher l'image réelle depuis le composant champ.

Actions

Les actions sont des composants qui exécutent des commandes pour l'adaptateur. Quand un générateur de page génère sa page, le langage de script appelle les composants action d'adaptation pour renvoyer le nom de l'action ainsi que tous les paramètres nécessaires à l'exécution de la commande. Par exemple, soit un bouton d'un formulaire HTML devant supprimer une ligne d'une table. Il renvoie dans la requête HTTP le nom de l'action associé au bouton et un paramètre indiquant le numéro de ligne. Le répartiteur d'adaptateur recherche le composant action ainsi nommé et lui transmet le numéro de ligne comme paramètre de l'action.

Erreurs

Les adaptateurs gèrent une liste des erreurs se produisant lors de l'exécution d'une action. Les générateurs de page peuvent accéder à cette liste afin de les afficher dans la page Web renvoyée par l'application à l'utilisateur final.

Enregistrements

Certains composants adaptateur, comme *TDataSetAdapter*, représentent plusieurs lignes. L'adaptateur propose alors une interface de scriptage qui permet de parcourir les lignes. Certains adaptateurs gèrent la pagination et ne permettent de parcourir que les lignes de la page en cours.

Générateurs de page

Vous pouvez utiliser les générateurs de page afin de produire le contenu pour un module de page Web. Vous pouvez aussi utiliser des générateurs comme dans des applications WebBroker, en associant le générateur à un élément d'action d'un répartiteur Web. L'utilisation des modules de page Web offre les avantages suivants :

- La possibilité de prévisualiser la disposition des pages sans avoir à exécuter l'application.
- L'association entre un nom de page et un module permet au répartiteur de page d'appeler automatiquement le générateur de page.

Modèles

Les générateurs proposent les fonctionnalités suivantes :

- Ils génèrent du contenu HTML.
- Ils peuvent référencer un fichier externe en utilisant la propriété HTMLfile ou une chaîne interne avec la propriété HTMLDoc.

- S'ils sont utilisés en association avec un module de page Web, le modèle peut-être un fichier associé à une unité.
- Ils génèrent dynamiquement du HTML qui peut être inséré dans le modèle en utilisant des balises transparentes ou du scriptage actif. Les balises transparentes s'utilisent de la même manière que dans une application agent Web. Pour plus d'informations, voir "Conversion des balises HTML transparentes" à la page 28-16. La gestion du scriptage actif vous permet d'incorporer du code JavaScript ou VBscript dans la page HTML.

Utilisation de scripts serveur avec WebSnap

Les modèles de générateur de page peuvent utiliser du code écrit avec des langages de script comme JScript ou VBScript. Le générateur de page exécute le script en réponse à une demande du contenu du générateur. Comme c'est l'application Web qui évalue le script, on parle de script serveur (ou de script côté serveur) par opposition au script côté client qui est exécuté par le navigateur.

Scriptage actif

WebSnap utilise le *scriptage actif* pour implémenter les scripts serveur. Le scriptage actif est une technologie crée par Microsoft permettant l'utilisation d'un langage de script par des objets d'application via des interfaces COM. Microsoft fournit deux langages de script, VBScript et JScript. La gestion d'autres langages est proposée par d'autres fournisseurs.

Moteur de script

La propriété *ScriptEngine* du générateur de page identifie le moteur de script qui doit évaluer les scripts placés dans un modèle.

Blocs de script

Les blocs de script sont délimités par <% et %>. Le moteur de script évalue tout texte placé à l'intérieur d'un bloc de script. Le résultat est intégré au contenu du générateur de page. Le générateur de page écrit le texte placé hors d'un bloc de script après avoir traduit les balises transparentes incorporées. Les blocs de script peuvent également contenir du texte, ce qui permet de définir le texte en sortie à l'aide de conditions ou de boucles. Par exemple, le bloc JScript suivant génère une liste de cinq lignes numérotées :

```
<ul>
<% for (i=0;i<5;i++) { %>
     <li>Item <% Response.Write(i) %></li>
<% } %>
</ul>
```

Le bloc de script suivant donne un résultat équivalent :

```
<ul>
<% for (i=0;i<5;i++) { %>
     <li>Item <%=i %></li>
<% } %>
</ul>
```

Le délimiteur <%= est un raccourci de *Response.Write*.

Création de scripts

Les développeurs peuvent profiter des caractéristiques de WebSnap pour générer des scripts automatiquement.

Experts modèles

Lors de la création d'une nouvelle application WebSnap ou d'un module de page, les experts WebSnap proposent une zone Modèle qui permet de sélectionner le contenu initial d'un modèle de module de page. Par exemple, le modèle nommé "Défaut" génère du code JScript affichant le titre de l'application, le nom de la page et des liens vers les pages publiées.

TAdapterPageProducer

TAdapterPageProducer produit des formulaires et des tableaux en générant du HTML et du code JScript. Le code JScript généré appelle les objets adaptateurs pour récupérer la valeur des champs, les paramètres des champs image et les paramètres d'action.

Modification et visualisation des scripts

Le concepteur de surface WebSnap propose une visualisation des modules de page Web qui permet de prévisualiser une page scriptée. Utilisez la vue Résultat HTML pour afficher le HTML produit par l'exécution du script. Utilisez l'onglet Prévisualiser pour afficher le résultat dans un navigateur. Si le module de page Web utilise *TAdapterPageProducer*, l'onglet Script HTML est également proposé. Cet onglet affiche le HTML et le code JScript généré par l'objet *TAdapterPageProducer*. Consultez cette vue pour voir comment écrire des scripts construisant des formulaires HTML afin d'afficher des champs adaptateur et d'exécuter des actions d'adaptation.

Comment inclure un script dans une page

Un modèle peut inclure un script depuis un fichier ou depuis une autre page. Pour inclure un script se trouvant dans un fichier, utilisez l'instruction suivante :

```
<!-- #include file="nomfichier.html" "-->
```

Quand le modèle inclut un script contenu dans une autre page, le script est évalué par la page qui inclut le script. Utilisez le code suivant pour inclure une version non évaluée de page1.

```
<!-- #include page="page1" -- >
```

Objets scriptables

Les objets scriptables sont des objets VCL ou CLX qui peuvent être référencés par un script. Pour que des objets VCL ou CLX soient accessibles par script, vous devez recenser une interface *IDispatch* avec l'objet dans le moteur de script actif. Les objets suivants sont manipulables par script :

- **Application**. L'objet application (qui peut être null) donne accès à l'adaptateur d'application du module d'application Web. Le bloc de code JScript suivant écrit le titre de l'application :

  ```
  <%= Application.Title %>
  ```

- **EndUser**. L'objet EndUser donne accès à l'adaptateur utilisateur final du module d'application Web. Le bloc de code JScript suivant écrit le nom de l'utilisateur final :

  ```
  <%= EndUser.DisplayName %>
  ```

- **Session**. L'objet session donne accès à l'objet session du module d'application Web. Le bloc de code JScript suivant écrit l'ID de la session :

  ```
  <%= Session.SessionID %>
  ```

- **Pages**. Les objets page (*Pages*) donnent accès aux pages de l'application. Le bloc de code JScript suivant écrit des liens vers toutes les pages publiées :

  ```
  <%  e = new Enumerator(Pages)
       for (; !e.atEnd(); e.moveNext())
       {
         if (e.item().Published)
         {
             Response.Write('<A HREF="' + e.item().HREF +'">' + e.item().Title + '</A>')
         }
       }
  %>
  ```

Remarquez que la page prévisualisation de l'éditeur n'affiche pas un résultat correct pour ce bloc de script. En effet, les objets pages sont toujours vides à la conception car les fabricants de module de page Web n'ont pas été recensés.

- **Modules**. Les objets module donnent accès aux modules de l'application. Le bloc de code JScript suivant écrit le contenu d'un champ adaptateur dans un module nommé DM :

  ```
  <%= Modules.DM.Adapter1.Field1.DisplayText %>
  ```

- **Page**. L'objet Page donne accès à la page en cours. Le bloc de code JScript suivant écrit le titre de la page en cours :

  ```
  <%= Page.Title %>
  ```

- **Producer**. L'objet Producer donne accès au générateur de page du module de page Web. Le bloc de code JScript suivant évalue une balise transparente avant d'écrire le contenu :

```
<% Producer.Write('Voici une balise <#TAG>') %>
```

Remarquez que la page prévisualisation de l'éditeur n'affiche pas un résultat correct pour ce bloc de script. En effet, les gestionnaires d'événements qui généralement remplacent les balises transparentes ne sont exécutés que lorsque l'application s'exécute.

- **Response**. L'objet Response donne accès à WebResponse. Utilisez cet objet quand vous voulez éviter d'employer la substitution de balises.

```
<% Response.Write('Hello World!') %>
```

- **Request**. L'objet Request donne accès à WebRequest. Le bloc de code JScript suivant écrit le pathinfo.

```
<%= Request.PathInfo %>
```

- **Objets adaptateur**. Il est possible de référencer sans qualification tous les composants de l'adaptateur de la page en cours. Pour les adaptateurs d'autres modules, utilisez une qualification avec les objets modules. Le bloc de code JScript suivant affiche la valeur du champ *FirstName* de toutes les lignes de Adapter1 :

```
<% e = new Enumerator(Adapter1.Records) %>
  <% for (; !e.atEnd(); e.moveNext()) %>
  <% { %>
      <p><%= Adapter1.FirstName.DisplayText %>
  <% } %>
```

Répartition des requêtes

Quand une application WebSnap reçoit un message de requête HTTP, elle crée un objet *TWebRequest* pour représenter le message de requête HTTP et un objet *TWebResponse* pour représenter la réponse qui doit être renvoyée.

WebContext

Avant de traiter la requête, le module d'application Web initialise un objet WebContext (de type *TWebContext*). WebContext est une variable thread qui donne un accès global aux variables utilisées par les composants traitant la requête. Par exemple, WebContext contient les objets *TWebResponse* et *TWebRequest* ainsi que les objets adaptateur de requête et de réponse décrits plus bas dans cette section.

Composants répartiteur

Les composants répartiteur du module d'application Web contrôlent le flux de l'application. Les répartiteurs déterminent comment traiter certains types de messages de requête HTTP en analysant la requête HTTP.

Le composant répartiteur d'adaptateur (*TAdapterDispatcher)* recherche un champ de contenu ou un champ de requête qui identifie un composant action d'adaptation ou un composant d'adaptation de champ image. Si le répartiteur d'adaptateur trouve un composant, il transmet le contrôle au composant.

Le composant répartiteur Web (*TWebDispatcher*) gère une collection des éléments action (*TWebActionItem*) qui savent gérer certains types de messages de requête HTTP. Le répartiteur Web recherche un élément action correspondant à la requête. S'il en trouve un, il transmet le contrôle à cet élément action. Le répartiteur Web recherche également les composants d'auto-répartition qui peuvent gérer la requête.

Le composant répartiteur de page (*TPageDispatcher*) examine la propriété PathInfo de l'objet *TWebRequest*, en y recherchant le nom d'un module de page Web recensé. Si le répartiteur trouve le nom d'un module de page Web, il transmet le contrôle à ce module de page Web.

Fonctions d'un répartiteur d'adaptateur

Le composant répartiteur d'adaptateur gère automatiquement la validation des formulaires HTML et les requêtes d'image dynamique en appelant des composants action d'adaptation et des composants champ.

Utilisation de composants d'adaptation pour générer du contenu

Pour que les applications WebSnap exécutent automatiquement les actions d'adaptation et récupèrent les images dynamiques depuis les champs d'adaptation, il faut que le contenu HTML soit construit correctement. Si le contenu HTML n'est pas construit correctement, la requête HTTP correspondante ne contient pas les informations dont le répartiteur d'adaptateur a besoin pour appeler les composants action d'adaptation et champ.

Afin de limiter les erreurs lors de la construction de la page HTML, les composants adaptateur indiquent le nom et la valeur que doivent prendre les éléments HTML. Les composants adaptateur ont des méthodes pour obtenir le nom et la valeur des champs cachés qui doivent apparaître dans un formulaire HTML utilisé pour actualiser des champs d'adaptation. Généralement les générateurs de page utilisent des scripts serveur pour obtenir des composants adaptateur les noms et les valeurs puis génèrent le HTML en les utilisant. Par exemple, le script suivant construit un élément <IMG> qui référence le champ Graphic de Adapter1 :

```
<img src="<%=Adapter1.Graphic.Image.AsHREF%>" alt="<%=Adapter1.Graphic.DisplayText%>">
```

Quand l'application Web évalue le script, l'attribut src HTML contiendra les informations nécessaires à l'identification du champ ainsi que les paramètres dont le composant champ peut avoir besoin pour récupérer l'image. Le HTML résultat aura la forme suivante :

```
<img src="?_lEspece No=90090&__id=DM.Adapter1.Graphic" alt="(GRAPHIC)">
```

Quand le navigateur envoie à l'application Web une requête HTTP pour obtenir cette image, le répartiteur d'adaptateur est capable de déterminer que le champ Graphic de Adapter1 dans le module DM, doit être appelé avec le paramètre "Espece No=90090". Le répartiteur d'adaptateur appelle le champ Graphic pour écrire la réponse HTTP appropriée.

Le script suivant construit un élément <A> faisant référence à l'action EditRow de Adapter1 dans la page appelée "Details" :

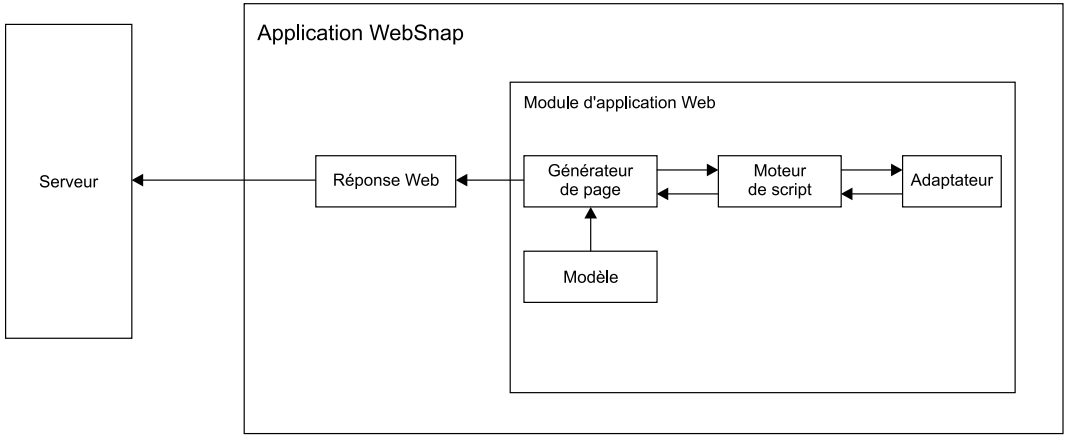
```
<a href="<%=Adapter1.EditRow.LinkToPage("Details", Page.Name).AsHREF%>">Edit...</a>
```

Le HTML résultant a la forme suivante :

```
<a href="?&_lEspece No=90310&__sp=Edit&__fp=Grid&__id=DM.Adapter1.EditRow">Edit...</a>
```

Quand l'utilisateur final clique sur ce lien hypertexte et que le navigateur envoie une requête HTTP, le répartiteur d'adaptateur peut déterminer que l'action EditRow de Adapter1 dans le module DM, doit être appelée avec le paramètre "Espece No=903010". Le répartiteur d'adaptateur indique également que la page Edit doit être affichée si l'action est correctement exécutée et que la page Grid doit être affichée si l'exécution de l'action échoue. Il appelle ensuite l'action EditRow pour rechercher la ligne à modifier et la page nommée Edit est appelée pour générer une réponse HTTP. La figure suivante illustre comment les composants adaptateur sont utilisés pour générer du contenu.

Figure 29.1 Flux de la génération du contenu



Requêtes et réponses des adaptateurs

Quand le répartiteur d'adaptateur reçoit la requête du client, il crée des objets requête et réponse d'adaptation pour stocker des informations sur la requête HTTP. Les objets requête et réponse d'adaptation sont stockés dans le WebContext afin de pouvoir y accéder lors du traitement de la requête.

Le répartiteur d'adaptateur crée deux types d'objet requête d'adaptation : action et image. Il crée l'objet requête action lors de l'exécution d'une action d'adaptation. Il crée l'objet requête d'image pour obtenir une image d'un champ adaptateur.

L'objet réponse d'adaptation est utilisé par le composant adaptateur pour indiquer la réponse à une action d'adaptation ou à une requête image d'adaptation. Il y a deux types d'objet réponse d'adaptation : action et image.

Requêtes action

L'objet requête action est responsable de la décomposition de la requête HTTP afin d'obtenir les informations nécessaires à l'exécution d'une action d'adaptation. Les types d'informations nécessaires à l'exécution d'une action d'adaptation peuvent être :

- **Le nom du composant**. Identifie le composant action d'adaptation.
- **Le mode d'adaptation**. Les adaptateurs peuvent définir un mode. Par exemple, *TDataSetAdapter* gère les modes modification, insertion et consultation. Une action d'adaptation peut s'exécuter différemment selon le mode. Par exemple, avec *TDataSetAdapter*, l'action Apply ajoute un nouvel enregistrement en mode insertion et actualise l'enregistrement en mode modification.
- **Page de réussite**. La page de réussite indique la page à afficher si l'action arrive à s'exécuter.
- **Page d'échec**. Indique la page à afficher en cas d'erreur lors de l'exécution de l'action.
- **Paramètres de requête action**. Identifie les paramètres nécessaires à l'action d'adaptation. Par exemple, l'action Apply de *TDataSetAdapter* a besoin de la valeur de clé identifiant l'enregistrement à actualiser.
- **Valeurs de champ d'adaptation**. Ce sont les valeurs des champs d'adaptation transmis dans la requête HTTP lors de la validation d'un formulaire HTML. Les valeurs de champ peuvent être les nouvelles valeurs saisies par l'utilisateur final, les valeurs initiales du champ d'adaptation ou des fichiers téléchargés.
- **Clés d'enregistrement**. Si un formulaire HTML apporte des modifications à plusieurs enregistrements, les clés utilisées par le composant action d'adaptation sont nécessaires pour identifier de manière unique chacun des enregistrements afin que l'action d'adaptation soit effectuée sur chaque enregistrement. Par exemple, quand l'action Apply de *TDataSetAdapter* est effectuée sur plusieurs enregistrements, les clés d'enregistrement sont utilisées pour rechercher chaque enregistrement de l'ensemble de données avant d'actualiser les champs de l'ensemble de données.

Réponse action

L'objet réponse action génère une réponse HTTP pour le composant action d'adaptation. L'action d'adaptation indique le type de réponse en initialisant les propriétés de l'objet ou en appelant des méthodes de l'objet action réponse. Les propriétés sont :

- *Options de redirection*. Les options de redirection indiquent s'il faut effectuer une redirection HTTP au lieu de renvoyer un contenu HTML.
- *Statut d'exécution*. L'initialisation du statut d'exécution à "réussi" fait que la réponse action par défaut devient le contenu de la page de réussite définie par la requête action.

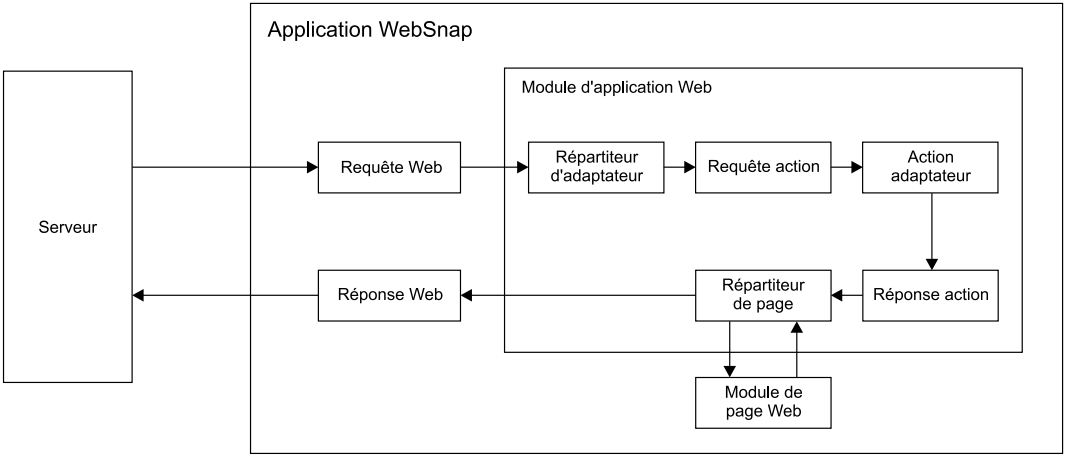
Parmi les méthodes de réponses action :

- *RespondWithPag*. L'action d'adaptation appelle cette méthode quand un module de page Web particulier doit générer la réponse.
- *RespondWithComponent*. L'action d'adaptation appelle cette méthode quand la réponse doit provenir du module de page Web contenant ce composant.
- *RespondWithURL*. L'action d'adaptation appelle cette méthode quand la réponse est une redirection vers l'URL spécifiée.

Quand la réponse est une page, l'objet réponse action tente d'utiliser le répartiteur de page pour générer le contenu de la page. S'il ne trouve pas le répartiteur de page, il appelle directement le module de page Web.

La figure suivante illustre comment les objets réponse action et requête action traitent une requête.

Figure 29.2 Requête et réponse action



Requête d'image

L'objet requête d'image est responsable de la décomposition de la requête HTTP afin d'obtenir les informations dont a besoin le champ d'adaptation d'image

pour générer une image. Les informations représentées par la requête d'image sont :

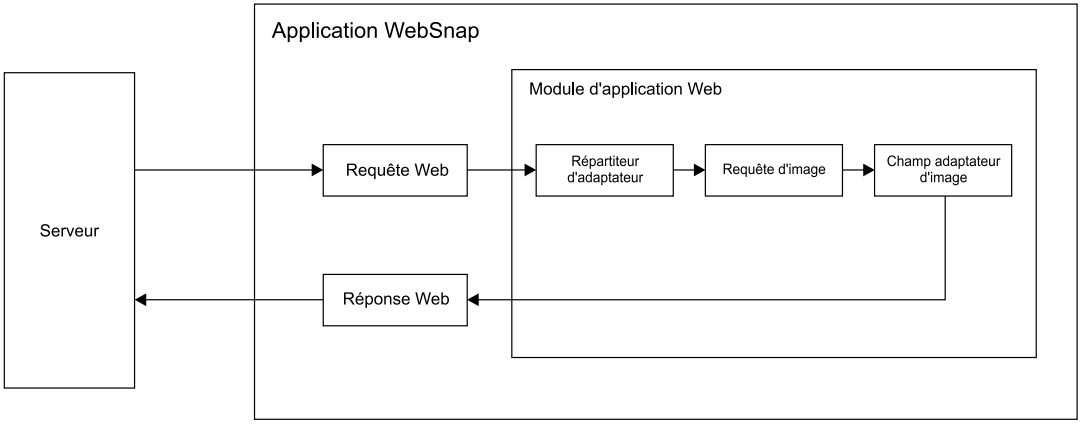
- Nom du composant. Identifie le composant champ d'adaptation.
- Paramètres de requête d'image. Identifie les paramètres nécessaires. Par exemple, l'objet *TDataSetAdapterImageField* a besoin des valeurs de clé nécessaires à l'identification de l'enregistrement contenant l'image.

Réponse image

L'objet réponse image contient l'objet *TWebResponse*. Les champs d'adaptation répondent à une requête d'adaptation en écrivant une image dans l'objet réponse Web.

La figure suivante illustre comment les champs d'adaptation d'image répondent à une requête.

Figure 29.3 Réponse à une requête d'image



Répartition des éléments d'action

Le répartiteur Web (*TWebDispatcher*) recherche dans sa liste d'éléments d'action celui qui :

- correspond à la partie Pathinfo du message de requête de l'URL cible ;
- peut fournir le service spécifié par la méthode du message de requête.

Pour ce faire, il compare les propriétés *PathInfo* et *MethodType* de l'objet *TWebRequest* avec les propriétés de même nom des éléments d'action.

Quand le répartiteur trouve l'élément d'action approprié, il provoque le déclenchement de l'élément d'action. Quand l'élément d'action est déclenché, il effectue l'une des opérations suivantes :

- Il remplit le contenu de la réponse et envoie la réponse, ou signale que la requête a été entièrement traitée.

- Il ajoute du contenu à la réponse, puis permet à d'autres éléments d'action de poursuivre le travail.
- Il délègue la requête à d'autres éléments d'action.

Une fois que le répartiteur a vérifié tous les éléments d'action, si le message n'a pas été correctement traité, il recherche dans les composants d'auto-répartition recensés n'utilisant pas d'éléments d'action. Ces composants sont spécifiques aux applications de bases de données multiniveaux. Si le message de requête n'est toujours pas entièrement traité, le répartiteur appelle l'élément d'action par défaut. Il n'est pas nécessaire que l'élément d'action par défaut corresponde à l'URL cible ou à la méthode de la requête.

Fonctions du répartiteur de page

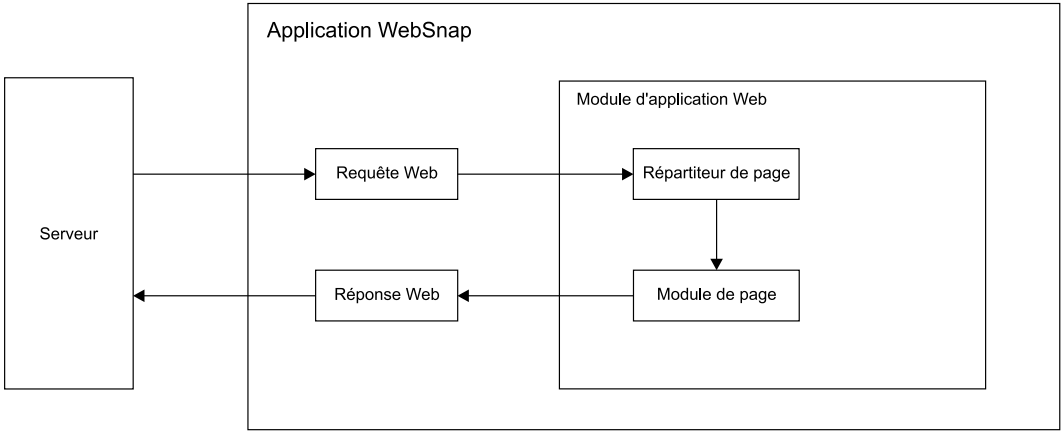
Quand le répartiteur de page reçoit une requête d'un client, il détermine le nom de page en vérifiant la partie Pathinfo du message de requête de l'URL cible. Si elle n'est pas vide, le répartiteur de page en utilise le dernier mot comme nom de page. Si la partie pathinfo est vide, le répartiteur de page tente de déterminer un nom de page par défaut.

Si la propriété *DefaultPage* du répartiteur de page contient un nom de page, le répartiteur utilise ce nom comme nom de page par défaut. Si la propriété *DefaultPage* n'est pas renseignée et si le module d'application Web est un module de page, le répartiteur de page utilise le nom de module de l'application Web comme nom de page par défaut.

Si le nom de page n'est pas vide, le répartiteur de page recherche un module de page Web ayant le nom correspondant. S'il trouve un module de page Web, il appelle ce module pour générer une réponse. Si le nom de page est vide, ou si le répartiteur de page ne trouve pas un module de page Web, le répartiteur de page déclenche une exception.

La figure suivante illustre comment le répartiteur de page répond à une requête.

Figure 29.4 Répartition d'une page



Tutoriel WebSnap

La section suivante décrit les étapes de la conception d'une application WebSnap. Réalisez ce tutoriel pour vous familiariser avec l'architecture WebSnap et ses nouveaux concepts en incorporant les nouveaux composants répartiteur et adaptateur dans un module de page Web. L'application WebSnap illustre la manière d'utiliser les composants HTML WebSnap pour concevoir une application qui permet de modifier le contenu d'une table.

Création d'une nouvelle application

Pour créer une nouvelle application WebSnap :

Etape 1. Démarrage de l'expert application WebSnap

1 Démarrez Delphi et choisissez Fichier | Nouveau | Autre.

2 Dans la boîte de dialogue Nouveaux éléments, sélectionnez l'onglet WebSnap et choisissez Application WebSnap.

3 Dans la boîte de dialogue Nouvelle application WebSnap :

- Sélectionnez Exécutable débogueur d'application Web.
- Dans la zone CoClasse, saisissez **CountryTutorial.**
- Sélectionnez Module Page comme type de composant.
- Dans la zone Nom de page, saisissez **Home.**

4 Choisissez OK.

Etape 2. Enregistrement des fichiers générés et du projet

Pour enregistrer les fichiers unités Pascal et le projet :

1 Sélectionnez Fichier | Tout enregistrer.
2 Dans la zone Nom de fichier, entrez **HomeU.pas** et choisissez OK.
3 Dans la zone Nom de fichier, entrez **CountryU.pas** et choisissez OK.
4 Dans la zone Nom de fichier, entrez **CountryTutorial.dpr** et choisissez OK.
5 Choisissez OK.

Remarque Enregistrez l'unité et le projet dans le même répertoire car l'application recherche le fichier HomeU.html dans le même répertoire que l'exécutable.

Etape 3. Spécification du titre de l'application

Le titre de l'application est le nom affiché dans le navigateur de l'utilisateur final. Pour spécifier le titre de l'application :

1 Sélectionnez Voir | Gestionnaire de projet.

2 Dans la fenêtre du gestionnaire de projet, développez CountryTutorial.exe et double-cliquez sur l'entrée HomeU.

3 Dans la fenêtre de l'inspecteur d'objets (en bas à gauche), sélectionnez ApplicationAdapter dans la liste déroulante.

4 Dans l'onglet Propriétés, saisissez **Tutoriel Country** dans la zone ApplicationTitle.

5 Cliquez sur l'onglet Prévisualiser dans la fenêtre de l'éditeur. Le titre de l'application est affiché en haut de la page.

Création de la page CountryTable

Un module de page Web va servir à définir une page publiée et servir de conteneur aux composants de données.

Etape 1. Ajout d'un nouveau module

Pour ajouter un nouveau module :

1 Sélectionnez Fichier | Nouveau | Autre.

2 Dans la boîte de dialogue Nouveaux éléments, sélectionnez l'onglet WebSnap et choisissez Module de page WebSnap.

3 Dans la boîte de dialogue, choisissez AdapterPageProducer comme valeur pour la liste Type de générateur.

4 Dans la zone Nom de page, saisissez **CountryTable.**

5 Laissez vide les autres champs, et laissez aux sélections leur valeur par défaut.

6 Choisissez OK.

Etape 2. Enregistrement du nouveau module

Enregistrez l'unité dans le répertoire du fichier projet. A l'exécution de l'application, elle recherche le fichier CountryTableU.html dans le même répertoire que l'exécutable.

1 Sélectionnez Fichier | Enregistrer.
2 Dans la zone Nom de fichier, entrez **CountryTableU.pas** et choisissez OK.

Ajout des composants de données au module CountryTable

Un composant *TTable* fournit les données du tableau HTML. Le composant *TDataSetAdapter* permet au script serveur d'accéder au composant *TTable*.

Etape 1. Ajout des composants orientés données

1 Sélectionnez Voir | Gestionnaire de projet.

2 Dans la fenêtre du gestionnaire de projet, développez CountryTutorial.exe et double-cliquez sur l'entrée CountryTableU.

3 Sélectionnez Voir | Arborescence d'objets. La fenêtre vue arborescente de l'objet (côté gauche) devient active.

4 Sélectionnez l'onglet AccèsBD de la palette d'outils.

5 Sélectionnez un composant Table (cliquez dessus avec le bouton gauche de la souris et maintenez-le enfoncé) et faites glisser le composant dans la fenêtre vue arborescente de l'objet.

6 Sélectionnez un composant Session (cliquez dessus avec le bouton gauche de la souris et maintenez-le enfoncé) et faites glisser le composant dans la fenêtre vue arborescente de l'objet.

7 Sélectionnez le composant Session dans la fenêtre vue arborescente de l'objet. Cela affiche les valeurs du composant Session dans la fenêtre Inspecteur d'objets.

8 Dans l'inspecteur d'objets, initialisez la propriété AutoSessionName à *True*.

9 Sélectionnez le composant Table dans la fenêtre vue arborescente de l'objet. Cela affiche les valeurs du composant Table dans la fenêtre Inspecteur d'objets.

10 Dans l'inspecteur d'objets, initialisez les propriétés suivantes :

- *Active* à *True*.
- *DatabaseName* à DBDEMOS.
- Dans la propriété *Name*, saisissez **Country**.
- *TableName* à country.db.

Remarque Le composant Session est nécessaire car nous utilisons le composant BDE (*TTable*) dans une application multithread.

Etape 2. Spécification du champ clé

Le champ clé est utilisé pour identifier les enregistrements d'une table. Cela devient important lors de l'ajout d'une page de modification à l'application. Pour spécifier un champ clé :

1 Dans la fenêtre vue arborescente de l'objet, développez les nœuds Session et DBDemos, et sélectionnez le nœud country.db. Ce nœud correspond au composant table Country.

2 Cliquez avec le bouton droit de la souris sur le nœud country.db et sélectionnez Editeur de champs.

3 Cliquez avec le bouton droit de la souris dans la fenêtre éditeur de CountryTable.Country et sélectionnez la commande Ajouter tous les champs.

4 Sélectionnez le champ Name dans la liste des champs ajoutés.

5 Dans l'inspecteur d'objets, développez la propriété *ProviderFlags*.

6 Initialisez la propriété pfInKey à *True*.

Etape 3. Ajout d'un composant adaptateur

Pour rendre les données d'un composant *TTable* accessibles aux scripts côté serveur, vous devez inclure un composant DataSetAdapter (*TDataSetAdapter*). Pour ce faire :

1 Sélectionnez l'onglet WebSnap de la palette d'outils.

2 Sélectionnez le composant DataSetAdapter (cliquez dessus avec le bouton gauche de la souris et maintenez-le enfoncé) et faites glisser le composant dans la fenêtre vue arborescente de l'objet.

3 Dans l'inspecteur d'objets, modifiez les propriétés suivantes :

- Initialisez DataSet à Country.
- Dans la propriété Name, saisissez **Adapter**.

Création d'une grille pour afficher les données

Le *AdapterPageProducer* permet d'exploiter les scripts côté serveur afin de construire rapidement un tableau HTML.

Etape 1. Ajout d'une grille

Pour ajouter une grille afin d'afficher les données de la table :

1 Sélectionnez Voir | Gestionnaire de projet.

2 Dans la fenêtre du gestionnaire de projet, développez CountryTutorial.exe et double-cliquez sur l'entrée CountryTableU.

3 Sélectionnez Voir | Arborescence d'objets. Le fenêtre vue arborescente de l'objet (côté gauche) devient active.

4 Développez le composant *AdapterPageProducer*.

5 Cliquez avec le bouton droit de la souris sur l'entrée WebPageItems et choisissez Nouveau composant.

6 Dans la fenêtre Ajout de composant Web, sélectionnez AdapterForm, puis choisissez OK. Un composant AdapterForm1 apparaît dans la fenêtre vue arborescente de l'objet.

7 Cliquez avec le bouton droit de la souris sur AdapterForm1 et choisissez Nouveau composant.

8 Dans la fenêtre Ajout de composant Web, sélectionnez AdapterGrid, puis choisissez OK. Un composant AdapterGrid1 apparaît dans la fenêtre vue arborescente de l'objet.

9 Dans l'inspecteur d'objets, initialisez la propriété Adapter à Adapter.

10 Pour prévisualiser la page, sélectionnez l'onglet CountryTableU.pas dans la fenêtre éditeur de code et choisissez en bas l'onglet Prévisualiser. Si l'onglet Prévisualiser n'apparaît pas, faites défiler les onglets.

11 Sélectionnez l'onglet Script HTML pour visualiser le code JScript généré par les composants WebSnap.

Etape 2. Ajout de commandes de modification à la grille

Les utilisateurs peuvent avoir besoin de modifier le contenu de la table. Pour permettre aux utilisateurs de faire des changements (modification, ajout ou suppression de ligne), ajoutez des composants commande.

Pour ajouter des composants commande :

1. Dans la fenêtre vue arborescente de l'objet pour CountryTable, développez le composant *AdapterPageProducer* et toutes ses branches.
2. Cliquez avec le bouton droit de la souris sur le composant AdapterGrid1 et choisissez Nouveau composant. Une entrée AdapterCommandColumn1 est ajoutée au composant AdapterGrid1.
3. Cliquez avec le bouton droit de la souris sur AdapterCommandColumn1 et choisissez Ajouter des commandes.
4. Sélectionnez les commandes DeleteRow, EditRow et NewRow puis choisissez OK.
5. Pour prévisualiser la page, cliquez sur l'onglet Prévisualiser en bas de l'éditeur de code.

Ajout d'une fiche de modification

Créez un module de page Web qui va servir de fiche de modification de la table country.

Etape 1. Ajout d'un nouveau module

Pour ajouter un nouveau module de page WebSnap :

1. Sélectionnez Fichier | Nouveau | Autre.
2. Dans la boîte de dialogue Nouveaux éléments, sélectionnez l'onglet WebSnap et choisissez Module de page WebSnap.
3. Dans la boîte de dialogue, choisissez AdapterPageProducer comme valeur pour la liste Type de générateur.
4. Dans la zone Nom de page, saisissez **CountryForm.**
5. Ne cochez pas la case Publié.
6. Laissez vide les autres champs et laissez aux sélections leur valeur par défaut.
7. Choisissez OK.

Etape 2. Enregistrement du nouveau module

Enregistrez l'unité dans le répertoire du fichier projet. A l'exécution de l'application, elle recherche le fichier CountryFormU.html dans le même répertoire que l'exécutable.

1. Sélectionnez Fichier | Enregistrer.
2. Dans la zone Nom de fichier, entrez **CountryFormU.pas** et choisissez OK.

Etape 3. Utilisation de l'unité CountryTableU

Ajoutez l'unité CountryTableU à la clause **uses** afin de permettre au module d'accéder au composant adaptateur :

1. Sélectionnez Fichier | Utiliser l'unité.
2. Sélectionnez la CountryTableU dans la liste et choisissez OK.

Etape 4. Ajout des zones de saisie

Ajoutez des composants au composant *AdapterPageProducer* afin de générer les zones de saisie dans le formulaire HTML.

Pour ajouter des zones de saisie :

1. Sélectionnez Voir | Gestionnaire de projet.
2. Dans la fenêtre vue arborescente de l'objet, développez CountryTutorial.exe et double-cliquez sur l'entrée CountryFormU.
3. Sélectionnez Voir | Arborescence d'objets. La fenêtre vue arborescente de l'objet (côté gauche) devient active.
4. Dans la fenêtre vue arborescente de l'objet, développez le composant *AdapterPageProducer*, cliquez avec le bouton droit de la souris sur WebPageItems et choisissez Nouveau composant.
5. Sélectionnez AdapterForm puis choisissez OK. Une entrée AdapterForm1 apparaît dans la fenêtre vue arborescente de l'objet.
6. Cliquez avec le bouton droit de la souris sur WebPageItems et choisissez Nouveau composant.
7. Sélectionnez AdapterFieldGroup puis choisissez OK. Une entrée AdapterFieldGroup1 apparaît dans la fenêtre vue arborescente de l'objet.
8. Dans l'inspecteur d'objets, initialisez la propriété Adapter à CountryTable.Adapter.
9. Pour prévisualiser la page, cliquez sur l'onglet Prévisualiser en bas de l'éditeur de code.

Etape 5. Ajout de boutons

Ajoutez des composants au composant *AdapterPageProducer* afin de générer les boutons de validation dans le formulaire HTML. Pour ajouter des composants :

1. Dans la fenêtre vue arborescente de l'objet, développez le composant *AdapterPageProducer* et toutes ses branches.
2. Cliquez avec le bouton droit de la souris sur AdapterForm1 et choisissez Nouveau composant.
3. Sélectionnez AdapterCommandGroup puis choisissez OK. Une entrée AdapterCommandGroup1 apparaît dans la fenêtre vue arborescente de l'objet.
4. Dans l'inspecteur d'objets, initialisez la propriété DisplayComponent à AdapterFieldGroup1.

5 Cliquez avec le bouton droit de la souris sur l'entrée AdapterCommandGroup1 et choisissez Ajouter des commandes.

6 Sélectionnez les commandes Cancel, Apply et Refresh Row puis choisissez OK.

7 Pour prévisualiser la page, sélectionnez l'onglet CountryTableU.pas dans la fenêtre éditeur de code et choisissez en bas l'onglet Prévisualiser. Si l'onglet Prévisualiser n'apparaît pas, faites défiler les onglets.

Etape 6. Liaisons des actions du formulaire à la page de la grille

Quand l'utilisateur clique sur un bouton, une action d'adaptation est exécutée. Pour spécifier la page à afficher une fois l'action d'adaptation exécutée :

1 Dans la fenêtre vue arborescente de l'objet, développez AdapterCommandGroup1 afin d'afficher les entrées CmdCancel, CmdApply et CmdRefreshRow.

2 Sélectionnez CmdCancel. Dans l'inspecteur d'objets, saisissez **CountryTable** dans la propriété PageName.

3 Sélectionnez CmdApply. Dans l'inspecteur d'objets, saisissez **CountryTable** dans la propriété PageName.

Etape 7. Liaison des actions de la grille à la page formulaire

Pour spécifier la page à afficher quand une action d'adaptation est exécutée en choisissant un bouton de la grille :

1 Sélectionnez Voir | Gestionnaire de projet.

2 Dans la fenêtre vue arborescente de l'objet, développez CountryTutorial.exe et double-cliquez sur l'entrée CountryTableU.

3 Dans la fenêtre vue arborescente de l'objet, développez le composant *AdapterPageProducer* et toutes ses branches afin d'afficher les entrées CmdNewRow, CmdEditRow et CmdDeleteRow. Ces entrées apparaissent en dessous de l'entrée AdapterCommandColumn1.

4 Sélectionnez CmdNewRow. Dans l'inspecteur d'objets, saisissez **CountryForm** dans la propriété PageName.

5 Sélectionnez CmdEditRow. Dans l'inspecteur d'objets, saisissez **CountryForm** dans la propriété PageName.

6 Pour vérifier que l'application fonctionne et que tous les boutons effectuent une action, exécutez l'application.

Remarque Il n'y a pas d'indication sur les erreurs de bases de données, par exemple un type incorrect. Pour le vérifier, essayez d'ajouter un nouveau pays avec un code incorrect (par exemple 'abc') dans le champ Area.

Ajout de la gestion des erreurs

Pour signaler les erreurs à l'utilisateur final, ajoutez un composant AdapterErrorList qui sert à afficher une liste des erreurs qui se sont produites lors de l'exécution des actions d'adaptation qui modifient la table country.

Etape 1. Ajout de la gestion des erreurs à la grille

Dans la fenêtre vue arborescente de l'objet pour CountryTable, développez le composant *AdapterPageProducer* et toutes ses branches afin d'afficher AdapterForm1.

1. Cliquez avec le bouton droit de la souris sur AdapterForm1 et choisissez Nouveau composant.
2. Sélectionnez AdapterErrorList puis choisissez OK. Une entrée AdapterErrorList1 apparaît dans la fenêtre vue arborescente de l'objet.
3. Déplacez AdapterErrorList1 au-dessus de AdapterGrid1 (en le faisant glisser ou en utilisant la flèche orientée vers le haut dans la barre d'outils de la vue arborescente de l'objet).
4. Dans l'inspecteur d'objets, initialisez la propriété Adapter à Adapter.

Etape 2. Ajout de la gestion des erreurs au formulaire

1. Dans la fenêtre vue arborescente de l'objet pour CountryTable, développez le composant *AdapterPageProducer* et toutes ses branches afin d'afficher AdapterForm1.
2. Cliquez avec le bouton droit de la souris sur AdapterForm1 et choisissez Nouveau composant.
3. Sélectionnez AdapterErrorList puis choisissez OK. Une entrée AdapterErrorList1 apparaît dans la fenêtre vue arborescente de l'objet.
4. Déplacez AdapterErrorList1 au-dessus de AdapterGrid1 (en le faisant glisser ou en utilisant la flèche orientée vers le haut dans la barre d'outils de la vue arborescente de l'objet).
5. Dans l'inspecteur d'objets, initialisez la propriété Adapter à CountryTable.Adapter.

Etape 3. Test de la gestion d'erreurs

Pour tester la gestion d'erreur de la grille :

1. Exécutez l'application et placez-vous sur la page CountryTable.
2. Démarrez une autre instance de votre navigateur et placez-vous sur la page CountryTable.
3. Cliquez sur le bouton DeleteRow pour la première ligne de la grille.
4. Sans rafraîchir dans le second navigateur, cliquez sur le bouton DeleteRow pour la première ligne de la grille.

5 Un message d'erreur est affiché au-dessus de la grille.

Pour tester la gestion d'erreur du formulaire :

1 Exécutez l'application et placez-vous sur la page CountryTable.

2 Choisissez le bouton EditRow.

3 La page CountryForm est affichée.

4 Changez la zone Area en 'abc' et choisissez le bouton Apply.

5 Un message d'erreur est affiché au-dessus du premier champ.

Exécution de l'application

Une fois l'application terminée, pour l'exécuter :

1 Sélectionnez Exécuter | Exécuter. Une fenêtre est alors affichée. Les applications Exécutable débogueur d'application Web sont des serveurs COM, la fenêtre qui apparaît alors est la fenêtre console du serveur COM. A la première exécution du projet, les objets COM pouvant être utilisés directement par le débogueur d'application Web sont recensés.

2 Sélectionnez Outils | Débogueur d'application Web.

3 Cliquez sur le lien URL par défaut pour afficher la page ServerInfo. La page ServerInfo affiche le nom de tous les exécutables débogueur d'application Web recensés.

4 Sélectionnez CountryTutorial dans la liste déroulante et choisissez le bouton Go.

Chapitre

30

Utilisation de documents XML

XML (Extensible Markup Language) est un langage de balisage pour la description de données structurées. Il ressemble au HTML, à cette différence près que les balises décrivent la structure des informations et non leurs caractéristiques d'affichage. Les documents XML proposent un moyen simple, sous forme de texte, de stocker des informations qui peuvent être simplement modifiées ou recherchées. Ils sont souvent utilisés comme format standard et transportable pour les données d'applications Web, les communication business-to-business, etc.

Les documents XML donnent une vue hiérarchisée du corps des données. Les balises d'un document XML décrivent le rôle et la signification de chaque élément de données, comme le montre le document suivant qui décrit une collection d'actions boursières :

```
<?xml version="1.0" standalone='yes' ?>
<!DOCTYPE stockholdings SYSTEM "sth.dtd">
<StockList>
   <Stock exchange=NASDAQ>
      <name>Inprise (Borland)</name>
      <price>15.375</price>
      <symbol>INPR</symbol>
      <shares>100</shares>
   </Stock>
   <Stock exchange=NYSE>
      <name>Pfizer</name>
      <price>42.75</price>
      <symbol>PFE</symbol>
      <shares type=preferred>25</shares>
   </Stock>
</StockList>
```

Cet exemple illustre des éléments typiques d'un document XML. La première ligne est une instruction de traitement. Elle donne des informations sur la manière d'interpréter le fichier mais ne contient aucune donnée.

La seconde ligne, qui commence avec la balise <!DOCType>, est une déclaration de type de document. Elle nomme la structure du document et référence un autre fichier (sth.dtd) qui décrit cette structure. Dans ce cas, la structure est définie par un fichier DTD (Document Type Definition, définition de type de document). Il existe d'autres types de fichiers décrivant la structure d'un document XML : XDR (Reduced XML Data) et XSD (XML schemas).

Les lignes restantes sont organisées de manière hiérarchique avec un seul nœud racine (la balise <StockList>). Chaque nœud de la hiérarchie contient soit un ensemble de nœuds enfant, soit une valeur texte. Certaines balises (les balises <Stock> et <shares>) contiennent des attributs qui sont des paires Nom=Valeur donnant des détails sur la manière d'interpréter la balise.

Même s'il est possible de travailler directement sur le texte d'un document XML, les applications utilisent généralement des outils spécifiques pour analyser et modifier les données. Le W3C définit un ensemble d'interfaces standard pour représenter un document XML analysé, appelé le modèle d'objet document (DOM, Document Object Model). Divers éditeurs proposent des analyseurs XML qui implémentent les interfaces DOM afin de vous permettre d'interpréter et de modifier plus facilement les documents XML.

Delphi propose plusieurs outils pour manipuler les documents XML. Ces outils utilisent un analyseur DOM fournis par un tiers et simplifient l'utilisation des documents XML. Ce chapitre décrit ces outils.

Remarque Outre les outils décrits dans ce chapitre, Delphi dispose de composants et d'outils assurant la conversion de documents XML en paquets de données afin de les intégrer dans l'architecture de base de données Delphi. Pour davantage d'informations sur les outils d'intégration de documents XML dans des applications de bases de données, voir chapitre 26, "Utilisation de XML dans les applications de bases de données".

Utilisation du modèle DOM

DOM (Document Object Model) est un ensemble d'interfaces standard définissant la manière de représenter un document XML analysé. Delphi est livré avec deux implémentations de DOM (celle de Microsoft et celle d'IBM). De plus, il propose un mécanisme de recensement qui vous permet d'intégrer à Delphides implémentations de DOM proposées par d'autres fournisseurs.

L'unité XMLDOM contient la déclaration de toutes les interfaces DOM définies dans la spécification W3C XML DOM niveau 2. Chaque fournisseur DOM utilise une implémentation de ces interfaces.

- Pour utiliser l'implémentation Microsoft, incluez l'unité MSXMLDOM dans votre clause uses. Comme l'implémentation Microsoft utilise COM, vous devez également recenser la bibliothèque msxml.dll comme serveur COM. Vous pouvez utiliser Regsvr32.exe pour recenser cette DLL.
- Pour utiliser l'implémentation IBM, incluez l'unité IBMXMLDOM dans votre clause uses.

- Pour utiliser d'autres implémentations de DOM, vous devez créer une unité qui inclut une fonction renvoyant l'interface de niveau le plus élevé (*IDOMImplementation*). Votre unité doit recenser cette fonction en appelant la procédure globale *RegisterDOMImplementation*.

Certains fournisseurs proposent des extensions aux interfaces standard DOM. Pour pouvoir utiliser ces extensions, l'unité XMLDOM définit également une interface *IDOMNodeEx*. *IDOMNodeEx* est un descendant de l'interface standard *IDOMNode* qui inclut les plus utiles de ces extensions.

Vous pouvez manipuler directement les interfaces DOM pour analyser et modifier des documents XML. Appelez simplement la fonction *GetDOM* pour obtenir une interface *IDOMImplementation* que vous pouvez utiliser comme point de départ.

Remarque Pour une description détaillée des interfaces DOM, voir les déclarations dans l'unité XMLDOM, la documentation de votre fournisseur DOM ou les spécifications fournies par le site Web W3C (www.w3.org).

Vous pouvez choisir d'utiliser les classes XML Delphi plutôt que de manipuler directement les interfaces DOM. Elles sont décrites ci-dessous

Utilisation des composants XML

Delphi définit diverses classes et interfaces pour manipuler les documents XML. Elles simplifient le processus de chargement, de modification et d'enregistrement des documents XML.

Utilisation de TXMLDocument

Le composant *TXMLDocument* sert de point de départ à l'utilisation d'un document XML. Les étapes suivantes décrivent comment utiliser *TXMLDocument* pour manipuler directement un document XML :

1. Ajoutez un composant *TXMLDocument* dans votre fiche ou votre module de données. *TXMLDocument* apparaît dans la page Internet de la palette des composants.
2. Initialisez la propriété *DOMVendor* afin de spécifier l'implémentation de DOM que vous souhaitez que le composant utilise pour analyser et modifier un document XML. L'inspecteur d'objets liste tous les fournisseurs DOM recensés. Pour davantage d'informations sur les implémentations DOM, voir "Utilisation du modèle DOM" à la page 30-2.
3. Selon votre implémentation, vous pouvez initialiser la propriété *ParseOptions* pour configurer comment l'implémentation sous-jacente de DOM analyse le document XML.

4 Si vous manipulez un document XML, spécifiez le document de la façon suivante :

- Si le document XML est stocké dans un fichier, initialisez la propriété *FileName* avec le nom de ce fichier.
- Vous pouvez spécifier le document XML sous forme de chaîne au lieu d'utiliser un fichier grâce à la propriété *XML*.

5 Initialisez la propriété *Active* à *True*.

Quand vous disposez d'un objet *TXMLDocument* actif, vous pouvez parcourir la hiérarchie de ses nœuds en lisant ou en modifiant leur valeur. Le nœud racine de cette hiérarchie est accessible via la propriété *DocumentElement*.

Utilisation des nœuds XML

Quand un document XML a été analysé par une implémentation de DOM, les données qu'il représente sont accessibles via une hiérarchie de nœuds. Chaque nœud correspond à un élément balisé du document. Par exemple, étant donné le code XML suivant :

```
Component palette<?xml version="1.0" standalone='yes' ?>
<!DOCTYPE stockholdings SYSTEM "sth.dtd">
<StockList>
   <Stock exchange=NASDAQ>
      <name>Inprise (Borland)</name>
      <price>15.375</price>
      <symbol>INPR</symbol>
      <shares>100</shares>
   </Stock>
   <Stock exchange=NYSE>
      <name>Pfizer</name>
      <price>42.75</price>
      <symbol>PFE</symbol>
      <shares type=preferred>25</shares>
   </Stock>
</StockList>
```

TXMLDocument génère la hiérarchie de nœuds suivante : la racine de la hiérarchie est le nœud *StockList*. *StockList* contient deux nœuds enfant qui correspondent aux deux balises *Stock*. Chacun de ces nœuds enfant a quatre nœuds enfant (*name*, *price*, *symbol* et *shares*). Ces quatre nœuds enfant sont des nœuds feuille, ou nœuds terminaux. Le texte qu'ils contiennent apparaît comme valeur de chaque nœud feuille.

Remarque Cette division en nœuds peut varier selon la manière dont l'implémentation de DOM génère les nœuds pour un document XML. En particulier, un analyseur DOM traite tous les éléments balisés comme des nœuds internes. Des nœuds supplémentaires (de type nœud texte) sont créés pour les valeurs des nœuds *name*, *price*, *symbol* et *shares*. Ces nœuds texte apparaissent alors comme des enfants des nœuds *name*, *price*, *symbol et shares*.

Pour accéder à chaque nœud, utilisez une interface *IXMLNode* en partant du nœud racine qui est la valeur de la propriété *DocumentElement* du composant document XML.

Utilisation de la valeur d'un nœud

Etant donné une interface *IXMLNode*, vous pouvez vérifier si elle représente un nœud interne ou un nœud feuille en testant la valeur de la propriété *IsTextElement*.

- Si elle représente un nœud feuille, vous pouvez lire ou écrire sa valeur en utilisant la propriété *Text*.
- Si elle représente un nœud interne, vous pouvez accéder à ses nœuds enfant en utilisant la propriété *ChildNodes*.

Ainsi, par exemple, en utilisant le document XML précédent, vous pouvez lire la valeur de l'action Inprise de la manière suivante :

```
InpriseStock := XMLDocument1.DocumentElement.ChildNodes[0];
Price := InpriseStock.ChildNodes['price'].Text;
```

Utilisation des attributs d'un nœud

Si le nœud a des attributs, vous pouvez les manipuler en utilisant la propriété *Attributes*. Vous pouvez lire ou écrire la valeur des attributs en spécifiant un nom d'attribut existant. Vous pouvez ajouter de nouveaux attributs en spécifiant le nom du nouvel attribut lorsque vous initialisez la propriété *Attributes* :

```
InpriseStock := XMLDocument1.DocumentElement.ChildNodes[0];
InpriseStock.ChildNodes['shares'].Attributes['type'] := 'common';
```

Ajout et suppression de nœuds enfant

Vous pouvez ajouter des nœuds enfant en utilisant la méthode *AddChild*. *AddChild* crée de nouveau nœuds correspondants aux éléments balisés du document XML. De tels nœuds sont appelées nœuds élément.

Pour créer un nouveau nœud élément, spécifiez le nom qui doit apparaître dans la nouvelle balise et, éventuellement, la position où le nouveau nœud doit apparaître. Par exemple, le code suivant ajoute une nouvelle cotation au document précédent :

```
var
  NewStock: IXMLNode;
  ValueNode: IXMLNode;
begin
  NewStock := XMLDocument1.DocumentElement.AddChild('stock');
  NewStock.Attributes['exchange'] := 'NASDAQ';
  ValueNode := NewStock.AddChild('name');
  ValueNode.Text := 'Cisco Systems'
  ValueNode := NewStock.AddChild('price');
  ValueNode.Text := '62.375';
  ValueNode := NewStock.AddChild('symbol');
  ValueNode.Text := 'CSCO';
  ValueNode := NewStock.AddChild('shares');
```

```
  ValueNode.Text := '25';
end;
```

Une version redéfinie de *AddChild* vous permet de spécifier le domaine d'appellation dans lequel le nom de balise est défini.

Vous pouvez supprimer des nœuds enfant en utilisant les méthodes de la propriété *ChildNodes*. *ChildNodes* est une interface *IXMLNodeList* qui gère les enfants d'un nœud. Vous pouvez utiliser sa méthode *Delete* pour supprimer un seul nœud enfant qui est identifié par sa position ou par son nom. Par exemple, le code suivant supprime la dernière cotation énumérée dans le document précédent :

```
StockList := XMLDocument1.DocumentElement;
StockList.ChildNodes.Delete(StockList.ChildNodes.Count - 1);
```

Abstraction de documents XML avec l'expert liaison de données

Même s'il est possible de manipuler un document XML en utilisant uniquement le composant *TXMLDocument* et l'interface *IXMLNode* pour parcourir les nœuds du document, ou même d'utiliser uniquement les interfaces DOM (sans même utiliser *TXMLDocument*), vous pouvez écrire du code beaucoup plus simple et plus lisible en utilisant l'expert Liaison de données XML.

L'expert Liaison de données XML part d'un schéma XML ou d'un fichier de données et génère un ensemble d'interfaces correspondantes. Soit, par exemple, les données XML suivantes :

```
<customer id=1>
  <name>Mark</name>
  <phone>(831) 431-1000</phone>
</customer>
```

L'expert Liaison de données XML génère l'interface suivante (ainsi que la classe qui l'implémente) :

```
ICustomer = interface(IXMLNode)
  property id: Integer read Getid write Setid;
  property name: DOMString read Getname write Setname;
  property phone: DOMString read Getphone write Setphone;
  function Getid: Integer;
  function Getname: DOMString;
  function Getphone: DOMString;
  procedure Setid(Value: Integer);
  procedure Setname(Value: DOMString);
  procedure Setphone(Value: DOMString);
end;
```

Chaque nœud enfant est associé à une propriété dont le nom correspond au nom de la balise du nœud enfant et dont la valeur est l'interface du nœud enfant (si l'enfant est un nœud interne) ou la valeur du nœud enfant (pour les nœuds feuille). Chaque attribut de nœud est également associé à une propriété, le nom de la propriété étant le nom de l'attribut et sa valeur étant celle de l'attribut.

Outre la création des interfaces (et des classes d'implémentation) pour chaque élément balisé du document XML, l'expert crée une fonction globale pour obtenir une interface sur le nœud racine. Si, par exemple, le code XML précédent provient d'un document dont le nœud racine est la balise <Customers>, l'expert Liaison de données crée la routine globale suivante :

```
function GetCustomers(XMLDoc: TXMLDocument): ICustomers;
```

L'utilisation des interfaces générées simplifie votre code car il reflète plus directement la structure du document XML. Ainsi, au lieu d'écrire du code comme le suivant :

```
CustName := XMLDocument1.DocumentElement.ChildNodes[0].ChildNodes['name'].Value;
```

Votre code peut avoir la forme suivante :

```
CustName := GetCustomers(XMLDocument1)[0].Name;
```

Remarquez que les interfaces générées par l'expert Liaison de données descendent toutes de *IXMLNode*. Cela signifie que vous pouvez toujours ajouter ou supprimer des nœuds enfant comme si vous n'utilisiez pas l'expert Liaison de données (voir "Ajout et suppression de nœuds enfant" à la page 30-5.) De plus, quand les nœuds enfant représentent des éléments répétés (tous les enfants d'un nœud ayant le même type), le nœud parent dispose de deux méthodes *Add* et *Insert*, pour ajouter d'autres répétitions. Ces méthodes sont plus simples que *AddChild* car il n'est pas nécessaire de spécifier le type de nœud à créer.

Utilisation de l'expert Liaison de données XML

Pour utiliser l'expert Liaison de données :

1. Choisissez Fichier | Nouveau | Autre et sélectionnez l'icône intitulée Expert Liaison de données XML dans la page Nouveaux.
2. L'expert Liaison de données XML s'affiche.
3. Dans la première page de l'expert :
   - Spécifiez le document XML ou le schéma pour lequel vous voulez générer les interfaces. Ce peut être un document XML, un fichier de définition de type (.dtd), un fichier de données XML abrégées (.xdr) ou un fichier schéma XML (.xsd).
4. Choisissez le bouton Options pour spécifier la stratégie de nomenclature que doit utiliser l'expert lors de la génération des interfaces et des classes d'implémentation ainsi que les correspondances par défaut entre les types définis dans le schéma et les types de données Pascal.
5. Passez à la seconde page de l'expert. Cette page vous permet de fournir des informations détaillées sur chaque type de nœud du document ou du schéma. A gauche, se trouve une vue arborescente montrant tous les types de nœuds du document. Pour les nœuds complexes (les nœuds ayant des enfants), la vue arborescente peut être développée pour afficher les éléments enfant. Quand vous sélectionnez un nœud dans cette vue arborescente, le volet de

droite de la boîte de dialogue affiche des informations sur ce nœud et vous permet de spécifier comme l'expert doit le traiter.

- La zone Nom de la source affiche le nom du type de nœud dans le schéma XML.
- La zone Type de données affiche le type de la valeur du nœud comme spécifiée par le schéma XML.
- La zone Documentation vous permet d'associer des commentaires au schéma décrivant l'utilisation ou la fonction du nœud.
- Si l'expert génère du code pour le nœud sélectionné (c'est-à-dire s'il s'agit d'un type complexe pour lequel l'expert génère une interface et une classe d'implémentation ou s'il s'agit d'un des nœuds enfant d'un type complexe pour lequel l'expert génère une propriété dans l'interface du type complexe), vous pouvez utiliser la case à cocher Générer la liaison pour spécifier si l'expert doit gérer du code pour ce nœud. Si vous ne cochez pas la case, l'expert ne génère pas l'interface et la classe d'implémentation pour un type complexe et ne crée pas de propriété dans l'interface parent pour un nœud enfant ou un attribut.
- La section Options de liaison vous permet de contrôler le code généré par l'expert pour l'élément sélectionné. Pour tous les nœuds, vous pouvez spécifier le Nom d'identifiant (le nom de l'interface générée ou de la propriété). De plus, pour les interfaces, vous devez indiquer celle qui représente le nœud racine du document. Pour les nœuds représentant des propriétés, vous pouvez spécifier le type de la propriété et, si la propriété n'est pas une interface, s'il s'agit d'une propriété en lecture seule.

6. Une fois spécifié le code que l'expert doit générer pour chaque nœud, passez à la troisième page. Cette page vous permet de choisir certaines options globales contrôlant comment l'expert génère le code, vous permet de prévisualiser le code qui sera généré et vous permet de demander à conserver vos options pour des utilisations ultérieures.
   - Pour prévisualiser le code généré par l'expert, sélectionnez une interface dans la liste Récapitulatif des liaisons et visualisez la définition d'interface dans la zone Prévisualisation du code.
   - Utilisez la zone Paramètres de liaison des données pour indiquer comment l'expert doit enregistrer vos choix. Vous pouvez stocker vos choix comme annotations dans un fichier schéma associé au document (le fichier schéma spécifié dans la première page de la boîte de dialogue) ou vous pouvez nommer un fichier schéma autonome utilisé uniquement par l'expert.
7. Quand vous choisissez Terminer, l'expert Liaison de données génère une nouvelle unité définissant les interfaces et les classes d'implémentation pour tous les types de nœuds de votre document XML. De plus, il crée une fonction globale qui attend un objet *TXMLDocument* et renvoie l'interface sur le nœud racine de la hiérarchie de données.

Utilisation du code généré par l'expert Liaison de données XML

Quand l'expert a généré l'ensemble d'interfaces et de classes d'implémentation, vous pouvez les utiliser pour manipuler des documents XML correspondants à la structure du document ou du schéma fourni à l'expert. Tout comme pour l'utilisation des composants XML intégrés livrés avec Delphi, votre point de départ est le composant *TXMLDocument* qui apparaît dans la page Internet de la palette des composants.

Pour manipuler un document XML, suivez les étapes ci-après :

1. Placez un composant *TXMLDocument* dans votre fiche ou votre module de données.
2. Liez le *TXMLDocument* à un document XML en initialisant la propriété *FileName* (vous pouvez aussi utiliser à l'exécution une chaîne XML avec la propriété *XML*).
3. Dans votre code, appelez la fonction globale générée par l'expert pour obtenir une interface sur le nœud racine du document XML. Si, par exemple, l'élément racine du document XML est la balise <StockList>, l'expert génère par défaut une fonction *GetStockListType*, qui renvoie une interface *IStockListType* :

```
var
  StockList: IStockListType;
begin
  StockList := GetStockListType(XMLDocument1);
```

4. Cette interface a des propriétés qui correspondent aux sous-nœuds de l'élément racine du document ainsi que des propriétés qui correspondent aux attributs de cet élément racine. Vous pouvez les utiliser pour parcourir la hiérarchie du document XML, modifier les données du document, etc.
5. Pour enregistrer les modifications effectuées en utilisant les interfaces générées par l'expert, appelez la méthode *SaveToFile* du composant *TXMLDocument* ou lisez sa propriété *XML*.

Chapitre

31

Utilisation de services Web

Les services Web sont des applications modulaires autonomes qui peuvent être publiées et invoquées à travers un réseau (comme le Web). Les services Web proposent des interfaces bien définies qui décrivent les services fournis.

Les services Web sont conçus pour permettre un couplage flexible entre client et serveur. C'est-à-dire que l'implémentation du serveur n'impose pas au client l'utilisation d'une plate-forme ou d'un langage de programmation spécifique. Non seulement les interfaces sont définies en restant neutre quant au langage, mais elles sont également conçues pour utiliser différents mécanismes de communication.

La gestion par Delphi des services Web est conçue pour fonctionner en utilisant le protocole SOAP (Simple Object Access Protocol). SOAP est un protocole standard allégé définissant l'échange d'informations dans un environnement décentralisé et distribué. Il utilise le XML pour encoder des appels de procédures distantes et utilise généralement HTTP comme protocole de communication. Pour davantage d'informations sur SOAP, voir ses spécifications à l'adresse suivante :

```
http://www.w3.org/TR/SOAP/
```

Remarque Même si la gestion par Delphi des services Web est basée sur l'utilisation de SOAP et d'HTTP, le modèle est suffisamment souple pour être étendu afin d'utiliser d'autres protocoles d'encodage ou de communications.

La technologie basée sur SOAP proposée par Delphi est disponible pour le moment sous Windows et elle sera ultérieurement implémentée sous Linux, afin de servir de base à des applications distribuées multiplates-formes. Il n'y a pas de runtime client particulier à installer comme c'est le cas pour des applications distribuées utilisant CORBA. Comme cette technologie repose sur des messages HTTP ; elle a cet avantage d'être disponible sur diverses machines. La gestion des services Web constitue le sommet de l'architecture multiplate-forme d'applications serveur Web de Delphi.

Vous pouvez utiliser Delphi pour concevoir aussi bien des serveurs implémentant des services Web que des clients appelant ces serveurs. Si vous

utilisez Delphi pour créer les applications client et serveur, il est possible de partager entre elles une même unité définissant les interfaces des services Web. De plus, vous pouvez avec Delphi écrire des clients pour tout serveur qui implémente des services Web répondant aux messages SOAP ou des serveurs Delphi publiant des services Web utilisables par des clients quelconques.

Si le client ou le serveur n'est pas écrit en utilisant Delphi, vous pouvez publier ou importer les informations décrivant les interfaces disponibles et la manière de les appeler en utilisant un document WSDL (Web Service Definition Language). Côté serveur, votre application peut publier un document WSDL décrivant votre service Web. Côté client, un expert peut importer un document WSDL publié afin de vous proposer les définitions d'interfaces et les informations de connexion dont vous avez besoin.

Conception de serveurs gérant les services Web

Dans Delphi, les serveurs qui gèrent les services Web sont conçus en utilisant des interfaces invocables. Ces interfaces sont des interfaces compilées afin de contenir des informations de type à l'exécution (RTTI). Ces informations sont utilisées pour interpréter les appels de méthodes entrant des clients afin de pouvoir les répartir correctement.

Outre les interfaces invocables, et les classes qui les implémentent, le serveur a besoin de deux composants : un répartiteur et un invocateur. Le répartiteur (*THTTPSoapDispatcher*) est un composant d'auto-répartition qui reçoit les messages SOAP entrant et les transmet à l'invocateur. L'invocateur (*THTTPSoapPascalInvoker*) interprète le message SOAP, identifie l'interface invocable auquel il fait référence, exécute l'appel et assemble le message de réponse.

Remarque *THTTPSoapDispatcher* et *THTTPSoapPascalInvoker* sont conçus pour répondre à des messages HTTP contenant une requête SOAP. Cependant, l'architecture sous-jacente est suffisamment générale pour qu'elle puisse gérer d'autres protocoles en lui substituant d'autres composants répartiteur et invocateur.

Une fois votre interface invocable et les classes d'implémentation recensées, le répartiteur et l'invocateur gèrent automatiquement tous les messages qui identifient ces interfaces dans l'en-tête SOAP Action du message de requête HTTP.

Conception d'un serveur service Web

Suivez les étapes suivantes pour concevoir une application serveur qui implémente un service Web :

1 Définissez les interfaces constituant votre service Web. Ces définitions d'interface doivent être des interfaces invocables. Il est judicieux de créer vos définitions d'interface dans des unités distinctes de celles contenant les classes d'implémentation. Ainsi, l'unité qui définit les interfaces peut être incluse à la fois dans l'application serveur et l'application client. Dans la section initialisation de cette unité, ajoutez du code pour recenser les interfaces. Pour

davantage d'informations sur la conception et le recensement d'interfaces invocables, voir "Définition des interfaces invocables" à la page 31-4.

2. Si votre interface utilise des types complexes (non-scalaires), vous devez vous assurer qu'ils peuvent être transmis correctement. L'application service Web ne peut les gérer qu'en utilisant des objets spéciaux contenant des informations de type à l'exécution (RTTI) qui décrivent leur structure. Pour davantage d'informations sur la création et le recensement d'objets servant à représenter des types complexes, voir "Utilisation de types complexes dans des interfaces invocables" à la page 31-5.
3. Définissez et implémentez les classes qui implémentent les interfaces invocables définies à l'étape 1. Pour chaque classe d'implémentation, il peut être nécessaire de créer une procédure de fabrication qui instancie la classe. Dans la section initialisation de l'unité, ajoutez du code pour recenser la classe d'implémentation. Ce processus est décrit dans la section "Création et recensement de l'implémentation" à la page 31-6.
4. Si votre application déclenche une exception lors d'une tentative d'exécution d'une requête SOAP, l'exception est automatiquement encodée dans un paquet d'erreur SOAP qui est renvoyé à la place du résultat de l'appel de méthode. Si vous voulez fournir davantage d'informations qu'un simple message d'erreur, vous pouvez créer vos propres classes d'exception qui sont encodées et transmises au client. Pour davantage d'informations sur ce sujet, voir "Création de classes d'exception personnalisées pour les services Web" à la page 31-7.
5. Choisissez Fichier | Nouveau | Autre puis, dans la page Services Web, double-cliquez sur l'icône application service Web. Choisissez le type d'application serveur Web que doit implémenter votre service Web. Pour davantage d'informations sur les différents types d'applications serveur Web, voir "Types d'applications serveur Web" à la page 27-6.
6. L'expert génère une nouvelle application service Web qui inclut un composant invocateur (*THTTPSOAPPascalInvoker*) et un composant répartiteur (*THTTPSoapDispatcher*). L'invocateur assure la conversion entres les messages SOAP et les méthodes des interfaces que vous avez définies à l'étape 1. Le répartiteur répond automatiquement aux messages SOAP entrant et les transmet à l'invocateur. Vous pouvez utiliser sa propriété *WebDispatch* pour identifier les messages de requête HTTP que gère votre application. Cela suppose l'initialisation de la propriété *PathInfo* pour indiquer la partie chemin d'accès de toute URL dirigée sur votre application, ainsi que la propriété *MethodType* afin d'indiquer l'en-tête de méthode des messages de requête.
7. Choisissez Projet | Ajouter au projet et ajoutez à votre application serveur Web les unités créées lors des étapes 1 à 4.
8. Si vous voulez que votre application puisse fonctionner avec des clients qui n'ont pas été conçus avec Delphi, publiez un document WSDL définissant vos interfaces et la manière de les appeler. Pour davantage d'informations sur la génération d'un document WSDL décrivant votre application service Web, voir "Génération de documents WSDL pour une application service Web" à la page 31-8.

Définition des interfaces invocables

Pour créer une interface invocable, il vous suffit de compiler une interface avec l'option de compilation {$M+}. Le descendant d'une interface invocable est également invocable. Par contre, si une interface invocable dérive d'une autre interface qui ne l'est pas, les clients de votre serveur service Web ne peuvent appeler que les méthodes définies par l'interface invocable ou par ses descendants. Les méthodes héritées d'un ancêtre non-invocable ne sont pas compilées avec des informations de type et de ce fait ne peuvent être appelées par les clients.

Delphi définit une interface invocable de base, *IInvokable*, qui peut servir de base à toute interface exposée aux clients par un serveur service Web. *IInvokable* est identique à l'interface de base (*IInterface*) si ce n'est qu'elle est compilée en utilisant l'option de compilation {$M+} afin qu'elle, et tout ses descendants, soient compilés avec des informations de type à l'exécution.

Par exemple, le code suivant définit une interface invocable contenant deux méthodes pour coder et décoder des valeurs numériques :

```
IEncodeDecode = interface(IInvokable)
  ['{C527B88F-3F8E-1134-80e0-01A04F57B270}']
    function EncodeValue(Value: Integer): Double; stdcall;
    function DecodeValue(Value: Double): Integer; stdcall;
end;
```

Pour qu'une application service Web puisse utiliser cette interface invocable, elle doit être recensée dans le registre d'invocation. Sur le serveur, l'entrée du registre d'invocation permet aux composants invocateur (*THTTPSOAPPascalInvoker*) d'identifier la classe d'implémentation à utiliser pour l'exécution des appels de l'interface. Dans les applications client, une entrée du registre d'invocation permet aux composants de rechercher les informations identifiant l'interface invocable et la manière de l'appeler.

Dans la section initialisation de l'unité définissant l'interface, ajoutez du code pour recenser l'interface dans le registre d'invocation. Pour accéder au registre d'invocation, ajoutez l'unité InvokeRegistry à la clause uses de votre unité. L'unité InvokeRegistry déclare une variable globale, *InvRegistry*, qui stocke en mémoire un catalogue de toutes les interfaces invocables recensées, de leurs classes d'implémentation et des fabricants qui créent des instances de ces classes d'implémentation.

Une fois terminée, l'unité qui définit l'interface doit ressembler à ce qui suit :

```
unit EncodeDecode;

interface
type

IEncodeDecode = interface(IInvokable)
  ['{C527B88F-3F8E-1134-80e0-01A04F57B270}']
    function EncodeValue(Value: Integer): Double; stdcall;
    function DecodeValue(Value: Double): Integer; stdcall;
end;

implementation
```

```
uses  InvokeRegistry;

initialization
InvRegistry.RegisterInterface(TypeInfo(IEncodeDecode));
end.
```

Les interfaces de services Web doivent avoir un domaine d'appellation pour pouvoir s'identifier parmi toutes les interfaces possibles de tous les services Web, donc lors du recensement d'une interface dans le registre d'invocation, un domaine d'appellation est généré automatiquement pour l'interface. Le domaine d'appellation par défaut est créé à partir d'une chaîne qui identifie l'application de manière unique (la variable *AppNamespacePrefix*), le nom de l'interface et le nom de l'unité dans laquelle elle est définie.

Astuce Il est judicieux d'utiliser, pour définir les interfaces invocables, une unité distincte de celle utilisée pour les classes d'implémentation des interfaces. Cette unité peut ainsi être incluse à la fois dans l'application serveur et dans l'application client. Comme le domaine d'appellation généré contient le nom de l'unité dans laquelle l'interface est définie, le partage de la même unité dans les applications serveur et client permet d'utiliser automatiquement le même domaine d'appellation.

Utilisation de types complexes dans des interfaces invocables

Le composant invocateur (*THTTPSOAPPascalInvoker*) sait automatiquement comment transmettre les types scalaires aux interfaces invocables. Il est également capable de gérer des tableaux dynamiques à partir du moment ou ils sont recensés dans le registre des classes distantes (voir ci-dessous). Cependant, vous devez fournir une gestion supplémentaire si vous voulez transmettre des données ayant un type plus complexe : tableaux statiques, interfaces, enregistrements, ensembles ou classes. Cette gestion est assurée par une classe qui contient des informations de type à l'exécution qu'utilise l'invocateur pour effectuer la conversion entre données du flux SOAP et valeurs typées.

Utilisez *TRemotable* comme classe de base pour définir une classe représentant un type de données complexe d'une interface invocable. Si par exemple, il vous faut transmettre comme paramètre un enregistrement, il faut définir un descendant de *TRemotable* dans lequel chaque membre de l'enregistrement est une propriété publiée de la nouvelle classe.

Si la valeur de votre descendant de *TRemotable* représente un type scalaire d'un document WSDL qui ne correspond pas à un type scalaire Pascal Objet, utilisez à la place *TRemotableXS* comme classe de base. *TRemotableXS* est un descendant de *TRemotable* qui introduit deux nouvelles méthodes pour assurer la conversion entre votre nouvelle classe et sa représentation sous forme de chaîne. Fournissez ces méthodes en surchargeant les méthodes *XSToNative* et *NativeToXS*.

Dans la section initialisation de l'unité qui définit le descendant de *TRemotable*, vous devez recenser la classe dans le registre des classe distantes. Pour accéder au registre des classes distantes, vous devez inclure l'unité InvokeRegistry à la clause uses. Cette unité déclare une variable globale, *RemClassRegistry*, qui stocke un catalogue de toutes les classes distantes recensées et indique pour chacune si leurs valeurs sont transmises sous forme de chaînes. Par exemple, la ligne

suivante provient de l'unité XSBuiltIns. Elle recense *TXSDateTime*, un descendant de *TRemotable* qui représente des valeurs *TDateTime* :

```
RemClassRegistry.RegisterXSClass(TXSDateTime, XMLSchemaNameSpace, 'dateTime', True);
```

Le premier paramètre est le nom du descendant de *TRemotable*. Le deuxième est un identificateur de ressource uniforme (URI) qui identifie de manière unique le domaine d'appellation de la nouvelle classe. Si vous spécifiez une chaîne vide, le registre peut générer l'URI à votre place. Le troisième paramètre est le nom du type de données que la classe représente. Si vous spécifiez une chaîne vide, le registre utilise simplement le nom de la classe. Le dernier paramètre indique si la valeur de l'instance de classe peut être transmise sous forme de chaîne (c'est-à-dire si vous avez implémenté les méthodes *XSToNative* et *NativeToXS*).

Astuce Il est judicieux d'implémenter et de recenser les descendants de *TRemotable* dans une unité distincte du reste de votre application serveur, y compris des unités qui déclarent et recensent les interfaces invocables. Vous pouvez, ainsi, utiliser l'unité définissant votre type à la fois dans le client et dans le serveur et vous pouvez également utiliser le type dans plusieurs interfaces.

Si vous utilisez des tableaux dynamiques pour les paramètres, vous n'avez pas besoin de créer une classe distante pour les représenter, mais vous devez les recenser dans le registre des classes distantes. Si, par exemple, votre interface utilise un type comme le suivant :

```
type
  TDateTimeArray = array of TXSDateTime;
```

Vous devez ajouter le recensement suivant à la section initialisation de l'unité dans laquelle vous déclarez le tableau dynamique :

```
RemClassRegistry.RegisterXSInfo(TypeInfo(TDateTimeArray), MyNameSpace, 'DTarray', False);
```

Les paramètres sont identiques à ceux utilisés par *RegisterXSClass*, sauf pour le premier qui est un pointeur sur les informations de type du tableau dynamique à la place d'une référence à la classe.

Création et recensement de l'implémentation

Le moyen le plus simple d'écrire l'implémentation d'une interface invocable consiste à créer une classe dérivant de *TInvokableClass*. Ajoutez la déclaration de classe, y compris les interfaces invocables que vous gérez, puis appuyez sur *Ctrl+Maj+C* pour obtenir l'achèvement de classe. Les membres de l'interface apparaissent dans votre déclaration de classe et les méthodes vides sont placées dans la section implémentation de l'unité.

Par exemple, la déclaration de la classe d'implémentation de l'interface déclarée dans la section "Définition des interfaces invocables" peut ressembler à :

```
TEncodeDecode = class(TInvokableClass, IEncodeDecode)
protected
  function EncodeValue(Value: Integer): Double; stdcall;
  function DecodeValue(Value: Double): Integer; stdcall;
end;
```

Dans la section implémentation de l'unité déclarant cette classe, remplissez les méthodes *EncodeValue* et *DecodeValue*.

Une fois la classe d'implémentation créée, vous devez la recenser dans le registre d'invocation. Ce registre utilise cela pour identifier la classe qui implémente une interface recensée et pour la rendre utilisable par le composant invocateur quand il a besoin d'appeler l'interface. Pour recenser la classe d'implémentation, ajoutez un appel à la méthode *RegisterInvokableClass* de la variable globale *InvRegistry* dans la section initialisation de votre unité d'implémentation :

```
InvRegistry.RegisterInvokableClass(TEncodeDecode);
```

Vous pouvez également créer des classes d'implémentation qui ne dérivent pas de *TInvokableClass*. Cependant, vous devez dans ce cas fournir une procédure de fabrication qu'utilise le registre d'invocation pour créer des instances de votre classe.

La procédure de fabrication doit être de type *TCreateInstanceProc*. Elle renvoie une instance de votre classe d'implémentation. Si la procédure crée une nouvelle instance, l'objet doit se libérer lui-même quand le compteur de référence à ses interfaces devient nul, en effet le registre d'invocation ne libère pas explicitement les instances d'objet. Sinon, autre solution, la procédure de fabrication peut renvoyer une référence à une instance globale partagée par tous les appelants. Le code suivant illustre cette dernière approche :

```
procedure CreateEncodeDecode(out obj: TObject);
begin
  if FEncodeDecode = nil then
  begin
    FEncodeDecode := TEncodeDecode.Create;
    {enregistre une référence à l'interface afin que l'instance globale ne se libère pas
elle-même }
    FEncodeDecodeInterface := FEncodeDecode as IEncodeDecode;
  end;
  obj := FEncodeDecode; { renvoie l'instance globale }
end;
```

Quand vous utilisez une procédure de fabrication, fournissez la procédure de fabrication comme second paramètre de la méthode *RegisterInvokableClass* :

```
InvRegistry.RegisterInvokableClass(TEncodeDecode, CreateEncodeDecode);
```

Création de classes d'exception personnalisées pour les services Web

Si lorsqu'elle tente d'exécuter une requête SOAP, votre application service Web déclenche une exception, les informations sur cette exception sont automatiquement codées dans un paquet d'erreur SOAP qui est renvoyé à la place du résultat de l'appel de méthode. L'application client déclenche alors l'exception.

Par défaut, l'application client déclenche juste une exception générique (*Exception*)avec le message d'erreur contenu dans le paquet d'erreur SOAP. Vous pouvez cependant transmettre des informations complémentaires sur l'exception en utilisant une classe d'exception dérivant de *ERemotableException*. La valeur de toutes les propriétés publiées ajoutées à votre classe d'exception sont placées

dans le paquet d'erreur SOAP afin que le client puisse déclencher une exception équivalente.

Pour utiliser un descendant de *ERemotableException*, vous devez le recenser dans le registre des classes distantes. C'est-à-dire que vous devez, dans l'unité qui définit votre descendant *de ERemotableException,* ajouter l'unité InvokeRegistry à la clause uses et appeler la méthode *RegisterXSClass* de la variable globale *RemClassRegistry*.

Si le client utilise la même unité que celle qui définit et recense votre descendant de *ERemotableException*, lorsqu'il reçoit le paquet d'erreur SOAP, il déclenche automatiquement une instance de la classe d'exception appropriée en initialisant toutes ses propriétés avec les valeurs contenues dans le paquet d'erreur SOAP.

Génération de documents WSDL pour une application service Web

Si vous incluez les mêmes unités qui définissent et recensent vos interfaces invocables, les classes qui représentent les informations des types complexes et vos exceptions distantes dans une application client Delphi, celle-ci peut générer des appels utilisant votre service Web. Il vous suffit juste de spécifier l'URL où est installée l'application service Web.

Cependant, vous pouvez choisir de proposer votre service Web à une gamme plus large de clients. Vous pouvez, par exemple, avoir des clients qui ne sont pas écrits avec Delphi. Si vous déployez plusieurs versions de votre application serveur, vous pouvez préférer ne pas utiliser une seule URL codée en dur pour le serveur mais laisser plutôt le client rechercher dynamiquement l'emplacement du serveur. Dans ces cas, vous pouvez publier un document WSDL décrivant les types et interfaces de votre service Web, ainsi que des informations sur la manière de les appeler.

Pour publier un document WSDL décrivant votre service Web, ajoutez simplement un composant *TWSDLHTMLPublish* à votre module Web. *TWSDLHTMLPublish* est un composant d'auto-répartition, ce qui signifie qu'il répond automatiquement aux messages entrant qui demandent une liste des documents WSDL pour votre service Web. Utilisez la propriété *WebDispatch* pour spécifier le chemin d'information que les clients URL doivent utiliser pour accéder à la liste des documents WSDL. Les navigateurs Web peuvent alors demander une liste de documents WSDL en spécifiant une URL qui est constituée de l'emplacement de l'application serveur suivie du chemin spécifié par la propriété WebDispatch. Cette URL a la forme suivante :

```
http://www.myco.com/MyService.dll/WSDL
```

Astuce Si vous voulez utiliser un fichier WSDL physique à la place, vous pouvez afficher le document WSDL dans votre navigateur Web puis l'enregistrer afin de générer le fichier document WSDL.

Il n'est pas nécessaire de publier le document WSDL depuis la même application que celle implémentant votre service Web. Pour créer une application qui publie simplement le document WSDL, omettez les unités contenant les objets d'implémentation et n'incluez que les unités définissant et recensant les

interfaces invocables, les classes distantes qui représentent des types complexes et toutes les exceptions distantes.

Par défaut, quand vous publiez un document WSDL, il indique que les services sont disponibles à la même adresse que celle publiant le document WSDL (mais avec un chemin différent). Si vous déployez plusieurs versions de votre application service Web ou si vous déployez votre document WSDL depuis une application différente de celle qui implémente le service Web, vous devez modifier le document WSDL afin qu'il contienne des informations à jour sur l'emplacement du service Web.

Pour changer l'URL, utilisez l'administrateur WSDL. Vous devez tout d'abord activer l'administrateur. Pour ce faire, initialisez la propriété *AdminEnabled* du composant *TWSDLHTMLPublish* à *True*. Après, lorsque vous utilisez votre navigateur pour afficher la liste des documents WSDL, elle contient un bouton permettant également de les administrer. Utilisez l'administrateur WSDL pour spécifier les emplacements (URL) où vous avez déployé votre application service Web.

Conception de clients pour les services Web

Delphi permet la conception de clients appelant des services Web utilisant une liaison de type SOAP. Ces services Web peuvent être fournis par un serveur créé avec Delphi, ou par tout autre serveur définissant ses services Web dans un document WSDL.

Si le serveur n'a pas été écrit avec Delphi, vous devez tout d'abord importer le document WSDL décrivant le serveur. Ce processus est décrit plus bas. Si le serveur a été écrit en utilisant Delphi, il n'est pas nécessaire d'utiliser un document WSDL : il suffit d'ajouter à votre projet les unités qui définissent les interfaces invocables que vous souhaitez utiliser, ainsi que les unités définissant les classes distantes représentant des types complexes ou des exceptions distantes que le service Web peut déclencher.

Remarque Pour davantage d'informations sur la création des unités définissant une interface invocable dans un serveur de services Web Delphi, voir "Définition des interfaces invocables" à la page 31-4. Pour davantage d'informations sur la création d'une unité définissant une classe distante pour un type complexe, voir "Utilisation de types complexes dans des interfaces invocables" à la page 31-5. Pour davantage d'informations sur la création d'une unité définissant une exception distante, voir "Création de classes d'exception personnalisées pour les services Web" à la page 31-7.

Importation de documents WSDL

Avant de pouvoir utiliser un service Web qui n'a pas été écrit avec Delphi, vous devez importer un document WSDL (ou un fichier schéma XML) qui définit le service. L'importateur de service Web crée une unité qui définit et recense les interfaces et les types dont vous avez besoin.

Pour utiliser l'importateur de service Web, choisissez Fichier | Nouveau | Autre et, dans la page Services Web, double-cliquez sur l'icône de l'importateur de services Web. Dans la boîte de dialogue qui est affichée, spécifiez le nom de fichier du document WSDL (ou d'un fichier schéma XML) ou spécifiez l'URL où est publiée ce document. Quand vous cliquez sur le bouton Générer, l'importateur crée de nouvelles unités qui définissent et recensent les interfaces invocables pour les opérations définies dans le document, ainsi que pour les classes distantes correspondant aux types définis par le document.

Si le document WSDL, ou le fichier schéma XML, utilise des identificateurs qui sont également des mots réservés Pascal Objet, l'importateur adapte automatiquement leur nom afin que le code généré soit compilable. Quand des types complexes sont déclarés en ligne, l'importateur ajoute du code pour définir et recenser les classes distantes correspondantes dans la même unité que l'interface invocable qui les utilise. Sinon, les types sont définis et recensés dans une unité distincte.

Appel des interfaces invocables

Pour appeler une interface invocable, l'application client doit inclure toutes les unités définissant les interfaces invocables et toutes les classes distantes implémentant des types complexes. Si le serveur est conçu avec Delphi, ce sont les mêmes unités que celles utilisées par l'application serveur pour définir et recenser ces interfaces et ces classes. Il est préférable d'utiliser les mêmes unités, car lorsque vous recensez une interface invocable ou une classe distante, un identificateur de ressource uniforme (URI) lui est attribué pour l'identifier de manière unique. Cette URI est construite à partir du nom de l'interface (ou de la classe) et du nom de l'unité dans laquelle elle est définie. Si le client et le serveur ne recensent pas l'interface (ou la classe) en utilisant la même URI, ils ne peuvent communiquer. Si vous n'utilisez pas la même unité, le code qui recense l'interface et la classe d'implémentation doit spécifier explicitement une URI de domaine d'appellation pour s'assurer que le client et le serveur utilisent le même domaine d'appellation.

Si le serveur n'est pas conçu avec Delphi ou si vous ne voulez pas utiliser pour le client la même unité que celle utilisée par le serveur, ces unités peuvent être créées par l'importateur de services Web.

Quand l'application client dispose d'une déclaration de l'interface invocable, créez une instance de *THTTPRio* pour l'interface voulue :

```
X := THTTPRio.Create(nil);
```

Fournissez ensuite à l'objet *THTTPRio* les informations permettant d'identifier l'interface du serveur et de localiser le serveur. Il y a deux manières de spécifier ces informations :

- Si le serveur a été conçu avec Delphi, l'identification de l'interface du serveur se fait automatiquement en se basant sur l'URI générée lors du recensement de l'interface. Il suffit donc d'initialiser la propriété *URL* afin d'indiquer l'emplacement du serveur. La partie chemin d'accès de cette URL doit

correspondre au chemin d'accès spécifié au composant répartiteur du module Web du serveur :

```
X.URL := 'http://www.myco.com/MyService.dll/SOAP/';
```

- Si le serveur n'a pas été conçu en utilisant Delphi, *THTTPRio* doit rechercher dans un document WSDL l'URI de l'interface, information qui doit être placée dans l'en-tête Soap Action, ainsi que l'emplacement du serveur. Pour ce faire, utilisez les propriétés *Service* et *Port* de *WSDLLocation* :

```
X.WSDLLocation := 'Cryptography.wsdl';
X.Service := 'Cryptography';
X.Port := 'SoapEncodeDecode';
```

Vous pouvez ensuite utiliser l'opérateur **as** pour transtyper l'instance de *THTTPRio* en l'interface invocable. En procédant ainsi, cela crée dynamiquement en mémoire une vtable pour l'interface associée, ce qui permet d'effectuer les appels de l'interface :

```
InterfaceVariable := X as IEncodeDecode;
Code := InterfaceVariable.EncodeValue(5);
```

THTTPRio dépend du registre d'invocation pour obtenir des informations sur l'interface invocable. Si l'application client n'a pas de registre d'invocation ou si l'interface invocable n'est pas recensée, *THTTPRio* ne peut pas construire la vtable en mémoire.

Chapitre

32

Utilisation des sockets

Les composants socket vous permettent de créer une application pouvant communiquer avec d'autres systèmes par TCP/IP et ses protocoles associés. A l'aide des sockets, vous pouvez lire et écrire sur des connexions à d'autres machines sans vous soucier des détails concernant le logiciel réseau sous-jacent. Les sockets offrent des connexions basées sur le protocole TCP/IP et sont assez génériques pour fonctionner avec des protocoles tels que User Datagram Protocol (UDP), Xerox Network System (XNS), DECnet de Digital ou la famille IPX/SPX de Novell.

L'utilisation des sockets vous permet d'écrire des applications serveur ou client qui lisent et écrivent sur des systèmes distants. Une application serveur ou client est en général dédiée à un service unique tel que HTTP (Hypertext Transfer Protocol) ou FTP (File Transfer Protocol). En utilisant les sockets serveur, une application offrant l'un de ces services peut se lier aux applications client qui souhaitent utiliser ce service. Les sockets client permettent à une application utilisant l'un de ces services de se lier à des applications serveur offrant ce service.

Implémentation des services

Les sockets offrent l'un des composants dont vous avez besoin pour créer des applications serveur ou client. Pour beaucoup de services, tels que HTTP ou FTP, des serveurs développés par diverses sociétés sont disponibles. Certains sont même fournis en standard avec le système d'exploitation, ce qui vous évite d'en créer un vous-même. Cependant, si vous désirez affiner la façon dont le service est implémenté (pour obtenir, par exemple, une meilleure intégration de votre application et de la communication réseau) ou si aucun serveur n'est disponible pour le service précis dont vous avez besoin, vous devrez créer votre propre application serveur ou client. Par exemple, lorsque vous manipulez des ensembles de données distribués, vous pouvez créer une couche pour communiquer avec des bases de données sur les systèmes distants.

Description des protocoles de services

Avant de créer un serveur ou un client réseau, vous devez comprendre le service que votre application offrira ou utilisera. La majorité des services ont des protocoles standard que votre application doit supporter. Si vous créez une application réseau pour un service standard tel que HTTP, FTP, Finger ou Time, vous devez comprendre les protocoles utilisés pour communiquer avec les systèmes distants. Consultez la documentation se rapportant au service que vous comptez offrir ou utiliser.

Si vous offrez un nouveau service pour une application qui communique avec des systèmes distants, la première étape consiste à concevoir le protocole de communication pour les serveurs et les clients de ce service. Quels messages sont envoyés ? Comment ces messages sont-ils structurés ? Comment les informations sont-elles codées ?

Communication avec les applications

Souvent, votre application serveur ou client offre une couche entre le logiciel réseau et l'application qui utilise le service. Par exemple, un serveur HTTP est placé entre Internet et l'application serveur Web qui fournit le contenu et gère les messages de demandes HTTP.

Les sockets sont l'interface entre votre application serveur ou client et le logiciel réseau. Vous devez fournir l'interface entre votre application et les clients qui l'utilisent. Vous pouvez copier l'API d'un autre serveur standard (comme Apache) ou vous pouvez concevoir et publier votre propre API.

Services et ports

La plupart des services standard sont associés, par convention, à des numéros de ports précis. Nous étudierons plus tard ces numéros de ports. Pour l'instant, considérez le numéro de port comme un code numérique pour ce service.

Si vous implémentez un service standard, les objets socket Linux fournissent des méthodes de recherche de numéro de port pour le service. Si vous offrez un service nouveau, vous pouvez spécifier son numéro de port dans un fichier /etc/services. Voir votre documentation Linux, pour davantage d'informations sur la configuration d'un fichier de services.

Types de connexions par socket

Les connexions par socket peuvent être scindées en trois groupes principaux indiquant la façon dont la connexion a été ouverte et le type de connexion :

- Connexions client
- Connexions d'écoute
- Connexions serveur

Lorsque la connexion au socket client est effective, la connexion serveur est identique à une connexion client. Les deux extrémités ont les mêmes possibilités et reçoivent des événements de même type. Seule la connexion d'écoute est fondamentalement différente, car elle ne comporte qu'une extrémité.

Connexions client

Les connexions client connectent un socket client sur le système local à un socket serveur sur un système distant. Les connexions client sont lancées par le socket client. En premier lieu, le socket client doit décrire le socket serveur auquel il souhaite se connecter. Le socket client recherche ensuite le socket serveur et, lorsqu'il l'a trouvé, demande une connexion. Le socket serveur peut ne pas établir immédiatement la connexion. Les sockets serveur gèrent une file d'attente des demandes de clients et établissent la connexion lorsqu'ils le peuvent. Lorsque le socket serveur accepte la connexion du client, il envoie au socket client une description complète du socket serveur auquel il se connecte et la connexion est finalisée par le client.

Connexions d'écoute

Les sockets serveur ne localisent pas les clients : ils génèrent des "demi-connexions" passives qui restent à l'écoute des requêtes des clients. Les sockets serveur associent une file d'attente à leurs connexions d'écoute ; la file d'attente enregistre les requêtes de connexion lorsqu'elles lui parviennent. Lorsque le socket serveur accepte une demande de connexion client, il forme un nouveau socket pour se connecter au client pour que la connexion d'écoute reste ouverte afin d'accepter d'autres requêtes de clients.

Connexions serveur

Les connexions serveur sont formées par des sockets serveur lorsque le socket d'écoute accepte une requête du client. La description du socket serveur ayant effectué la connexion au client est envoyée au client lorsque le serveur accepte la connexion. La connexion est établie lorsque le socket client reçoit cette description et effectue véritablement la connexion.

Description des sockets

Les sockets permettent à votre application de communiquer avec des systèmes distants par le biais d'un réseau. Chaque socket peut être considéré comme un point de terminaison dans une connexion réseau. Il possède une adresse qui spécifie :

- le système sur lequel il s'exécute ;
- les types d'interfaces qu'il comprend ;
- le port qu'il utilise pour la connexion.

Une description complète d'une connexion par socket inclut les adresses du socket à chaque extrémité de la connexion. Vous pouvez décrire l'adresse de chaque extrémité du socket en fournissant l'hôte ou l'adresse IP et le numéro de port.

Avant de faire une connexion par socket, vous devez décrire complètement les sockets formant ses extrémités. Certaines informations sont disponibles sur le système exécutant votre application. Par exemple, il n'est pas nécessaire de décrire l'adresse IP locale d'un socket client car elle est dans le système d'exploitation.

Les informations à fournir dépendent du type de socket implémenté. Les sockets client doivent décrire le serveur auquel ils souhaitent se connecter. Les sockets serveur d'écoute doivent décrire le port représentant le service qu'ils offrent.

Description des hôtes

L'hôte est le système qui exécute l'application contenant le socket. Vous pouvez décrire les hôtes à un socket en fournissant son adresse IP, qui consiste en une chaîne de quatre valeurs numériques au format Internet standard, par exemple :

```
123.197.1.2
```

Un système peut gérer plusieurs adresses IP.

Les adresses IP sont difficiles à mémoriser. L'alternative consiste à utiliser le nom de l'hôte. Les noms d'hôtes sont des alias d'adresses IP exprimées au format URL (Uniform Resource Locator). Une chaîne d'URL comporte un nom de domaine et un service, par exemple :

```
http://www.ASite.com
```

La majorité des intranets fournit des noms d'hôtes pour les adresses IP des systèmes sur Internet. Pour déterminer le nom d'hôte éventuellement associé à une adresse IP, exécutez la commande suivante depuis la ligne de commande :

```
nslookup IPADDRESS
```

Où *IPADDRESS* est l'adresse IP qui vous intéresse. Si votre adresse IP locale n'a pas de nom d'hôte et que vous en souhaitez une, contactez votre administrateur réseau.

Les sockets serveur n'ont pas besoin de spécifier d'hôtes. L'adresse IP locale peut être obtenue auprès du système. Si le système local gère plusieurs adresses IP, les sockets écoutent les requêtes client sur toutes ces adresses en même temps. Lorsqu'un socket serveur accepte une connexion, le socket client indique l'adresse IP du système distant.

Les sockets client doivent spécifier les hôtes distants en fournissant leur nom ou leur adresse IP.

Choix entre le nom de l'hôte et son adresse IP

La majorité des applications utilise un nom d'hôte pour désigner un système. Les noms d'hôtes sont en effet plus faciles à mémoriser. De plus, les serveurs peuvent changer l'adresse IP ou le système associé à un nom d'hôte précis. Le fait d'utiliser les noms d'hôtes permet au socket client de trouver le site abstrait représenté par le nom d'hôte même s'il a changé d'adresse IP.

Si le nom de l'hôte vous est inconnu, le socket client doit spécifier le système serveur par son adresse IP. Ce processus donne des recherches plus rapides car, lorsque vous spécifiez un nom d'hôte, le socket doit rechercher l'adresse IP associée à ce nom pour localiser le système serveur.

Utilisation des ports

Même si une adresse IP contient assez d'informations pour trouver le système à l'autre bout de la connexion socket, vous devez également indiquer un numéro de port sur ce système. Sans les numéros de port, un système ne pourrait former qu'une connexion à la fois. Les numéros de port sont des identificateurs uniques permettant à un ordinateur d'accepter plusieurs connexions simultanées en attribuant à chaque connexion un numéro de port distinct.

Nous avons décrit précédemment les numéros de port comme des codes numériques pour les services implémentés par les applications réseau. Il s'agit d'une convention permettant aux connexions serveur d'écoute de se libérer sur un numéro de port fixe pour qu'elles puissent être trouvées par les sockets client. Les sockets serveur écoutent sur le numéro de port associé au service qu'ils offrent. Lorsqu'ils acceptent une connexion à un socket client, ils créent une connexion socket distincte utilisant un autre numéro de port attribué arbitrairement. De cette façon, la connexion d'écoute peut rester vigilante sur le numéro de port associé au service.

Les sockets client utilisent un numéro de port local arbitraire car ils n'ont pas besoin d'être détectés par les autres sockets. Ils spécifient le numéro de port du socket serveur auquel ils désirent se connecter pour pouvoir trouver l'application serveur. Souvent, ce numéro de port est spécifié indirectement en nommant le service souhaité.

Utilisation des composants socket

La page Internet de la palette des composants propose trois composants socket permettant à votre application réseau de constituer des connexions avec d'autres machines et de lire ou d'écrire des informations à travers cette connexion. Ce sont :

- *TcpServer*
- *TcpClient*
- *UdpSocket*

Des objets socket sont associés à chacun de ces composants socket, et représentent l'extrémité d'une connexion socket existante. Les composants socket utilisent les objets socket pour encapsuler les appels du socket serveur pour que votre application n'ait pas à connaître les détails de l'établissement de la connexion ou de la gestion des messages du socket

Si vous souhaitez personnaliser les détails des connexions qu'un composant socket effectue pour vous, vous pouvez utiliser les propriétés, les événements et les méthodes des objets socket.

Obtenir des informations sur la connexion

Une fois la connexion établie avec le socket client ou serveur, vous pouvez utiliser l'objet socket client ou serveur associé à votre composant socket pour obtenir des informations sur la connexion. Utilisez les propriétés *LocalHost* et *LocalPort* pour déterminer l'adresse et le numéro de port utilisés par le socket client ou serveur local ; utilisez les propriétés *RemoteHost* et *RemotePort* pour déterminer l'adresse et le numéro de port utilisés par le socket client ou serveur distant. Utilisez la méthode *GetSocketAddr* pour construire une adresse de socket correcte à partir du nom d'hôte et du numéro de port. Vous pouvez utiliser la méthode *LookupPort* pour rechercher le numéro de port. Utilisez la méthode *LookupProtocol* pour rechercher le numéro de protocole. Utilisez la méthode *LookupHostName* pour déterminer le nom d'hôte à partir de l'adresse IP de la machine hôte.

Pour visualiser le trafic réseau entrant ou sortant par le socket, utilisez les propriétés *BytesSent* et *BytesReceived*.

Utilisation de sockets client

Ajoutez un composant socket client (*TTcpClient* ou *UdpSocket*) à votre fiche ou à votre module de données pour transformer votre application en client TCP/IP ou UDP. Les sockets client vous permettent de spécifier le socket serveur auquel vous souhaitez vous connecter et le service que vous attendez de ce serveur. Lorsque vous avez décrit la connexion voulue, vous pouvez utiliser le composant socket client pour établir la connexion au serveur.

Chaque composant socket client utilise un seul objet socket client pour représenter l'extrémité client d'une connexion.

Désignation du serveur souhaité

Les composants socket client ont de nombreuses propriétés qui vous permettent de spécifier le système serveur et le port auxquels vous souhaitez vous connecter. Utilisez la propriété *RemoteHost* pour spécifier le serveur hôte distant par son nom d'hôte ou son adresse IP.

En plus du système serveur, vous devez spécifier le port du système serveur sur lequel votre socket client se connectera. Vous pouvez utiliser la propriété *RemotePort* pour spécifier ce numéro de port directement ou indirectement en nommant le service de destination.

Formation de la connexion

Lorsque les propriétés de votre composant socket client décrivent le serveur auquel vous souhaitez vous connecter, vous pouvez former la connexion à l'exécution en appelant la méthode *Open*. Si vous souhaitez que votre application forme automatiquement la connexion à son ouverture, mettez la propriété *Active* à *True* lors de la conception, à l'aide de l'inspecteur d'objets.

Obtention d'informations sur la connexion

Lorsque vous avez ouvert une connexion d'écoute avec votre socket serveur, vous pouvez utiliser l'objet socket serveur associé à votre composant socket serveur pour obtenir des informations sur la connexion. Utilisez les propriétés *LocalHost* et *LocalPort* pour déterminer l'adresse et le numéro de port utilisés par les sockets client et serveur pour former les extrémités de la connexion. Utilisez la propriété *Handle* pour obtenir un handle sur la connexion socket à utiliser lorsque vous faites des appels de socket.

Fermeture de la connexion

Lorsque vous avez fini de communiquer avec une application serveur sur la connexion socket, vous pouvez fermer la connexion en appelant la méthode *Close*. La connexion peut également être fermée depuis le serveur. Si c'est le cas, vous en êtes informé par un événement *OnDisconnect*.

Utilisation de sockets serveur

Ajoutez un composant socket serveur (*TTcpServer* ou *UdpSocket*) à votre fiche ou à votre module de données pour transformer votre application en serveur IP. Les sockets serveur vous permettent de spécifier le service que vous offrez ou le port que vous utilisez pour écouter les requêtes client. Vous pouvez utiliser le composant socket serveur pour écouter et accepter les requêtes de connexion client.

Chaque composant socket serveur utilise un seul objet socket serveur pour représenter l'extrémité serveur d'une connexion d'écoute. Il utilise également un objet socket serveur comme extrémité serveur de chaque connexion active avec un socket client qui a été acceptée par le serveur.

Désignation du port

Pour que votre socket serveur puisse écouter les requêtes de connexion client, vous devez spécifier un port d'écoute. Vous pouvez spécifier ce port à l'aide de la propriété *LocalPort*. Si votre application serveur offre un service standard associé par convention à un numéro de port précis, vous pouvez aussi spécifier le nom du service à l'aide de la propriété *LocalPort*. Il est recommandé d'utiliser plutôt le nom du service car il est facile de faire des fautes de saisie en entrant le numéro de port.

Ecoute des requêtes client

Lorsque vous avez défini le numéro de port de votre composant socket serveur, vous pouvez former une connexion d'écoute à l'exécution en appelant la méthode *Open*. Si vous souhaitez que cette connexion soit formée automatiquement au lancement de l'application, mettez la propriété *Active* à *True* lors de la conception, à l'aide de l'inspecteur d'objets.

Connexion aux clients

Un composant socket serveur d'écoute accepte automatiquement les requêtes de connexion des clients à leur réception. Vous en recevez à chaque fois automatiquement la notification dans un événement *OnAccept*.

Fermeture des connexions serveur

Lorsque vous souhaitez fermer la connexion d'écoute, appelez la méthode *Close* ou définissez la propriété *Active* à *False*. Ceci ferme toutes les connexions établies avec les applications client, annule les connexions en attente non encore acceptées et met fin à la connexion d'écoute, après quoi votre composant socket serveur n'accepte pas de nouvelles connexions.

Lorsque les clients TCP mettent fin à leur connexion à votre socket serveur, vous en êtes informé par un événement *OnDisconnect*.

Réponse aux événements socket

Quand vous écrivez des applications utilisant les sockets, vous pouvez écrire ou lire dans le socket n'importe où dans le programme. Vous pouvez écrire en utilisant les méthodes *SendBuf*, *SendStream* ou *Sendln* lorsque le socket a été ouvert. Vous pouvez lire depuis le socket en utilisant les méthodes *ReceiveBuf*et *Receiveln*. Les événements *OnSend*et *RecvBuf* sont déclenchés chaque fois que quelque chose a été écrit ou lu dans le socket. Ils peuvent être utilisés pour le filtrage. Chaque fois que vous lisez ou écrivez, un événement de lecture ou d'écriture est déclenché.

Les sockets client et les sockets serveur génèrent tous deux des événements d'erreurs lorsqu'ils reçoivent des messages d'erreur émis par la connexion.

Les composants socket reçoivent également deux événements au cours de l'ouverture et de la terminaison de la connexion. Si votre application doit influer sur l'ouverture du socket, vous devez utiliser les méthodes *SendBuf* et *ReceiveBuf* pour répondre à ces événements client ou événements serveur.

Evénements d'erreurs

Les sockets client et serveur génèrent un événement *OnError* lorsqu'ils reçoivent un message d'erreur émis par la connexion. Vous pouvez écrire un gestionnaire d'événement *OnError* pour répondre à ces messages d'erreur. Le gestionnaire d'événement reçoit des informations sur :

- l'objet socket ayant reçu la notification d'erreur ;
- ce que le socket tentait de faire lorsque l'erreur s'est produite ;
- le code d'erreur fourni par le message d'erreur.

Vous pouvez répondre à l'erreur dans le gestionnaire d'événement et mettre le code d'erreur à 0 pour empêcher le socket de déclencher une exception.

Evénements client

Lorsqu'un socket client ouvre une connexion, les événements suivants se produisent :

- Le socket est paramétré et initialisé pour la notification d'événements.
- Un événement *OnCreateHandle* se produit une fois que le serveur et le serveur socket sont créés. A ce moment, l'objet socket accessible par la propriété *Handle* peut fournir des informations sur le serveur ou client socket qui sera à l'autre extrémité de la connexion. Ceci est la première possibilité d'obtenir le numéro de port utilisé pour la connexion, qui peut différer du numéro de port des sockets d'écoute qui ont accepté la connexion.
- La demande de connexion est acceptée par le serveur et gérée par le socket client.
- Un événement de notification *OnConnect* se produit après l'établissement de la connexion.

Evénements serveur

Les composants socket serveur forment deux types de connexions : les connexions d'écoute et les connexions aux applications client. Le socket serveur reçoit des événements lors de la formation de ces deux types de connexions.

Evénements d'écoute

Juste avant que la connexion d'écoute soit formée, l'événement *OnListening* se produit. Vous pouvez utiliser la propriété *Handle* pour modifier le socket avant qu'il ne soit ouvert pour l'écoute. Par exemple, si vous souhaitez restreindre les adresses IP que le serveur utilise pour les écoutes, vous pouvez le faire dans un gestionnaire d'événement *OnListening*.

Evénements de connexions client

Lorsqu'un socket serveur accepte une demande de connexion client, les événements suivants se produisent :

- Un événement *OnAccept* se produit, et transmet le nouvel objet *TTcpClient* au gestionnaire d'événement. Ceci est le premier point lorsque vous pouvez utiliser les propriétés de *TTcpClient* pour obtenir des informations sur l'extrémité serveur de la connexion à un client.
- Si *BlockMode* est à *bmThreadBlocking*, un événement *OnGetThread* se produit. Si vous souhaitez fournir votre propre descendant de *TServerSocketThread*, vous pouvez en créer un dans un gestionnaire d'événement *OnCreateServerSocketThread* ; celui-ci sera utilisé à la place de *TServerSocketThread*. Si vous voulez effectuer une initialisation du thread ou des appels d'API avant que le thread ne commence à lire ou écrire via la connexion, vous devez également utiliser le gestionnaire d'événement *OnGetThread*.
- Le client termine la connexion et un événement *OnAccept* se produit. Si vous utilisez un serveur non bloquant, vous pouvez alors commencer les lectures et les écritures sur la connexion socket.

Lectures et écritures sur des connexions socket

L'une des raisons pour lesquelles vous formez des connexions socket avec d'autres machines est de pouvoir lire et écrire des informations par le biais de ces connexions. Le type d'informations que vous lisez et écrivez, ainsi que le moment auquel vous les lisez ou les écrivez, dépendent du service associé à la connexion socket.

Les lectures et les écritures sur sockets peuvent se dérouler en mode asynchrone pour ne pas bloquer l'exécution d'autre code dans votre application réseau. Ceci s'appelle une connexion non bloquante. Vous pouvez également former des connexions bloquantes, lors desquelles votre application attend la fin de la lecture ou de l'écriture avant d'exécuter les instructions suivantes.

Connexions non bloquantes

Les connexions non bloquantes lisent et écrivent de façon asynchrone, de sorte que le transfert de données ne bloque pas l'exécution de code dans votre application réseau. Pour créer une connexion non bloquante pour les sockets client et serveur, initialisez la propriété *BlockMode* à *bmNonBlocking*.

Lorsque la connexion est non bloquante, la lecture et l'écriture d'événements informent votre socket de la tentative de lecture et d'écriture d'informations par le socket de l'autre extrémité de la connexion.

Lecture et écriture d'événements

Les sockets non bloquants génèrent des événements de lecture et d'écriture lorsqu'il est nécessaire de lire ou d'écrire via la connexion. Avec les sockets client, vous pouvez répondre à ces notifications dans un gestionnaire d'événement *OnReceive* ou *OnSend*.

L'objet socket associé à la connexion socket est transmis comme paramètre aux gestionnaires d'événements en lecture et écriture. Cet objet socket propose plusieurs méthodes pour lire ou écrire via la connexion.

Pour lire via la connexion de socket, utilisez les méthodes *ReceiveBuf* ou *Receiveln*. Pour écrire via la connexion de socket, utilisez les méthodes *SendBuf*, *SendStream* ou *Sendln*.

Connexions bloquantes

Lorsque la connexion est bloquante, votre socket doit initier la lecture ou l'écriture sur la connexion, plutôt qu'attendre la notification émanant de la connexion socket. Utilisez un socket bloquant lorsque votre côté de la connexion décide du moment où doivent s'effectuer les lectures et les écritures.

Pour les sockets client, initialisez la propriété *BlockMode* à *bmBlocking* pour former une connexion bloquante. En fonction de ce dont est capable votre application client, il peut être recommandé de créer un nouveau thread d'exécution pour les lectures ou les écritures, de telle sorte que votre application puisse continuer l'exécution du code dans d'autres threads tout en attendant la fin des lectures ou des écritures sur la connexion.

Pour les sockets serveur, mettez la propriété *BlockMode* à *bmBlocking* ou à *bmThreadBlocking* pour former une connexion bloquante. Etant donné que les connexions bloquantes interrompent l'exécution de tout autre code lorsque le socket attend que des informations soient écrites ou lues sur la connexion, les composants de socket serveur génèrent toujours un nouveau thread d'exécution pour chaque connexion client lorsque *BlockMode* est à *bmThreadBlocking*. Quand *BlockMode* est à *bmBlocking*, l'exécution du programme est bloqué jusqu'à ce qu'une nouvelle connexion soit établie.

Partie

IV

Développement d'applications COM

Les chapitres de cette partie présentent les concepts nécessaires à la construction d'applications COM : contrôleurs Automation, serveurs Automation, contrôles ActiveX et applications COM+.

Remarque La prise en charge des clients COM est disponible dans toutes les éditions de Delphi. Néanmoins, vous avez besoin de l'édition Professionnelle ou Entreprise pour créer des serveurs.

Chapitre

33

Présentation des technologies COM

Delphi fournit des experts et des classes qui facilitent l'implémentation d'applications basées sur COM (Component Object Model) de Microsoft. Grâce à ces experts, vous pouvez créer des classes et des composants basés sur COM que vous utiliserez dans des applications, ou vous pouvez créer des clients ou des serveurs COM complètement fonctionnels qui implémentent des objets COM sophistiqués, des serveurs Automation (y compris, des objets Active Server), des contrôles ActiveX ou des fiches ActiveForms.

Remarque Les composants COM tels que ceux situés sur les pages ActiveX, COM+ et Servers de la palette des composants ne sont pas disponibles dans les applications CLX. Cette technologie est spécifique à Windows et n'est pas multiplate-forme.

COM est un modèle de composant logiciel indépendant du langage, conçu pour permettre l'interaction entre des composants logiciel et des applications s'exécutant dans Windows. L'aspect majeur de COM est qu'il permet la communication entre les composants, entre les applications et entre clients et serveurs par le biais d'interfaces clairement définies. Les interfaces représentent pour les clients un moyen d'obtenir à l'exécution les fonctionnalités prises en charge par un composant COM. Pour que votre composant fournisse des fonctionnalités supplémentaires, il suffit d'ajouter une autre interface à ces fonctionnalités.

Les applications peuvent accéder aux interfaces des composants COM s'ils figurent sur le même ordinateur que les applications ou, s'ils figurent sur un autre ordinateur du réseau, à l'aide d'un mécanisme appelé Distributed COM ou DCOM. Pour plus d'informations sur les clients, les serveurs et les interfaces, voir "Composantes d'une application COM" à la page 33-3.

Ce chapitre présente les concepts généraux de la technologie sur laquelle s'appuient l'Automation et les contrôles ActiveX. Les chapitres suivants traiteront en détail de la création d'objets Automation et de contrôles ActiveX dans Delphi.

COM, spécification et implémentation

COM est à la fois une spécification et une implémentation. La spécification COM définit comment des objets sont créés et comment ils communiquent entre eux. Selon cette spécification, les objets COM peuvent être écrits dans différents langages, exécutés dans différents espaces de processus et sur différentes plates-formes. Tant que les objets adhèrent à la spécification, ils peuvent communiquer. Cela vous permet d'intégrer le code de composants existants à de nouveaux composants implémentés dans des langages orientés objet.

L'implémentation COM est construite dans le sous-système Win32, qui fournit de nombreux services intégrés supportant la spécification écrite. La bibliothèque COM contient un ensemble d'interfaces standard définissant la fonctionnalité interne d'un objet COM et un petit ensemble de fonctions API pour la création et la gestion des objets COM.

Lorsque vous utilisez dans votre application les experts de Delphi et les objets de la VCL, vous utilisez l'implémentation Delphi de la spécification COM. En outre, Delphi fournit quelques enveloppes pour les services COM dont les fonctionnalités ne sont pas implémentées directement, comme les documents Active.Ces enveloppes sont définies dans l'unité ComObj et les définitions API se trouvent dans l'unité AxCtrls.

Remarque Les objets et les langages des interfaces Delphi sont conformes à la spécification COM. L'implémentation Delphi de la spécification COM utilise un ensemble de classes appelé cadre de travail Delphi ActiveX (DAX). Ces classes se trouvent dans les unités AxCtrls, OleCtrls et OleServer. En outre, l'interface Pascal à l'API COM se trouve dans ActiveX.pas et ComSvcs.pas.

Extensions de COM

COM a évolué et a été étendu au-delà des services COM de base. COM sert de fondement à d'autres technologies, comme l'Automation, les contrôles ActiveX, les documents Active et les annuaires Active. Pour plus de détails, voir "Extensions de COM" à la page 33-10.

En outre, si vous travaillez dans un environnement distribué important, vous pouvez créer des objets COM transactionnels. Avant Windows 2000, ces objets ne faisaient pas partie de COM, mais s'exécutaient dans l'environnement Microsoft Transaction Server (MTS). Avec l'arrivée de Windows 2000, cette gestion est intégrée dans COM+. Les objets transactionnels sont décrits en détail dans le chapitre 39, "Création d'objets MTS ou COM+".

Delphi fournit des experts permettant d'implémenter facilement des applications qui incorporent toutes ces technologies dans l'environnement Delphi. Pour plus de détails, voir "Implémentation des objets COM à l'aide d'experts" à la page 33-19.

Composantes d'une application COM

Quand vous implémentez une application COM, vous fournissez ceci :

Interface COM Le moyen par lequel un objet expose ses services aux clients. Un objet COM fournit une interface pour chaque ensemble de méthodes. Attention : les propriétés COM ne sont pas identiques aux propriétés des objets VCL. Les propriétés COM utilisent toujours des méthodes d'accès en lecture/écriture.

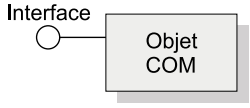
Serveur COM Un module, EXE, DLL ou OCX, contenant le code d'un objet COM. Les implémentations d'objets résident sur les serveurs. Un objet COM implémente une ou plusieurs interfaces.

Client COM Le code appelant les interfaces afin d'obtenir du serveur les services demandés. Les clients savent ce qu'ils veulent obtenir du serveur (via l'interface) ; les clients ne savent pas comment en interne le serveur fournit les services. Delphi facilite le processus de création de client en vous permettant d'installer des serveurs COM (tel qu'un document Word ou une diapositive PowerPoint) en tant que composants sur la palette de composants. Cela vous permet de vous connecter au serveur et de vous lier à ses événements en utilisant l'inspecteur d'objets.

Interfaces COM

Les clients COM communiquent avec des objets par le biais d'interfaces COM. Les interfaces sont des groupes de routines, liées par la logique ou par la sémantique, qui assurent la communication entre le fournisseur d'un service (objet serveur) et ses clients. Voici la représentation standard d'une interface COM :

Figure 33.1 Une interface COM



Par exemple, chaque objet COM implémente l'interface de base, *IUnknown*, qui indique au client les interfaces disponibles sur l'objet COM.

Les objets peuvent avoir plusieurs interfaces, où chacune implémente une fonctionnalité. L'interface est le moyen de mettre à disposition du client le service fourni par l'objet, sans lui donner les détails de l'implémentation sur la façon dont ce service est fourni.

Les aspects majeurs des interfaces COM sont les suivants :

- Une fois publiées, les interfaces sont immuables ; c'est-à-dire qu'elles ne changent plus. Une interface permet d'accéder à un ensemble précis de

fonctions. Les fonctionnalités supplémentaires sont fournies par le biais d'interfaces supplémentaires.

- Par convention, les identificateurs d'interfaces COM commencent par un I majuscule suivi d'un nom symbolique définissant l'interface, comme *IMalloc* ou *IPersist*.
- L'identification unique des interfaces est garantie par un **GUID (Globally Unique Identifier)**, qui est un nombre aléatoire de 128 bits. Les GUID utilisés pour identifier les interfaces sont appelés **IID (Identificateurs d'interfaces)**. Ils permettent d'éliminer les conflits de noms entre différentes versions d'un produit ou différents produits.
- Les interfaces sont indépendantes du langage. Vous pouvez utiliser n'importe quel langage pour implémenter une interface COM, à condition que ce langage supporte les structures de pointeurs et puisse appeler une fonction via un pointeur, de façon explicite ou implicite.
- Les interfaces ne sont pas elles-mêmes des objets ; elles fournissent l'accès à un objet. Donc, les clients n'ont pas accès directement aux données ; ils accèdent aux données par le biais d'un pointeur d'interface. Windows 2000 ajoute une couche supplémentaire d'indirection connue en tant qu'intercepteur à partir duquel il fournit des fonctionnalités COM+ telles que l'activation just-in-time et le regroupement d'objets.
- Les interfaces sont toujours dérivées de l'interface de base, *IUnknown*.
- Les interfaces peuvent être redirigées par COM via des proxy pour permettre aux appels de méthodes de l'interface de s'effectuer entre différents threads, processus et machines en réseau, sans que les objets client ou serveur ne soient jamais informés de la redirection. Pour plus d'informations, voir "Serveurs en processus, hors processus et distants" à la page 33-7.

L'interface COM de base, IUnknown

Les objets COM doivent tous supporter l'interface fondamentale, appelée *IUnknown*, un **typedef** du type d'interface de base IInterface. *IUnknown* contient les routines suivantes :

QueryInterface	Fournit des pointeurs sur d'autres interfaces supportées par l'objet.
AddRef et *Release*	Méthodes simples de décompte de références qui permettent à un objet de contrôler sa durée de vie et de se supprimer lui-même lorsque le client n'a plus besoin de son service.

Les clients obtiennent des pointeurs sur d'autres interfaces via la méthode *QueryInterface* de *IUnknown. QueryInterface* connaît chaque interface de l'objet serveur et peut donner au client un pointeur vers l'interface demandée. Lorsqu'il reçoit un pointeur vers une interface, le client est assuré de pouvoir appeler n'importe quelle méthode de l'interface.

Les objets contrôlent leur propre durée de vie grâce aux méthodes *AddRef* et *Release de IUnknown*, qui sont de simples méthodes de décompte de références.

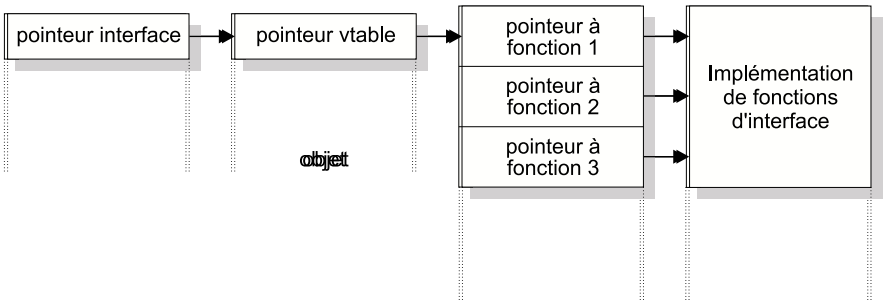
Tant que le décompte de références est différent de zéro, l'objet reste en mémoire. Dès qu'il atteint zéro, l'implémentation de l'interface peut en toute sécurité disposer du ou des objets sous-jacents.

Pointeurs d'interface COM

Un pointeur d'interface est un pointeur de 32 bits vers une instance d'objet qui pointe, à son tour, vers l'implémentation de chaque méthode de l'interface. L'implémentation est accédée via un tableau de pointeurs vers ces méthodes, appelé **vtable**. Les vtables sont similaires au mécanisme utilisé pour gérer les fonctions virtuelles sous Pascal Objet. A cause de cette similitude, le compilateur peut résoudre les appels de méthode de l'interface de la même manière qu'il résout les appels des méthodes de classes Pascal Objet.

La vtable est partagée par toutes les instances d'une classe objet, et pour chaque instance de l'objet, le code de l'objet alloue une deuxième structure contenant ses données privées. Le pointeur d'interface du client, est alors un pointeur vers *le pointeur* vers la vtable, comme le montre le diagramme suivant.

Figure 33.2 Vtable d'interface



Dans Windows 2000 et toutes les versions ultérieures de Windows, quand un objet s'exécute sous COM+, un niveau supplémentaire d'indirection est fourni entre le pointeur d'interface et le pointeur vtable. Le pointeur d'interface disponible sur le client pointe sur un intercepteur, qui à son tour pointe sur la vtable. Cela permet à COM+ de fournir des services comme l'activation just-in-time, par lequel le serveur peut être désactivé et réactivé dynamiquement d'une façon opaque pour le client. COM+ garantit que l'intercepteur se comporte comme s'il était un pointeur vtable ordinaire.

Serveurs COM

Un serveur COM est une application ou une bibliothèque qui fournit des services à une application ou bibliothèque client. Un serveur COM est constitué d'un ou de plusieurs objets COM, un objet COM étant un ensemble de propriétés (données membre ou contenu) et de méthodes (fonctions membre).

Les clients ne savent pas *comment* l'objet COM effectue son service ; l'implémentation de l'objet est encapsulée. Un objet met ses services à disposition par le biais de ses **interfaces** comme décrit précédemment.

En outre, les clients n'ont pas besoin de savoir *où* réside l'objet COM. COM fournit un accès transparent quel que soit l'**emplacement** de l'objet.

Quand il demande un service à un objet COM, le client transmet un identificateur de classe (CLSID) à COM. Un CLSID est juste un GUID qui référence un objet COM. COM utilise ce CLSID, recensé dans le registre système, pour localiser l'implémentation appropriée du serveur. Une fois le serveur localisé, COM amène le code en mémoire, et fait instancier par le serveur une instance de l'objet pour le client. Ce processus est géré indirectement par un objet spécial appelé un fabricant (basé sur les interfaces) qui crée sur demande des instances d'objet.

Au minimum, un serveur COM doit effectuer ceci :

- Recenser des entrées dans le registre système pour associer le module serveur à l'identificateur de classe (CLSID).
- Implémenter un objet fabricant de classe, qui fabrique un autre objet à partir d'un CLSID particulier.
- Exposer le fabricant d'objet à COM.
- Fournir un mécanisme de déchargement grâce auquel un serveur qui ne sert pas de client pourra être supprimé de la mémoire.

Remarque Les experts de Delphi automatisent la création des objets et des serveurs COM comme décrit dans "Implémentation des objets COM à l'aide d'experts" à la page 33-19.

CoClasses et fabricants de classes

Un objet COM est une instance d'une CoClasse, qui est une classe implémentant une ou plusieurs interfaces COM. L'objet COM fournit les services définis par ses interfaces.

Les CoClasses sont instanciées par un type d'objet spécial, appelé un *fabricant de classe*. Chaque fois que des services d'un objet sont demandés par un client, un fabricant de classe crée une instance de cet objet pour ce client particulier. Généralement, si un autre client demande les services de l'objet, le fabricant de classe crée une autre instance de l'objet pour ce deuxième client. Les clients peuvent également se lier à des objets COM en cours d'exécution qui se recensent eux-mêmes pour le gérer.

Une CoClasse doit posséder un fabricant de classe et un identificateur de classe (CLSID) de sorte qu'il puisse être instancié en externe, c'est-à-dire pour un autre module. L'utilisation de ces identificateurs uniques pour les CoClasses implique qu'elles peuvent être mises à jour chaque fois que de nouvelles interfaces sont implémentées dans leur classe. Une nouvelle interface peut modifier ou ajouter des méthodes sans affecter les versions antérieures, ce qui est un problème courant lorsqu'on utilise des DLL.

Les experts de Delphi prennent en compte l'attribution d'identificateurs de classe, l'implémentation et l'instanciation des fabricants de classe.

Serveurs en processus, hors processus et distants

Avec COM, un client n'a pas besoin de savoir où réside un objet, il suffit de faire un appel à une interface de l'objet. COM accomplit les étapes nécessaires à cet appel. Ces étapes sont différentes selon que l'objet réside dans le même processus que le client, dans un autre processus sur la machine du client ou sur une autre machine du réseau. Ces différents types de serveurs sont décrits ici :

Serveur en processus
Une bibliothèque (DLL) s'exécutant dans le *même espace processus* que le client, par exemple, un contrôle ActiveX incorporé dans une page Web visualisée sous Internet Explorer ou Netscape. Le contrôle ActiveX est alors téléchargé sur la machine du client et appelé dans le même processus que le navigateur Web.

Le client communique avec le serveur en processus grâce à des appels directs à l'interface COM.

Serveur hors processus (ou serveur local)
Une autre application (EXE) s'exécutant dans un *espace processus différent* mais sur la *même machine* que le client. Par exemple, une feuille de calcul Excel incorporée dans un document Word constitue deux applications distinctes tournant sur la même machine.

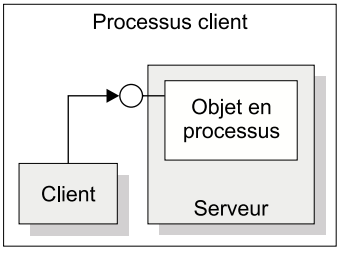
Le serveur local utilise COM pour communiquer avec le client.

Serveur distant
Une DLL ou une autre application s'exécutant sur une *machine différente* de celle du client. Par exemple, une application Delphi de base de données connectée à un serveur d'application sur une autre machine du réseau.

Le serveur distant utilise des interfaces COM distribuées (DCOM) pour accéder aux interfaces et communiquer avec le serveur d'application.

Comme illustré dans la figure suivante, pour les serveurs en processus, les pointeurs sur les interfaces de l'objet sont dans le même espace processus que le client, et COM fait des appels directs dans l'implémentation de l'objet.

Figure 33.3 Serveurs en processus

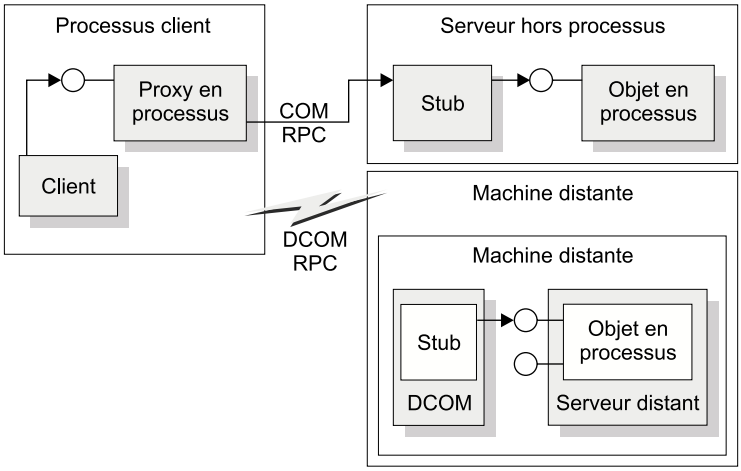


Remarque Cela n'est pas toujours vrai avec COM+. Quand le client appelle un objet dans un contexte différent, COM+ intercepte l'appel afin qu'il se comporte comme l'appel d'un serveur hors processus (voir plus bas) même si le serveur est en

processus. Pour davantage d'informations sur l'utilisation de COM+, voir chapitre 39, "Création d'objets MTS ou COM+".

Comme illustré dans la figure suivante, quand le processus est soit différent, soit sur une autre machine, COM utilise un proxy pour initier les appels de procédure distants. Le **proxy** réside dans le même processus que le client, de sorte que vu du client, tous les appels à des interfaces semblent pareils. Le proxy intercepte l'appel du client et le transmet là où l'objet réel s'exécute. Le mécanisme qui permet aux clients d'accéder aux objets d'un espace processus différent, ou même d'une machine différente, comme s'ils se trouvaient dans leur propre processus, est appelé le **marshaling.**

Figure 33.4 Serveurs hors processus et distants



La différence entre les serveurs hors processus et serveurs distants est le type de communication inter-processus utilisé. Le proxy utilise COM pour communiquer avec un serveur hors processus et COM distribué (DCOM) pour communiquer avec une machine distante. DCOM transfère de manière transparente une demande d'objet local dans l'objet distant s'exécutant sur une autre machine.

Remarque Pour les appels de procédure distantes, DCOM utilise le protocole RPC fourni par l'environnement de calcul distribué (DCE) de Open Source. Pour la sécurité distribuée, DCOM utilise le protocole de sécurité NT LAN Manager (NTLM). Pour les services d'annuaire, DCOM utilise DNS (Domain Name System).

Le mécanisme du marshaling

Le marshaling est le mécanisme qui permet à un client de faire des appels aux fonctions de l'interface d'objets distants qui se trouvent dans un autre processus ou sur une autre machine. Le marshaling

- Prend un pointeur d'interface dans le processus du serveur et rend un pointeur de proxy disponible au code dans le processus du client.
- Prend les arguments d'un appel à l'interface passés depuis le client et les place dans l'espace processus de l'objet distant.

Pour tout appel à l'interface, le client met les arguments sur une pile et émet un appel à une fonction via le pointeur d'interface. Si l'appel à l'objet n'est pas en processus, il est passé au proxy. Celui-ci compresse les arguments dans un paquet de marshaling et transmet la structure à l'objet distant. Le stub de l'objet décompresse le paquet, place les arguments sur la pile et appelle l'implémentation de l'objet. L'objet recrée l'appel du client dans son propre espace d'adressage.

Le type de marshaling dépend de ce que l'objet COM implémente. Les objets peuvent utiliser le mécanisme de marshaling standard fourni par l'interface *IDispatch*. C'est un mécanisme de marshaling générique qui permet la communication via un appel standard à une procédure distante (RPC). Pour plus de détails sur l'interface *IDispatch*, voir "Interfaces d'Automation" à la page 36-13. Quand l'objet n'implémente pas *IDispatch*, s'il se limite lui-même aux types compatibles Automation et s'il a une bibliothèque de types recensée, COM fournit automatiquement la gestion du marshaling.

Les applications qui ne se limitent pas eux-mêmes aux types compatibles automation ou recensent une bibliothèque de types doivent fournir leur propre marshaling. Le marshaling est fourni par le biais d'une implémentation de l'interface *IMarshal*, ou en utilisant une DLL proxy/stub générée séparément. Delphi ne prend pas en charge la génération automatique des DLL proxy/stub.

Agrégation

Dans certains cas, un objet serveur peut utiliser un autre objet COM pour effectuer certaines de ces fonctions. Par exemple, un objet de gestion de stock peut utiliser un objet commande séparé pour gérer les commandes des clients. Si l'objet de gestion de stock veut présenter une interface de commande au client, il y a un problème. Même si le client ayant une interface stock peut appeler *QueryInterface* pour obtenir l'interface commande, quand l'objet commande a été créé, il ne connaît pas l'objet gestion de stock et ne peut lui renvoyer une interface stock en réponse à un appel de *QueryInterface*. Un client qui a l'interface commande ne peut revenir dans l'interface stock.

Pour éviter ce problème, certains objets COM gèrent l'**agrégation**. Quand l'objet gestion de stock crée une instance de l'objet commande, il lui transmet une copie de sa propre interface *IUnknown*. L'objet commande peut ensuite utiliser cette interface *IUnknown* et gérer les appels de *QueryInterface* qui demandent une interface, comme l'interface stock, qu'il ne gère pas. Quand cela se produit, les deux objets pris ensemble s'appellent un agrégat. L'objet commande est appelé l'objet interne (ou objet contenu) de l'agrégat et l'objet stock est appelé l'objet externe.

Remarque Pour pouvoir se comporter comme objet externe d'un agrégat, un objet COM doit créer l'objet interne en utilisant l'API Windows *CoCreateInstance* ou *CoCreateInstanceEx*, en lui transmettant comme paramètre un pointeur *IUnknown* que l'objet interne peut utiliser pour les appels de *QueryInterface*.

Pour créer un objet qui puisse se comporter comme objet interne d'un agrégat, il doit descendre de *TContainedObject*. Quand l'objet est créé, l'interface *IUnknown* de l'objet externe est transmise au constructeur afin qu'elle puisse être utilisée par la méthode *QueryInterface* pour les appels que l'objet interne ne peut pas traiter.

Clients COM

Les clients peuvent toujours interroger les interfaces d'un objet COM pour déterminer ce qu'il est capable de faire. Tous les objets COM permettent aux clients de demander les interfaces connues. De plus, si le serveur gère l'interface *IDispatch*, les clients peuvent demander au serveur des informations sur les méthodes gérées par le serveur. Les objets serveurs n'ont pas d'attentes concernant l'utilisation de ses objets par le client. De même, les clients n'ont pas besoin de savoir comment (ou même si) un objet fournit les services ; ils s'en remettent simplement sur les objets serveurs pour fournir les services qu'ils annoncent via leurs interfaces.

Il y a deux types de clients COM : les contrôleurs et les conteneurs. Un contrôleur lance le serveur et interagit avec lui via son interface. Il demande des services de l'objet COM ou il le pilote comme processus distinct. Les conteneurs accueillent des contrôles visuels ou des objets qui apparaissent dans l'interface utilisateur du conteneur. Ils utilisent des interfaces prédéfinies pour négocier les problèmes d'affichage avec les objets serveur. Il est impossible d'avoir une relation conteneur sur DCOM ; par exemple, les contrôles visuels qui apparaissent dans l'interface utilisateur du conteneur doivent être localisés localement. En effet, les contrôles sont supposés se dessiner eux-mêmes, ce qui nécessite qu'ils puissent accéder aux ressources GDI locales.

Delphi facilite le développement d'un contrôleur Automation en permettant d'importer la bibliothèque de types ou un contrôle ActiveX dans un composant enveloppe de telle manière que les objets serveur apparaissent comme les autres composants VCL. Pour des détails sur ce processus, voir chapitre 35, "Création de clients COM".

Extensions de COM

COM a été initialement conçu pour fournir une fonctionnalité de communication de base et permettre l'enrichissement de cette fonctionnalité via des extensions. COM lui-même a étendu sa fonctionnalité première en définissant des ensembles spécialisés d'interfaces couvrant des besoins spécifiques.

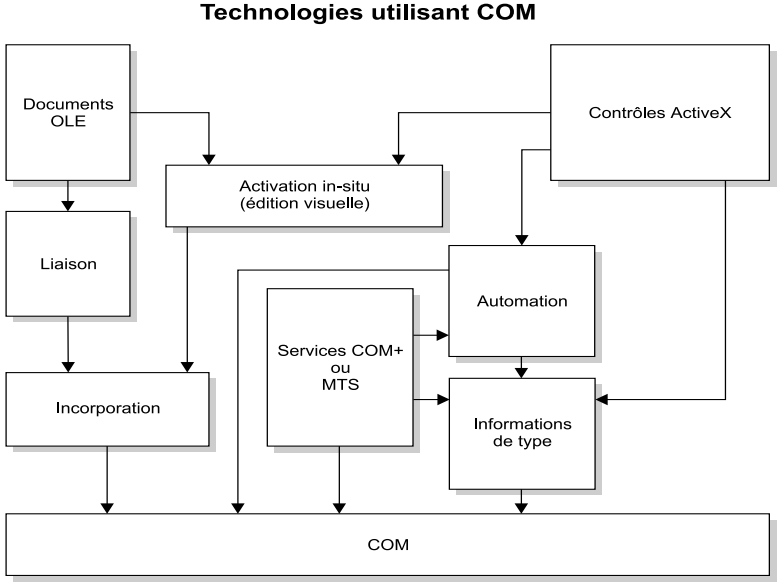
Le tableau suivant est un résumé de certaines des extensions de services que COM fournit actuellement. Les sections suivantes décrivent ces services en détail.

Serveurs Automation	L'Automation désigne la capacité d'une application à contrôler par programme les objets d'une autre application. Les serveurs Automation sont les objets qui peuvent être contrôlés par d'autres exécutables à l'exécution.
Contrôles ActiveX	Les contrôles ActiveX sont des serveurs en processus COM spécialisés, généralement conçus pour être incorporés dans une application client. Les contrôles proposent des comportements et des événements à la conception et à l'exécution.

Pages Active Server	Les pages Active Server sont des scripts qui génèrent des pages HTML. Le langage de script contient des structures permettant la création et l'exécution d'objets Automation, c'est-à-dire qu'une page Active Server se comporte comme un contrôleur Automation
Documents Active	Objets supportant la liaison et l'incorporation, le glisser-déplacer, l'édition visuelle et l'activation in-situ. Les documents Word et les feuilles de calcul Excel sont des exemples de documents Active.
Objets transactionnels	Objets capable de répondre à un grand nombre de clients. Ils proposent des caractéristiques comme l'activation juste-à-temps, les transactions, le regroupement de ressources et des services de sécurité. Ces caractéristiques étaient initialement gérées par MTS, mais ont été intégrées dans COM avec l'arrivée de COM+.
Bibliothèques de types	Collection de structures de données statiques, souvent enregistrées en tant que ressource, fournissant des informations de type détaillées sur un objet et ses interfaces. Les clients des serveurs Automation, les contrôles ActiveX et les objets transactionnels ont besoin des informations de type.

Le diagramme page suivante montre les relations entre les extensions de COM et la façon dont elles dérivent de COM.

Figure 33.5 Technologies COM



Les objets COM peuvent être visuels ou non visuels. Certains s'exécutent dans le même espace processus que leurs clients ; d'autres peuvent s'exécuter dans des

processus différents ou sur des machines distantes si les objets assurent le marshaling. Le tableau suivant récapitule les types d'objets COM que vous pouvez créer, s'ils sont visuels, les espaces processus dans lesquels ils peuvent s'exécuter, le marshaling qu'ils fournissent et s'ils ont besoin d'une bibliothèque de types.

Tableau 33.1 Exigences des objets COM

Objet	Objet visuel ?	Espace processus	Communication	Bibliothèque de types
Document Active	Habituellement	En processus ou hors processus	Verbes OLE	Non
Automation	Parfois	En processus, hors processus ou distant	Marshaling automatique via l'interface *IDispatch* (pour les serveurs hors processus ou distants)	Requise pour le marshaling automatique
Contrôle ActiveX	Habituellement	En processus	Marshaling automatique via l'interface *IDispatch*	Requise
MTS ou COM+	Parfois	En processus pour MTS, tous pour COM+	Marshaling automatique via une bibliothèque de types	Requise
Objet interface personnalisé	Facultativement	En processus	Pas de marshaling requis pour les serveurs en processus	Recommandée
Autre objet interface personnalisé	Facultativement	En processus, hors processus ou distant	Marshaling automatique via une bibliothèque de types ; sinon, marshaling manuel via des interfaces personnalisées	Recommandée

Serveurs Automation

L'Automation désigne la capacité d'une application à contrôler par programme les objets d'une autre application, comme une macro qui peut manipuler plusieurs applications à la fois. Le client d'un objet Automation est appelé contrôleur Automation, et l'objet serveur manipulé est appelé objet Automation.

L'Automation peut être utilisée sur des serveurs en processus, locaux ou distants.

L'Automation présente deux caractéristiques :

- L'objet Automation définit un ensemble de propriétés et de commandes, et décrit ses capacités via les descriptions de type. Pour ce faire, il doit disposer d'un moyen de fournir des informations sur les interfaces de l'objet, les méthodes des interfaces et les arguments de ces méthodes. Généralement, ces informations se trouvent dans des bibliothèques de types. Le serveur Automation peut aussi générer des informations dynamiquement quand elles lui sont demandées via son interface *IDispatch* interface (voir plus bas).

- Les objets Automation rendent ces méthodes accessibles pour que d'autres applications puissent les utiliser. Pour cela, ils implémentent l'interface *IDispatch*. C'est par le biais de cette interface qu'un objet peut exposer toutes ses méthodes et propriétés. Et c'est par le biais de la méthode primaire de cette interface que les méthodes de l'objet peuvent être appelées, une fois qu'elles ont été identifiées grâce aux informations de type.

Les développeurs utilisent souvent l'Automation pour créer et utiliser des objets OLE non visuels qui s'exécutent dans n'importe quel espace processus, car l'interface Automation *IDispatch* automatise le processus de marshaling. En revanche, l'Automation limite les types que vous pouvez utiliser.

Pour obtenir une liste de types compatibles avec les bibliothèques de types en général, et d'interfaces Automation en particulier, voir "Types autorisés" à la page 34-13.

Pour plus d'informations sur l'écriture d'un serveur Automation, voir chapitre 36, "Création de serveurs COM simples".

Pages Active Server

La technologie ASP (pages Active Server) vous permet d'écrire des scripts simples, appelés pages Active Server qui peuvent être exécutées par les clients via un serveur Web. A la différence des contrôles ActiveX qui s'exécutent sur le client, les pages Active Server s'exécutent sur le serveur et renvoient les pages HTML résultantes aux clients.

Les pages Active Server sont écrites en JScript ou en VBscript. Le script est exécuté à chaque fois que le serveur charge la page Web. Ce script peut alors lancer un serveur Automation incorporé (ou un Bean Java Entreprise). Vous pouvez, par exemple, écrire un serveur Automation pour se connecter à une base de données, ce serveur accédant à des données qui sont actualisées à chaque fois que le client charge la page Web.

Les experts Delphi facilitent la création d'un objet Active Server qui est un objet Automation spécialement conçu pour travailler avec une page Active Server. Pour plus d'informations sur la création et l'utilisation de ces types d'objets, voir chapitre 37, "Création d'une page Active Server".

Contrôles ActiveX

ActiveX est une technologie qui permet aux composants COM, particulièrement aux contrôles, d'être plus compacts et efficaces. Cela est particulièrement important pour les contrôles conçus pour des applications Intranet qui nécessitent leur téléchargement par le client avant de les utiliser.

Les contrôles ActiveX sont des contrôles visuels qui s'exécutent uniquement comme des serveurs en processus et qui peuvent s'intégrer dans une application conteneur ActiveX. Ce ne sont pas des applications complètes par eux-mêmes, vous pouvez les voir comme des contrôles OLE préfabriqués qui sont réutilisables dans diverses applications. Les contrôles ActiveX ont une interface

utilisateur apparente et reposent sur l'utilisation d'interfaces prédéfinies pour négocier les entrées/sorties et les questions d'affichage avec le conteneur hôte.

Les contrôles ActiveX utilisent l'Automation pour exposer leurs propriétés, méthodes et événements. Leurs fonctionnalités incluent la capacité à déclencher des événements, la liaison aux sources de données et la gestion de licence.

Les contrôles ActiveX s'utilisent parfois dans un site Web comme objets interactifs placés dans une page Web. Ainsi, ActiveX est devenu un standard particulièrement destiné à des contenus interactifs pour le Web, y compris l'utilisation de documents ActiveX employés pour visualiser des documents non HTML via un navigateur Web. Pour plus d'informations sur la technologie ActiveX, voir le site Web de Microsoft.

Les experts de Delphi facilitent la création des contrôles ActiveX. Pour plus d'informations sur la création et l'utilisation de ces types d'objets, voir chapitre 38, "Création d'un contrôle ActiveX".

Documents Active

Les documents Active (appelés auparavant documents OLE) sont un ensemble de services COM supportant la liaison et l'incorporation, le glisser-déplacer et l'édition visuelle. Les documents Active intègrent de façon transparente des données ou des objets de différents formats, par exemple des clips sonores, des feuilles de calcul, du texte et des images.

Contrairement aux contrôles ActiveX, les documents Active ne sont pas limités aux serveurs en processus ; ils peuvent être utilisés dans des applications inter-processus.

A la différence des objets Automation, qui ne sont presque jamais visuels, les objets document Active peuvent être visuellement actifs dans une autre application. Ils sont associés à deux types de données : les données de représentation utilisées pour l'affichage visuel à l'écran ou sur un périphérique de sortie, et les données natives utilisées pour modifier l'objet.

Les objets document Active peuvent être des conteneurs ou des serveurs de documents. Bien que Delphi ne fournisse pas d'expert pour créer automatiquement des documents Active, vous pouvez utiliser la classe *TOleContainer* de la VCL pour supporter la liaison et l'incorporation dans les documents Active existants.

Vous pouvez aussi utiliser *TOleContainer* comme base d'un conteneur de document Active. Pour créer des objets pour les serveurs de documents Active, utilisez une des classes de base COM de la VCL et implémentez les interfaces appropriées à ce type d'objet, en fonction des services que l'objet doit gérer. Pour plus d'informations sur la création et l'utilisation de serveurs de documents Active, voir le site Web Microsoft.

Remarque Bien que la spécification des documents Active contienne une gestion intégrée du marshaling des applications à processus croisé, les documents Active ne

s'exécutent pas sur des serveurs distants car les types qu'ils utilisent (handles de fenêtre, de menu, etc.) sont spécifiques à un système sur une machine donnée.

Objets transactionnels

Delphi utilise le terme "objets transactionnels" pour désigner des objets qui exploitent les services de transaction, la sécurité et la gestion des ressources proposées par MTS (pour les versions de Windows antérieures à Windows 2000) ou COM+ (pour Windows 2000 et plus). Ces objets sont conçus pour travailler dans des environnements distribués importants.

Les services de transaction garantissent la fiabilité assurant que des activités sont toujours achevées ou annulées (le serveur ne s'arrête jamais en ayant fait la moitié d'une activité). Les services de sécurité vous permettent d'exposer différents niveaux de services à différentes classes de clients. La gestion des ressources permet à un objet de répondre à davantage de clients en regroupant les ressources et en ne gardant des objets actifs que s'ils sont utilisés. Pour permettre au système de proposer ces services, l'objet doit implémenter l'interface *IObjectControl*. Pour accéder aux services, les objets transactionnels utilisent une interface appelée *IObjectContext* qui est créée à leur usage par MTS ou COM+.

Avec MTS, l'objet serveur doit être conçu dans une bibliothèque (DLL) qui est installée dans l'environnement d'exécution MTS. C'est-à-dire que l'objet serveur est un serveur en processus qui s'exécute dans l'espace de processus d'exécution MTS. Avec COM+, cette restriction ne s'applique plus car tous les appels COM sont redirigés par un intercepteur. Pour les clients, les différences entre MTS et COM+ sont transparentes.

Les serveurs MTS ou COM+ rassemblent les objets transactionnels dans le même espace de processus. Dans MTS, ce groupe est appelé un paquet MTS, alors que dans COM+ il est appelé application COM+. Une même machine peut exécuter plusieurs paquets MTS (ou applications COM+), chacun s'exécutant dans son propre espace de processus.

Pour les clients, les objets transactionnels apparaissent semblables aux autres objets serveur COM. Le client n'a pas besoin de savoir quoi que ce soit sur les transactions, la sécurité ou l'activation juste-à-temps, sauf s'il démarre lui-même une transaction.

MTS et COM+ proposent un outil distinct pour administrer les objets transactionnels. Cet outil vous permet de configurer les objets des paquets ou des applications COM+, de visualiser les paquets ou les applications COM+ installés dans une machine ou de modifier les attributs des objets, surveiller et gérer les transactions, mettre des objets à la disposition des clients, etc. Dans MTS, cet outil s'appelle explorateur MTS. Dans COM+ c'est le gestionnaire de composants COM+.

Bibliothèques de types

Les bibliothèques de types offrent un moyen d'obtenir davantage d'informations de type sur un objet que les interfaces de l'objet. Les bibliothèques de types contiennent les informations nécessaires sur les objets et leurs interfaces, comme les interfaces associées à tels objets (étant donné le CLSID), les fonctions membre de chaque interface et les arguments requis par ces fonctions.

Vous pouvez obtenir les informations de type en interrogeant une instance d'un objet pendant qu'elle s'exécute ou, en chargeant et en lisant les bibliothèques de types. Grâce à ces informations, vous pouvez implémenter un client qui utilise un objet souhaité, en sachant exactement les fonctions membre dont vous avez besoin, et ce qu'il faut passer à ces fonctions.

Les clients des serveurs Automation, des contrôles ActiveX et des objets transactionnels s'attendent à disposer de ces informations de type. Tous les experts Delphi génèrent automatiquement une bibliothèque de types (même si c'est facultatif avec l'expert objet COM). Vous pouvez voir ou modifier ces informations de type en utilisant l'**éditeur de bibliothèques de types** comme décrit au chapitre 34, "Utilisation des bibliothèques de types".

Cette section décrit le contenu d'une bibliothèque de types, comment la créer, quand l'utiliser et comment y accéder. Pour les développeurs souhaitant partager des interfaces à travers divers langages, la section se termine par des suggestions sur l'utilisation des outils de gestion de bibliothèques de types.

Contenu d'une bibliothèque de types

Les bibliothèques de types contiennent des *informations de type* qui indiquent quelles interfaces existent et dans quels objets COM, ainsi que le type et le nombre d'arguments des méthodes d'interface. Ces descriptions incluent les identificateurs uniques de CoClasses (CLSID) et d'interfaces (IID), pour que l'utilisateur y accède de façon correcte, ainsi que les identificateurs de répartition (dispID) pour les méthodes et propriétés d'interface Automation.

Les bibliothèques de types peuvent aussi contenir les informations suivantes :

- une description des informations personnalisées de type associées aux interfaces personnalisées
- des routines exportées par le serveur Automation ou ActiveX mais qui ne sont pas des méthodes d'interface
- des informations concernant l'énumération, les enregistrements (structures), les unions, les alias et les types des données des modules
- des références aux descriptions de types issues d'autres bibliothèques de types

Création de bibliothèques de types

Avec les outils de développement traditionnels, vous créez des bibliothèques de types en écrivant des scripts en IDL (Interface Definition Language) ou en ODL (Object Description Language), puis en compilant ces scripts. En revanche,

Delphi génère automatiquement une bibliothèque de types lorsque vous créez un objet COM (contrôles ActiveX, objets Automation, modules de données distant, etc.) en utilisant l'un des experts des pages ActiveX et Multiniveau de la boîte de dialogue Nouveaux Eléments. Vous pouvez décider de ne pas générer de bibliothèque de types si vous utilisez l'expert objet COM. Vous pouvez également créer une bibliothèque de types en sélectionnant Fichier | Nouveau | Autre, l'onglet ActiveX, puis Bibliothèque de types.

Vous pouvez ensuite voir la bibliothèque de types en utilisant l'éditeur de bibliothèques de types Delphi. Il est facile de modifier la bibliothèque de types à l'aide de l'éditeur de bibliothèques de types et Delphi met automatiquement à jour le fichier TLB correspondant quand la bibliothèque de types est enregistrée. Si vous modifiez les interfaces et les CoClasses créées en utilisant un expert, l'éditeur de bibliothèques de types actualise également les fichiers d'implémentation. Pour plus d'informations sur l'utilisation de l'éditeur de bibliothèques de types pour écrire des interfaces et des CoClasses, voir chapitre 34, "Utilisation des bibliothèques de types".

Quand utiliser les bibliothèques de types

Il est important de créer une bibliothèque de types pour chaque ensemble d'objets qui est présenté aux utilisateurs finaux, par exemple,

- Les contrôles ActiveX nécessitent une bibliothèque de types, qui doit être incluse en tant que ressource dans la DLL qui contient les contrôles ActiveX.
- Les objets exposés qui gèrent la liaison de vtable des interfaces personnalisées doivent être décrits dans une bibliothèque de types car les références à la vtable sont liées à la compilation. Les clients importent depuis la bibliothèque de types les informations sur les interfaces et utilisent ces informations pour compiler. Pour plus de détails sur les vtables et les liaisons effectuées lors de la compilation, voir "Interfaces d'Automation" à la page 36-13.
- Les applications qui implémentent des serveurs Automation doivent fournir une bibliothèque de types pour que les clients puissent faire une liaison immédiate.
- Les objets instanciés depuis des classes qui gèrent l'interface *IProvideClassInfo* tels que tous les descendants de la classe VCL *TTypedComObject*, doivent posséder une bibliothèque de types.
- Les bibliothèques de types ne sont pas nécessaires mais utiles pour identifier les objets OLE utilisables par glisser-déplacer.

Si vous définissez des interfaces à usage exclusivement interne (au sein d'une application), il n'est pas nécessaire de créer une bibliothèque de types.

Accès aux bibliothèques de types

En règle générale, une bibliothèque de types fait partie d'un fichier de ressource (.res) ou d'un fichier autonome à l'extension .tlb. Quand elle est placée dans un fichier ressource, la bibliothèque de types peut être liée à un serveur (.DLL, .OCX ou .EXE).

Lorsqu'une bibliothèque de types a été créée, les scruteurs d'objets, les compilateurs et les outils similaires peuvent y accéder par des interfaces spéciales :

Interface	Description
ITypeLib	Fournit des méthodes pour accéder à la description d'une bibliothèque de types.
ITypeLib2	Augmente *ITypeLib* pour inclure la gestion des chaînes de documentation, les données personnalisées et des statistiques sur la bibliothèque de types.
ITypeInfo	Fournit la description de chaque objet d'une bibliothèque de types. Par exemple, un navigateur utilise cette interface pour extraire des informations sur les objets de la bibliothèque de types.
ITypeInfo2	Augmente *ITypeInfo* pour accéder à des informations supplémentaires de la bibliothèque de types, comme les méthodes d'accès aux éléments de données personnalisés.
ITypeComp	Fournit un moyen rapide d'accéder aux informations dont le compilateur a besoin lors de la liaison avec une interface.

Delphi peut importer et utiliser des bibliothèques de types venant d'autres applications en choisissant Projet | Importer une bibliothèque de types. La plupart des classes de la VCL employées pour les applications COM supportent les interfaces essentielles utilisées pour stocker et récupérer les informations de types à partir des bibliothèques de types et des instances actives d'un objet. La classe *TTypedComObject* de la VCL supporte les interfaces qui fournissent des informations de type et s'utilise comme une fondation pour l'environnement objet ActiveX.

Avantages des bibliothèques de types

Même si votre application ne nécessite pas de bibliothèque de types, considérez les avantages suivants :

- La vérification des types peut se faire lors de la compilation.
- Vous pouvez utiliser la liaison immédiate avec l'automation (ce qui remplace les appels par le biais de variants) et les contrôleurs qui ne gèrent pas les vtables ou les interfaces doubles peuvent coder les dispID lors de la compilation pour améliorer les performances de l'application.
- Les scruteurs de types peuvent parcourir la bibliothèque. Les clients pourront donc voir les caractéristiques de vos objets.
- La fonction *RegisterTypeLib* peut être utilisée pour recenser vos objets présentés dans la base de données de recensement.
- La fonction *UnRegisterTypeLib* peut être utilisée pour désinstaller complètement la bibliothèque de types d'une application du registre.
- L'accès local au serveur est accéléré car l'automation utilise les informations de la bibliothèque de types pour regrouper les paramètres qui sont passés à un objet d'un autre processus.

Utilisation des outils de bibliothèques de types

Les outils permettant de manipuler des bibliothèques de types sont indiqués ci-dessous.

- L'outil TLIBIMP (importation de bibliothèque de types), qui crée des fichiers d'interface Delphi (fichiers _TLB.pas) à partir de bibliothèques de types est intégré dans l'éditeur de bibliothèques de types. TLIBIMP offre des options de configuration supplémentaires non disponibles dans l'éditeur de bibliothèques de types.
- TRegSvr est un outil pour recenser et dérecenser les serveurs et les bibliothèques de types, fourni avec Delphi. Le source de TRegSvr est disponible sous la forme d'un exemple dans le répertoire Demos.
- Le compilateur Microsoft IDL (MIDL) compile les scripts IDL pour créer une bibliothèque de types.
- RegSvr32.exe est un utilitaire standard de Windows pour recenser et dérecenser les serveurs et les bibliothèques de types.
- OLEView est un outil de visualisation de bibliothèque de types disponible sur le site Web de Microsoft.

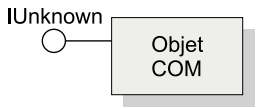
Implémentation des objets COM à l'aide d'experts

Delphi facilite l'écriture de serveurs COM en fournissant des experts qui gèrent nombre des problèmes posés. Delphi propose des experts distincts permettant de créer :

- Un simple objet COM
- Un objet Automation
- Un contrôle ActiveX
- Un objet Active Server (à incorporer dans une page Active Server)
- Un contrôle ActiveX
- Une fiche ActiveX
- Un objet transactionnel
- Une page de propriétés
- Une bibliothèque de types
- Une bibliothèque ActiveX

Les experts gèrent beaucoup des problèmes intervenant dans la création de chaque type d'objet. Ils fournissent les interfaces COM requises pour chaque type d'objet. Comme le montre la figure 33.6, avec un simple objet COM, l'expert implémente la seule interface COM obligatoire, *IUnknown*, qui fournit un pointeur d'interface vers l'objet.

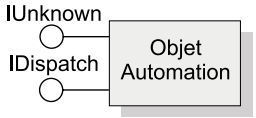
Figure 33.6 Interface d'un objet COM simple



L'expert objet COM propose également une implémentation de *IDispatch* si vous spécifiez que vous créez un objet gérant un descendant de *IDispatch*.

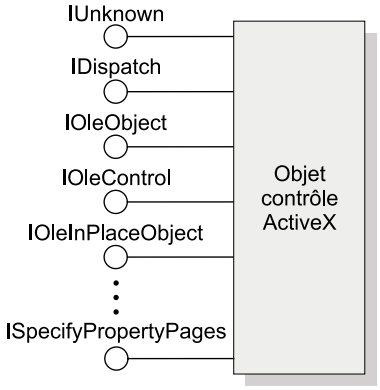
Comme le montre la figure 33.7, pour les objets Automation et Active Server, l'expert implémente *IUnknown* et *IDispatch*, qui fournissent le marshaling automatique.

Figure 33.7 Interfaces d'un objet Automation



Comme le montre la figure 33.8, pour les objets contrôles ActiveX et fiches ActiveX, l'expert implémente toutes les interfaces requises par les contrôles ActiveX : *IUnknown*, *IDispatch*, *IOleObject*, *IOleControl*, etc. Pour obtenir la liste complète des interfaces, reportez-vous à la page de référence de l'objet *TActiveXControl*.

Figure 33.8 Interfaces d'un contrôle ActiveX



Le tableau suivant énumère les divers experts et les interfaces qu'ils implémentent :

Tableau 33.2 Experts Delphi qui implémentent des objets COM, Automation et ActiveX

Expert	Interfaces implémentées	Ce que fait l'expert
serveur COM	IUnknown (et *IDispatch* si vous sélectionnez une interface par défaut qui descend de *IDispatch*)	Exporte les routines nécessaires pour gérer le recensement du serveur, le recensement des classes, le chargement et le déchargement du serveur et l'instanciation des objets. Crée et gère les fabricants de classes pour les objets implémentés sur le serveur. Fournit les entrées de registre pour l'objet spécifiant le modèle de thread sélectionné. Déclare les méthodes qui implémentent l'interface sélectionnée et fournit un squelette d'implémentation que vous devez compléter. Fournit une bibliothèque de types, si elle est demandée. Vous permet de sélectionner une interface arbitraire qui est recensée dans la bibliothèque de types et de l'implémenter. Pour cela, vous devez utiliser une bibliothèque de types.
serveur Automation	*IUnknown*, *IDispatch*	Effectue les actions d'un expert serveur COM (décrit plus haut) plus : Implémente l'interface que vous spécifiez, double ou de répartition.Fournit, si c'est demandé, une gestion par le serveur de la génération d'événements. Fournit automatiquement une bibliothèque de types.
objet Active Server	*IUnknown*, *IDispatch*, *(IASPObject)*	Effectue les actions d'un expert objet Automation (décrit plus haut) et génère facultativement une page .ASP qui peut être chargée dans un navigateur Web. Vous laisse dans l'éditeur de bibliothèques de types pour que vous puissiez modifier les propriétés et les méthodes de l'objet, si nécessaire. Restitue les éléments intrinsèques ASP sous la forme de propriétés afin de pouvoir disposer facilement d'informations sur l'application ASP et le message HTTP qui l'a démarrée.

Tableau 33.2 Experts Delphi qui implémentent des objets COM, Automation et ActiveX (suite)

Expert	Interfaces implémentées	Ce que fait l'expert
contrôle ActiveX	*IUnknown*, *IDispatch*, *IPersistStreamInit*, *IOleInPlaceActiveObject*, *IPersistStorage*, *IViewObject*, *IOleObject*, *IViewObject2*, *IOleControl*, *IPerPropertyBrowsing*, *IOleInPlaceObject*, *ISpecifyPropertyPages*	Effectue les actions de l'expert Automation (décrites plus haut) plus : Génère uneCoClasse qui correspond au contrôle VCL sur lequel est basé le contrôle ActiveX et qui implémente toutes les interfaces ActiveX. Vous laisse dans l'éditeur de code source pour que vous puissiez modifier la classe d'implémentation.
ActiveForm	Mêmes interfaces que contrôle ActiveX	Effectue les actions de l'expert contrôle ActiveX, plus : Crée un descendant de *TActiveForm qui prend la place de la classe* VCL préexistante dans l'expert contrôle ActiveX. Cette nouvelle classe vous permet de concevoir la fiche active de la même manière que vous concevez une application Windows.
objet transactionnel	*IUnknown*, *IDispatch*, *IObjectControl*	Ajoute une nouvelle unité au projet en cours contenant la définition de l'objet MTS ou COM+. Il insère des GUID propriétaires dans la bibliothèque de types afin que Dephi puisse installer correctement l'objet et vous laisse dans l'éditeur de bibliothèque de types pour que vous puissiez définir l'interface que l'objet expose aux clients. Vous devez installer l'objet séparément après l'avoir généré.
page de propriétés	*IUnknown*, *IPropertyPage*	Crée une nouvelle page de propriétés que vous pouvez concevoir dans le concepteur de fiche.
objet événement COM+	Aucune par défaut	Crée un objet événement COM+ que vous pouvez définir en utilisant l'éditeur de bibliothèque de types. A la différence des autres experts, l'expert objet événement COM+ ne crée pas d'unité d'implémentation car les objets événement n'ont pas d'implémentation (elle est fournie par les récepteurs d'événements des clients).
bibliothèque de types	Aucune par défaut	Crée une nouvelle bibliothèque de types et l'associe au projet actif.
bibliothèque ActiveX	Aucune par défaut	Crée une nouvelle DLL ActiveX ou serveur COM et expose les fonctions d'exportation nécessaires.

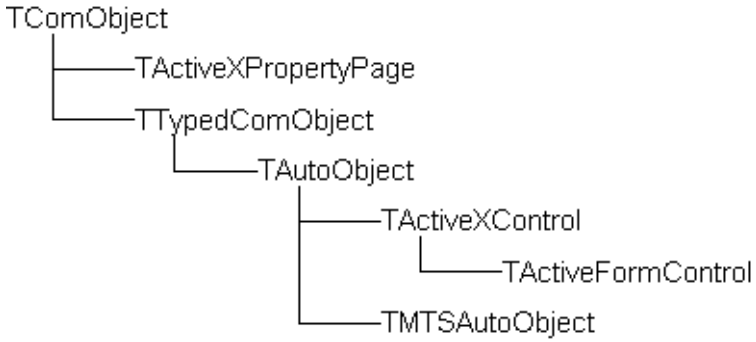
Si vous voulez, vous pouvez ajouter d'autres objets COM (ou refaire une implémentation existante). Pour ajouter un nouvel objet, il est plus simple d'utiliser l'expert une seconde fois. En effet, l'expert met en place l'association entre la bibliothèque de types et une classe d'implémentation, ainsi les

modifications effectuées dans l'éditeur de bibliothèques de types sont automatiquement appliquées à l'unité d'implémentation.

Code généré par les experts

DAXLes experts Delphi génèrent des classes qui dérivent du modèle ActiveX Delphi (DAX). En dépit de son nom, DAX gère tous les types d'objet COM et pas uniquement les contrôles ActiveX. Les classes de ce modèle fournissent l'implémentation sous-jacente des interfaces COM aux objets que vous créez en utilisant un expert. La figure suivante illustre les objets du modèle DAX :

Figure 33.9 Modèle ActiveX Delphi



Chaque expert génère une unité d'implémentation qui implémente votre objet serveur COM. L'objet serveur COM (l'objet implémentation) descend de l'une des classe DAX :

Tableau 33.3 Classes de base DAX utilisées par les classes d'implémentation générées

Expert	Classe de base DAX	Gestion héritée
serveur COM	TTypedCOMObject	Gestion des interfaces IUnknown ISupportErrorInfo. Gestion de l'agrégation, des exceptions OLE, de la convention d'appel safecall et des conventions sur les interfaces doubles. Gestion de la lecture des informations des bibliothèques de types.
Serveur Automation Serveur Active Object	TAutoObject	Tout ce qui est fourni par *TTypedCOMObject*, plus : Gestion de l'interface IDispatch. Gestion de l'auto-marshaling.

Tableau 33.3 Classes de base DAX utilisées par les classes d'implémentation générées (suite)

Expert	Classe de base DAX	Gestion héritée
contrôle ActiveX	TActiveXControl	Tout ce qui est fourni par *TAutoObject*, plus : Gestion de l'incorporation dans un conteneur. Gestion de l'activation in-situ. Gestion des propriétés et des pages de propriétés. La capacité de déléguer à un contrôle fenêtre associé qu'il crée.
ActiveForm	TActiveFormControl	Tout ce qui est fourni par *TAutoObject*, sauf qu'il fonctionne avec un descendant *TActiveForm* au lieu d'une autre classe de contrôle fenêtré.
objets MTS	TMTSAutoObject	Tout ce qui est fourni par *TAutoObject*, plus : Gestion de l'interface IObjectControl.
page de propriétés	TPropertyPage (utilise TActiveXPropertyPage en interne)	Gestion des interfaces IUnknown et ISupportErrorInfo. Gestion de l'agrégation, des exceptions OLE, de la convention d'appel safecall et des conventions sur les interfaces doubles. Gestion de l'interface IPropertyPage.

Une hiérarchie d'objets fabricant de classe correspond aux classes de la figure 33.9 et gère la création de ces objets COM. L'expert ajoute du code à la section initialisation de votre unité d'implémentation pour instancier le fabricant de classe approprié à votre classe d'implémentation.

Les experts génèrent également une bibliothèque de types et son unité associée, qui porte un nom de la forme Projet1_TLB. L'unité Project1_TLB contient les définitions dont votre application a besoin pour utiliser les types et les interfaces définis dans la bibliothèque de types. Pour davantage d'informations sur le contenu de ce fichier, voir "Code généré par l'importation des informations d'une bibliothèque de types" à la page 35-5.

Vous pouvez modifier l'interface générée par l'expert en utilisant l'éditeur de bibliothèques de types. Si vous le faites, la classe d'implémentation est automatiquement actualisée pour refléter vos modifications. Il vous suffit juste de remplir le corps des méthodes générées pour compléter l'implémentation.

Chapitre

34

Utilisation des bibliothèques de types

Ce chapitre décrit comment créer et modifier des bibliothèques de types en utilisant l'éditeur de bibliothèques de types Delphi. Les bibliothèques de types sont des fichiers incluant des informations sur les types de données, les interfaces, les fonctions membre et les classes d'objet exposés par un objet COM. Les bibliothèques de types permettent d'identifier le type des objets et des interfaces disponibles sur un serveur. Pour une présentation détaillée du rôle et de l'utilisation des bibliothèques de types, voir "Bibliothèques de types" à la page 33-16

Une bibliothèque de types peut contenir les éléments suivants :

- Des informations sur les types de données personnalisés, dont les alias, les énumérations, les structures et les unions.
- Les descriptions d'un ou de plusieurs éléments COM, par exemple une interface, une dispinterface ou une CoClasse. Ces descriptions sont appelées *informations de types.*
- Les descriptions des constantes et méthodes définies dans les unités externes.
- Des références aux descriptions de type situées dans d'autres bibliothèques de types.

Quand vous incluez une bibliothèque de types dans une application COM ou une bibliothèque ActiveX, vous mettez à la disposition d'autres applications et outils de programmation des informations sur les objets de votre application via les outils et les interfaces de la bibliothèque de types.

Avec les outils de développement traditionnels, vous créez des bibliothèques de types en écrivant des scripts utilisant le langage IDL (Interface Definition Language) ou ODL (Object Description Language), puis en exécutant ce script via un compilateur. L'éditeur de bibliothèques de types automatise ce processus, simplifiant ainsi la création ou la modification de vos bibliothèques de types.

Quand vous créez un serveur COM de type quelconque (contrôle ActiveX, objet Automation, module de données distant, etc.) en utilisant les experts Delphi, l'expert génère automatiquement une bibliothèque de types (même si dans le cas de l'expert objet COM, cela est optionnel). L'essentiel de votre travail de personnalisation de l'objet généré commence dans la bibliothèque de types. C'est là où vous définissez les propriétés et méthodes qu'elle expose à ses clients : vous modifiez l'interface de la CoClasse générée par l'expert en utilisant l'éditeur de bibliothèques de types. L'éditeur de bibliothèques de types actualise automatiquement l'unité d'implémentation de votre objet afin qu'il soit juste nécessaire de remplir le corps des méthodes générées.

Vous pouvez aussi utiliser l'éditeur de bibliothèques de types Delphi dans le développement d'applications CORBA (Common Object Request Broker Architecture). Avec les outils CORBA traditionnels, vous devez définir les interfaces d'objets en dehors de votre application, à l'aide du langage CORBA IDL (Interface Definition Language). Vous exécutez ensuite un utilitaire qui génère du code stub-et-squelette à partir de cette définition. Toutefois, Delphi génère le stub, le squelette et le code IDL automatiquement. Vous pouvez facilement modifier votre interface à l'aide de l'éditeur de bibliothèques de typesDelphi se chargeant de mettre automatiquement à jour les fichiers source correspondants.

L'éditeur de bibliothèques de types

L'éditeur de bibliothèques de types est un outil permettant aux développeurs d'examiner et de créer les informations de type sur les objets COM. L'utilisation de l'éditeur de bibliothèques de type permet de simplifier considérablement le développement d'objets COM en centralisant la définition des interfaces, des CoClasses et des types, l'obtention de GUID pour les nouvelles interfaces, l'association d'interfaces à des CoClasses, l'actualisation des unités d'implémentation, etc.

Remarque L'éditeur de bibliothèques de types est également utilisé pour définir des interfaces CORBA dans des projets utilisant l'objet CORBA ou l'expert module de données CORBA.

L'éditeur de bibliothèques de types produit deux types de fichiers représentant le contenu d'une bibliothèque de types :

Tableau 34.1 Fichiers de l'éditeur de bibliothèques de types

Fichier	Description
Fichier .TLB	C'est le fichier binaire de la bibliothèque de types. Par défaut, vous n'avez pas besoin d'utiliser ce fichier car la bibliothèque de types est automatiquement compilée dans l'application sous la forme d'une ressource. Cependant vous pouvez utiliser ce fichier pour compiler explicitement la bibliothèque de types dans un autre projet ou pour déployer la bibliothèque de types séparément du fichier .exe ou .ocx. Pour plus d'informations, voir "Ouverture d'une bibliothèque de types existante" à la page 34-22 et "Déploiement des bibliothèques de types" à la page 34-30. **Remarque :** si vous utilisez l'éditeur de bibliothèques de types pour des interfaces CORBA, l'éditeur de bibliothèques de types ne crée pas le fichier .tlb.
Unité _TLB	Cette unité interprète le contenu de la bibliothèques de types pour son utilisation par l'application. Elle contient toutes les déclarations dont l'application a besoin pour utiliser les éléments définis dans la bibliothèque de types. Même si vous pouvez ouvrir ce fichier dans l'éditeur de code, vous ne devez jamais le modifier : il est géré automatiquement par l'éditeur de bibliothèques de types et vos modifications seraient écrasées par l'éditeur de bibliothèques de types. Pour plus de détails sur le contenu de ce fichier, voir "Code généré par l'importation des informations d'une bibliothèque de types" à la page 35-5. **Remarque :** si vous utilisez l'éditeur de bibliothèques de types pour des interfaces CORBA, cette unité définit les objets stub et squelette nécessaires à l'application CORBA.

Composants de l'éditeur de bibliothèques de types

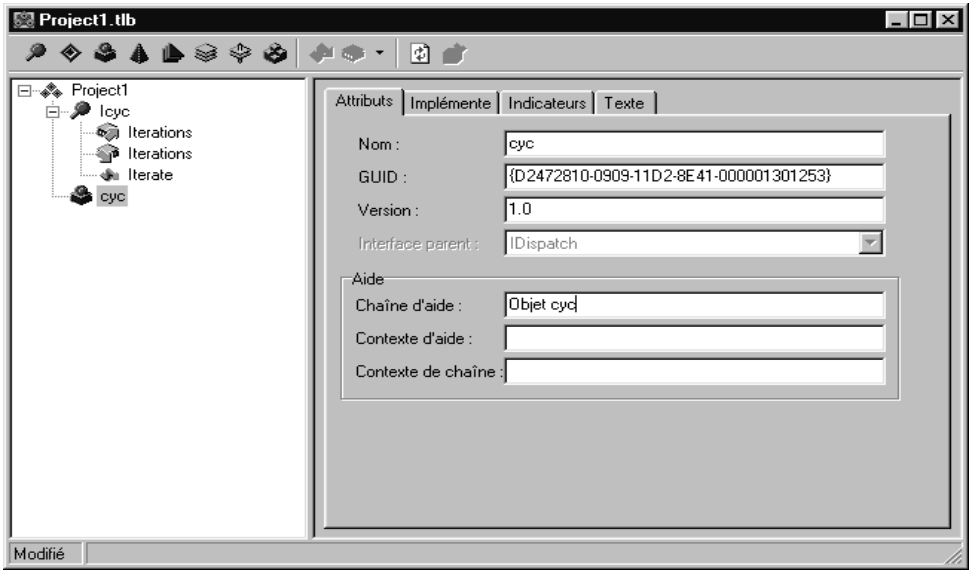
Les principaux éléments de l'éditeur de bibliothèques de types sont décrits dans le tableau suivant.

Tableau 34.2 Composants de l'éditeur de bibliothèques de types

Partie	Description
Barre d'outils	Comprend des boutons vous permettant d'ajouter à la bibliothèque de types de nouveaux types, CoClasses, interfaces et membres d'interface. La barre d'outils propose également des boutons pour actualiser l'unité d'implémentation, recenser la bibliothèque de types et enregistrer un fichier IDL avec les informations de la bibliothèques de types.
Volet liste des objets	Affiche tous les éléments existant dans la bibliothèque de types. Quand vous cliquez sur un élément du volet liste des objets, il affiche les pages adaptées à cet objet.
Barre d'état	Affiche les erreurs de syntaxe si vous essayez d'ajouter des types illégaux à la bibliothèque de types.
Pages	Affichent des informations sur l'objet sélectionné. Les pages affichées dépendent du type d'objet sélectionné.

Ces éléments sont illustrés dans le tableau suivant qui présente l'éditeur de bibliothèques de types affichant les informations de type pour un objet COM nommé cyc.

Figure 34.1 L'éditeur de bibliothèques de types



Barre d'outils

La barre d'outils de l'éditeur de bibliothèques de types, située en haut, contient des boutons sur lesquels vous cliquez pour ajouter de nouveaux objets à la bibliothèque de types.

Le premier groupe de boutons vous permet d'ajouter des éléments à la bibliothèque de types. Quand vous cliquez sur un bouton de la barre d'outils, l'icône de cet élément apparaît dans le volet liste des objets. Vous pouvez alors personnaliser ses attributs dans le volet de droite. Selon le type d'icône sélectionnée, différentes pages d'informations apparaissent à droite.

Le tableau suivant énumère les éléments qu'il est possible d'ajouter à une bibliothèque de types :

Icône	Signification
	Une description d'interface.
	Une description de dispinterface. (ne s'utilise pas pour la définition d'interface CORBA)
	Une CoClasse.

Icône	Signification
	Une énumération.
	Un alias.
	Un enregistrement.
	Une union.
	Un module.

Quand vous sélectionnez un des éléments listés ci-dessus, dans le volet liste des objets, le second groupe de boutons affiche les membres autorisés pour cet élément. Par exemple, quand vous sélectionnez Interface, les icônes de méthode et de propriété deviennent disponibles dans le second groupe car vous pouvez ajouter des propriétés et des méthodes à une définition d'interface. Quand vous sélectionnez une énumération, le second groupe de boutons n'affiche que le membre constante car c'est le seule membre utilisable pour spécifier les informations d'un type énumération.

Le tableau suivant énumère les membres qu'il est possible d'ajouter aux éléments du volet liste des objets :

Icône	Signification
	Une méthode de l'interface, de la dispinterface ou un point d'entrée d'un module.
	Une propriété d'une interface ou d'une dispinterface.
	Une propriété en écriture seule (disponible dans la liste déroulante du bouton propriété).
	Une propriété en lecture-écriture (disponible dans la liste déroulante du bouton propriété).
	Une propriété en lecture seule (disponible dans la liste déroulante du bouton propriété).
	Un champ d'un enregistrement ou d'une union.
	Une constante d'une énumération ou d'un module.

Dans le troisième groupe de boutons, vous pouvez actualiser, recenser ou exporter la bibliothèque de types (l'enregistrer dans un fichier), comme décrit

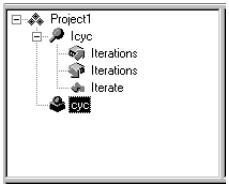
dans la section "Enregistrement et recensement des informations d'une bibliothèque de types" à la page 34-28.

Volet liste des objets

Le volet liste des objets affiche tous les éléments de la bibliothèque de types en cours dans une vue arborescente. La racine de l'arborescence représente la bibliothèque de types, elle est représentée par l'icône suivante :

Descendant du nœud bibliothèque de types, vous trouvez les éléments de la bibliothèque de types :

Figure 34.2 Le volet liste des objets



Quand vous sélectionnez l'un de ces éléments (y compris la bibliothèque de types), les pages d'informations de types à droite sont actualisées pour n'afficher que les informations pertinentes pour cet élément. Vous pouvez utiliser ces pages pour modifier la définition et les propriétés de l'élément sélectionné.

Vous pouvez manipuler les éléments du volet liste des objets en cliquant avec le bouton droit de la souris afin d'afficher le menu contextuel du volet liste des objets. Ce menu propose des commandes qui vous permettent d'utiliser le presse-papiers Windows pour déplacer ou copier des éléments, d'ajouter de nouveaux éléments ou de personnaliser l'aspect de l'éditeur de bibliothèques de types.

Barre d'état

Lors de la modification ou de l'enregistrement d'une bibliothèque de types, les erreurs de syntaxe, les erreurs de traduction et les avertissements sont affichés dans le volet barre d'état.

Si, par exemple, vous spécifiez un type non géré par l'éditeur de bibliothèques de types, vous obtiendrez une erreur de syntaxe. Pour une liste complète des types gérés par l'éditeur de bibliothèques de types, voir "Types autorisés" à la page 34-13.

Les pages d'informations de type

Quand vous sélectionnez un élément dans le volet liste des objets, les pages des informations de type qui apparaissent dans l'éditeur de bibliothèques de types sont celles autorisées pour l'objet sélectionné.

Le tableau suivant indique les pages affichées selon l'élément sélectionné dans le volet liste des objets :

Elément de type d'information	Page d'information de type	Contenu de la page
Bibliothèque de type	Attributs	Nom, version et GUID de la bibliothèque de types ainsi que des informations liant la bibliothèque de types à l'aide.
	Utilise	Liste des autres bibliothèques de types contenant des définitions dont celle-ci dépend.
	Indicateurs	Indicateurs déterminant comment d'autres applications peuvent utiliser la bibliothèque de types.
	Texte	Toutes les définitions et déclarations définissant la bibliothèque de types même (voir l'explication ci-dessous).
Interface	Attributs	Nom, version et GUID de l'interface, le nom de l'interface dont elle descend et des informations liant l'interface à l'aide.
	Indicateurs	Indicateurs spécifiant si l'interface est cachée, double, compatible Automation et/ou extensible.
	Texte	Les définitions et déclarations de l'interface (voir l'explication ci-dessous)
Dispinterface	Attributs	Nom, version et GUID de la dispinterface, le nom de l'interface dont elle descend et des informations liant l'interface à l'aide.
	Indicateurs	Indicateurs spécifiant si la dispinterface est cachée, double et/ou extensible.
	Texte	Les définitions et déclarations de la dispinterface (voir l'explication ci-dessous).
CoClasse	Attributs	Nom, version et GUID de la CoClasse et des informations la liant à l'aide.
	Implémente	Une liste des interfaces que la CoClasse implémente ainsi que leurs attributs.
	COM+	Les attributs des objets transactionnels (modèle de transaction, appel de synchronisation, activation juste à temps, groupement d'objets, etc. Contient également les attributs des objets événement COM+.
	Indicateurs	Indicateurs spécifiant divers attributs de la CoClasse, y compris la manière dont les clients peuvent créer et utiliser des instances, si elle est visible pour les utilisateurs dans un navigateur, si c'est un contrôle ActiveX, et si elle peut être agrégée (comme partie d'un composite).
	Texte	Les définitions et déclarations de la CoClasse (voir l'explication ci-dessous).
Enumération	Attributs	Nom, version et GUID de l'énumération et des informations la liant à l'aide.
	Texte	Les définitions et déclarations du type énuméré (voir l'explication ci-dessous).
Alias	Attributs	Nom, version et GUID du type que l'alias représente et des informations le liant à l'aide.
	Texte	Les définitions et déclarations de l'alias (voir l'explication ci-dessous)

Elément de type d'information	Page d'information de type	Contenu de la page
Enregistrement	Attributs	Nom, version et GUID de l'enregistrement et des informations le liant à l'aide.
	Texte	Les définitions et déclarations de l'enregistrement (voir l'explication ci-dessous)
Union	Attributs	Nom, version et GUID de l'union et des information la liant à l'aide.
	Texte	Les définitions et déclarations de l'union (voir l'explication ci-dessous).
Module	Attributs	Nom, version, GUID et DLL associée du module, et des informations le liant à l'aide.
	Texte	Les définitions et déclarations du module (voir l'explication ci-dessous).
Méthode	Attributs	Nom, identificateur de répartition ou point d'entrée de DLL et des informations liant la méthode à l'aide.
	Paramètres	Type de valeur renvoyée par la méthode et une liste de tous les paramètres avec leur type et tous les modificateurs.
	Indicateurs	Indicateurs spécifiant comment les clients peuvent visualiser et utiliser la méthode, si c'est la méthode par défaut de l'interface et si elle est remplaçable.
	Texte	Les définitions et déclarations de la méthode (voir l'explication ci-dessous).
Propriété	Attributs	Nom, identificateur de répartition, type de méthode d'accès à la propriété (lecture ou écriture) et des informations la liant à l'aide.
	Paramètres	Type de valeur renvoyée par la méthode d'accès et une liste de tous les paramètres avec leur type et tous les modificateurs.
	Indicateurs	Indicateurs spécifiant comment les clients peuvent visualiser et utiliser la propriété, si c'est la propriété par défaut de l'interface, si elle est remplaçable, si elle peut être liée, etc.
	Texte	Les définitions et déclarations de la méthode d'accès à la propriété (voir l'explication ci-dessous).
Constante	Attributs	Nom, valeur et type (pour les constantes de module) et des informations liant la constante à l'aide.
	Indicateurs	Indicateurs spécifiant comment les clients peuvent visualiser et utiliser la constante, si elle représente une valeur par défaut, si la constante peut être liée, etc.
	Texte	Les définitions et déclarations de la constante (voir l'explication ci-dessous).
Champ	Attributs	Nom, type et des informations liant le champ à l'aide
	Indicateurs	Indicateurs spécifiant comment les clients peuvent visualiser et utiliser le champ, s'il représente une valeur par défaut, si le champ peut être lié, etc.
	Texte	Les définitions et déclarations du champ (voir l'explication ci-dessous).

Remarque Pour davantage d'informations sur les diverses options définissables dans les pages d'informations de type, voir l'aide en ligne de l'éditeur de bibliothèques de types.

Vous pouvez utiliser chacune des pages d'informations de type pour visualiser ou modifier les valeurs qu'elles affichent. La plupart des pages organisent les informations dans un ensemble de contrôles afin que vous puissiez saisir des valeurs ou les sélectionner sans avoir à connaître la syntaxe des déclarations correspondantes. Cela évite beaucoup d'erreurs de saisie quand vous spécifiez une valeur comprise dans un ensemble déterminé. Cependant, vous pouvez trouver plus efficace de saisir directement la déclaration. Pour ce faire, utilisez la page Texte.

Tous les éléments des bibliothèques de types disposent d'une page Texte qui affiche la syntaxe de l'élément. Cette syntaxe utilise un sous-ensemble du langage Microsoft IDL (Interface Definition) ou Pascal Objet. Les modifications effectuées dans les autres pages de l'élément sont reflétées ici. Si vous ajoutez du code directement dans la page Texte, les modifications sont reflétées dans les autres pages de l'éditeur de bibliothèques de types.

L'éditeur de bibliothèques de types génère des erreurs de syntaxe si vous ajoutez des identificateurs qui ne sont pas gérés par l'éditeur ; l'éditeur ne gère actuellement que les identificateurs associés à la gestion de bibliothèques (et pas ceux associés à la gestion RPC ou les constructions utilisées par le compilateur Microsoft IDL pour la génération de code C++ ou la gestion du marshalling).

Eléments d'une bibliothèque de types

Au premier abord, l'interface d'une bibliothèque de types peut sembler un peu complexe. Cela est du au fait qu'elle présente des informations sur un grand nombre d'éléments, chacun ayant des caractéristiques spécifiques. Cependant, la plupart de ces caractéristiques sont communes à tous les éléments. Ainsi, chaque élément (dont la bibliothèque de types) dispose des caractéristiques suivantes :

- Un nom utilisé pour décrire l'élément et désigner l'élément dans du code.
- Un GUID (identificateur global unique) qui est une valeur globalement unique de 128 bits utilisée par COM pour identifier l'élément. Cette valeur doit toujours être fournie pour la bibliothèque de types, les CoClasses et les interfaces. Sinon elle est facultative.
- Un numéro de version, qui distingue différentes versions de l'élément. Cette information est toujours facultative, mais elle doit être spécifiée pour les CoClasses et les interfaces car certains outils ne peuvent les utiliser si elles n'ont pas un numéro de version.
- Des informations liant l'élément à une rubrique d'aide. Elles sont constituées d'une chaîne d'aide, d'un contexte d'aide ou d'une valeur de contexte de chaîne d'aide. Le contexte d'aide est utilisé avec le système d'aide classique Windows si la bibliothèque d'aide dispose d'un fichier d'aide autonome propre. Le contexte de chaîne d'aide est utilisé quand l'aide est fournie par une DLL distincte. Le contexte d'aide ou le contexte de chaîne d'aide fait

référence à un fichier d'aide ou une DLL dont le nom est spécifié dans la page Attributs de la bibliothèque de types. Ces informations sont toujours facultatives.

Interfaces

L'interface décrit les méthodes (et les propriétés exprimées comme fonctions get/set) d'un objet auquel il faut d'accéder via une table de fonction virtuelle (VTable). Si une interface est définie comme double, ce qui est le cas par défaut, une dispinterface est également impliquée et peut être atteinte via l'automation OLE. Par défaut, la bibliothèque de types marque toutes les interfaces ajoutées comme double.

Il est possible d'attribuer des membres aux interfaces : des méthodes et des propriétés. Elles apparaissent dans le volet liste des objets comme enfant du nœud interface. Les propriétés des interfaces sont représentées par les méthodes get et set utilisées pour lire et écrire les données sous-jacentes de la propriété. Elles sont représentées dans la vue arborescente par des icônes spéciales qui indiquent leur fonction.

Remarque Si une propriété est spécifiée comme "Ecriture par référence", cela signifie qu'elle est transmise comme pointeur et non comme valeur. Certaines applications, comme Visual Basic, utilisent si possible l'écriture par référence pour optimiser les performances. Pour passer une propriété uniquement par référence plutôt que par valeur, utilisez le type de propriété Par référence seulement. Pour transmettre la propriété par adresse et par valeur, sélectionnez Lecture | Ecriture | Ecriture par référence. Pour activer ce menu, sélectionnez dans la barre d'outils la flèche en regard de l'icône de la propriété.

Une fois les propriétés et méthodes ajoutées en utilisant les boutons de la barre d'outils ou le menu contextuel du volet liste des objets, vous pouvez décrire leur syntaxe et leurs attributs en sélectionnant la propriété ou la méthode et en utilisant les pages d'informations de type.

La page Attributs vous permet d'attribuer un nom à la propriété ou méthode et un identificateur de répartition (afin de pouvoir l'appeler en utilisant IDispatch). Pour les propriétés, vous pouvez également attribuer un type. La signature de fonction est créée en utilisant la page Paramètres qui vous permet d'ajouter, de supprimer ou d'organiser les paramètres, de définir leur type et les modificateurs et de spécifier le type de retour de la fonction.

Remarque Les membres des interfaces qui veulent déclencher des exceptions doivent renvoyer un HRESULT et spécifier un paramètre de valeur renvoyée (PARAM_RETVAL) pour la valeur réellement renvoyée. Déclarez ces méthodes en utilisant la convention d'appel **safecall**.

Si vous attribuez des propriétés et des méthodes à une interface, elles sont implicitement attribuées à sa CoClasse associée. C'est pour cela que l'éditeur de bibliothèques de types ne vous permet pas d'ajouter directement des méthodes ou des propriétés à une CoClasse.

Dispinterfaces

Les interfaces sont plus fréquemment utilisées que les dispinterfaces pour décrire les propriétés et méthodes d'un objet. Les dispinterfaces sont accessibles uniquement au travers d'une liaison dynamique alors que les interfaces peuvent aussi utiliser une liaison statique grâce à une vtable.

Vous pouvez ajouter des propriétés et méthodes aux dispinterfaces comme pour les interfaces. Cependant, quand vous créez une propriété pour une dispinterface, vous ne pouvez spécifier le type de fonction ou le type des paramètres.

CoClasses

Une CoClasse décrit un objet COM unique qui implémente une ou plusieurs interfaces. Quand vous définissez une CoClasse, vous devez spécifier l'interface implémentée qui est celle par défaut pour l'objet et, de manière facultative, la dispinterface qui est la source par défaut des événements. Il n'est pas possible d'ajouter directement des propriétés ou méthodes à une CoClasse dans l'éditeur de bibliothèques de types. Les propriétés et méthodes sont exposées aux clients par les interfaces, qui sont associées à la CoClasse avec la page Implémente.

Définitions de types

Les énumérations, les alias, les enregistrements et les unions déclarent des types qui peuvent ensuite être utilisés ailleurs dans la bibliothèque de types.

Les énumérations sont constituées d'une liste de constantes numériques. Ces constantes sont généralement des valeurs entières au format décimal ou hexadécimal. Par défaut, la valeur de base est zéro. Vous pouvez ajouter des constantes à une énumération en la sélectionnant dans le volet liste des objets, puis en choisissant le bouton Constante de la barre d'outils ou en sélectionnant la commande Nouveau|Const dans le menu contextuel du volet liste des objets.

Remarque Il est fortement conseillé de spécifier une chaîne d'aide pour les énumérations. Voici un exemple d'entrée d'un type énumération pour un bouton de souris qui inclut une chaîne d'aide pour chacun des éléments de l'énumération :

```
mbLeft = 0 [helpstring 'mbLeft'];
mbRight = 1 [helpstring 'mbRight'];
mbMiddle = 3 [helpstring 'mbMiddle'];
```

Un alias définit un alias (une définition de type) pour un type. Un alias permet de définir des types s'utilisant dans d'autres types, par exemple des enregistrements ou des unions. Associez l'alias au type sous-jacent en spécifiant l'attribut Type dans la page Attributs.

Un enregistrement est constitué de la liste des membres de la structure, ou champs. Une union est un enregistrement ayant seulement une partie de type variant. Comme un enregistrement, une union est constituée de la liste des membres de la structure, ou champs. Mais, à la différence des membres d'un enregistrement, tous les membres d'une union occupent le même emplacement physique, il n'est donc possible de stocker qu'une seule valeur logique.

Ajoutez les champs à un enregistrement ou une union en le sélectionnant dans le volet liste des objets puis en choisissant le bouton Champ de la barre d'outils ou en utilisant le menu contextuel du volet liste des objets. Chaque champ dispose d'un nom et d'un type que vous pouvez spécifier en sélectionnant le champ et en attribuant les valeurs dans la page Attributs. Les enregistrements et les unions peuvent être définis avec un repère optionnel.

Les membres peuvent être de n'importe quel type prédéfini. Vous pouvez aussi spécifier un type en utilisant un alias avant de définir l'enregistrement.

Modules

Un module définit un groupe de fonctions, généralement un ensemble de points d'entrée de DLL. Vous définissez un module en :

- Spécifiant la DLL qu'il représente dans la page Attributs.
- Ajoutant des méthodes et des constantes en utilisant la barre d'outils ou le menu contextuel du volet liste des objets. Vous devez ensuite spécifier les attributs de chaque méthode ou constante en la sélectionnant dans le volet liste des objets, puis en définissant les valeurs dans la page Attributs.

Pour les méthodes de module, vous devez spécifier dans la page Attributs un nom et un point d'entrée dans la DLL. Déclarez les paramètres de la fonction et le type de valeur renvoyé dans la page Paramètres.

Pour les constantes de module, utilisez la page Attributs pour spécifier le nom, le type et la valeur.

Remarque L'éditeur de bibliothèques de types ne génère aucune déclaration ou implémentation associée à un module. La DLL spécifiée doit être créée dans un projet séparé.

Utilisation de l'éditeur de bibliothèques de types

En utilisant l'éditeur de bibliothèques de types, vous pouvez créer de nouvelles bibliothèques de types ou modifier des bibliothèques existantes. Généralement un concepteur d'application utilise l'expert pour créer les objets qui sont exposés dans la bibliothèques de types et laisse Delphi générer automatiquement la bibliothèque de types. La bibliothèque de types générée automatiquement est ensuite ouverte dans l'éditeur de bibliothèques de types afin de définir (ou modifier) les interfaces, d'ajouter des définitions de types, etc.

Cependant, même si vous n'utilisez pas l'expert pour définir les objets, vous pouvez utiliser l'éditeur de bibliothèques de types pour définir une nouvelle bibliothèque de types. Dans ce cas, vous devez créer vous-même les classes d'implémentation car l'éditeur de bibliothèques de types ne génère pas de code pour les CoClasses qui n'ont pas été associées par l'expert à une bibliothèque de types.

L'éditeur supporte un sous-ensemble de types autorisés et de SafeArray dans une bibliothèque de types comme décrit ci-dessous.

Vous pouvez effectuer les opérations suivantes dans une bibliothèque de types :

- Création d'une nouvelle bibliothèque de types
- Ouverture d'une bibliothèque de types existante
- Ajout d'une interface à une bibliothèque de types
- Modification d'une interface
- Ajout de propriétés et de méthodes à une bibliothèque de types
- Ajout d'une CoClasse à une bibliothèque de types
- Ajout d'une interface à une CoClasse
- Ajout d'une énumération à une bibliothèque de types
- Ajout d'un alias à une bibliothèque de types
- Ajout d'un enregistrement ou d'une union à une bibliothèque de types
- Ajout d'un module à une bibliothèque de types
- Enregistrement et recensement des informations d'une bibliothèque de types

Types autorisés

Dans l'éditeur de bibliothèques de types, vous utilisez des identificateurs de types différents, selon que vous travaillez en IDL ou en Pascal Objet. Spécifiez le langage que vous voulez utiliser dans la boîte de dialogue Options d'environnement.

Les types suivants sont autorisés dans une bibliothèque de types pour le développement COM. La colonne Compatible Automation spécifie si le type peut être utilisé par une interface dont l'indicateur Automation ou Dispinterface est activé. COM utilise automatiquement le marshalling pour les types suivants via la bibliothèque de types.

Type Pascal	Type IDL	Type variant	Compatible Automation	Description
Smallint	short	VT_I2	Oui	entier signé sur 2 octets
Integer	long	VT_I4	Oui	entier signé sur 4 octets
Single	single	VT_R4	Oui	Réel sur 4 octets
Double	double	VT_R8	Oui	Réel sur 8 octets
Currency	CURRENCY	VT_CY	Oui	Monétaire
TDateTime	DATE	VT_DATE	Oui	Date
WideString	BSTR	VT_BSTR	Oui	Chaîne binaire
IDispatch	IDispatch	VT_DISPATCH	Oui	Pointeur sur une interface IDispatch
SCODE	SCODE	VT_ERROR	Oui	Code d'erreur OLE
WordBool	VARIANT_BOOL	VT_BOOL	Oui	Vrai = -1, faux = 0
OleVariant	VARIANT	VT_VARIANT	Oui	Variant OLE
IUnknown	IUnknown	VT_UNKNOWN	Oui	Pointeur sur une interface IUnknown
Shortint	byte	VT_I1	Non	Entier signé sur 1 octet
Byte	unsigned char	VT_UI1	Oui	Entier non signé sur 1 octet
Word	unsigned short	VT_UI2	Oui*	Entier non signé sur 2 octets
LongWord	unsigned long	VT_UI4	Oui*	Entier non signé sur 4 octets

Type Pascal	Type IDL	Type variant	Compatible Automation	Description
Int64	__int64	VT_I8	Non	Entier signé sur 8 octets
Largeuint	uint64	VT_UI8	Non	Entier non signé sur 8 octets
SYSINT	int	VT_INT	Oui*	Entier dépendant du système (Win32=Integer)
SYSUINT	unsigned int	VT_UINT	Oui*	Entier non signé dépendant du système
HResult	HRESULT	VT_HRESULT	Non	Code d'erreur 32 bits
Pointer		VT_PTR -> VT_VOID	Non	Pointeur non typé
SafeArray	SAFEARRAY	VT_SAFEARRAY	Non	Tableau protégé OLE
PChar	LPSTR	VT_LPSTR	Non	Pointeur sur un Char
PWideChar	LPWSTR	VT_LPWSTR	Non	Pointeur sur un WideChar

* Word, LongWord, SYSINT et SYSUINT sont compatibles Automation dans la plupart des applications, mais ils peuvent ne pas l'être dans des anciennes applications.

Remarque Le type Byte (VT_UI1) est compatible avec l'Automation, mais il n'est pas autorisé dans un Variant ou un OleVariant car de nombreux serveurs Automation ne gèrent pas correctement cette valeur.

Outre ces types, toute interface ou type défini dans la bibliothèque ou dans les bibliothèques référencées peut s'utiliser dans une définition de bibliothèque de types.

L'éditeur de bibliothèques de types stocke les informations de type dans le fichier (.TLB) de bibliothèque de types générée sous forme binaire.

Si un paramètre est de type Pointer, l'éditeur de bibliothèques de types le convertit généralement en paramètre variable. Quand la bibliothèque de types est enregistrée, les indicateurs IDL des ElemDesc associés aux paramètres variable sont marqués IDL_FIN ou IDL_FOUT.

Souvent, les indicateurs IDL ElemDesc ne sont pas marqués par IDL_FIN ni par IDL_FOUT lorsque le type est précédé d'un pointeur. De même, s'il s'agit de dispinterfaces, les indicateurs IDL ne sont généralement pas utilisés. Dans ces situations, on peut voir à côté de l'identificateur de variable un commentaire comme {IDL_None} ou {IDL_In}. Ces commentaires sont utilisés lors de l'enregistrement d'une bibliothèque de types, pour marquer correctement les indicateurs IDL.

Les SafeArray

COM requiert que les tableaux soient transmis via un type de données spécial appelé SafeArray. Vous pouvez créer et détruire les SafeArrays en appelant des fonctions COM particulières, et tous les éléments d'un SafeArray doivent être de types compatibles automation-valides. Le compilateur Delphi dispose d'informations intégrées sur les SafeArrays COM et appellera automatiquement l'API COM pour créer, copier et détruire des SafeArrays.

Dans l'éditeur de bibliothèques de types, un *SafeArray* doit spécifier son type de composant. Par exemple, la ligne suivante de la page Texte déclare une méthode avec un paramètre *SafeArray* avec un élément de type Integer :

```
procedure HighLightLines(Lines: SafeArray of Integer);R
```

Remarque Même si vous devez spécifier le type de l'élément lors de la déclaration d'un type *SafeArray* dans l'éditeur de bibliothèques de types, la déclaration du fichier unité _TLB généré n'indique pas le type de l'élément.

Utilisation de la syntaxe Pascal Objet ou IDL

La page Texte de l'éditeur de bibliothèques de types peut afficher vos informations de type de deux manière différentes :

- En utilisant une extension de la syntaxe Pascal Objet.
- En utilisant le langage Microsoft IDL.

Remarque Si vous travaillez sur un objet CORBA, vous ne devez utiliser aucune de ces syntaxes mais employer à la place l'IDL CORBA.

Pour sélectionner le langage utilisé, il vous suffit de modifier un paramètre de la boîte de dialogue Options d'environnement. Choisissez Outils | Options d'environnement, et spécifiez Pascal ou IDL comme langage de l'éditeur dans la page Bibliothèques de types de la boîte de dialogue.

Remarque Le choix de la syntaxe Pascal Objet ou IDL affecte également les choix disponibles dans la page Attributs des paramètres.

Comme c'est généralement le cas pour les applications Pascal Objet, les identificateurs des bibliothèques de types ne distinguent pas les majuscules des minuscules. Ils peuvent avoir jusqu'à 255 caractères et doivent commencer par une lettre ou un caractère de soulignement (_).

Spécifications des attributs

Le Pascal Objet a été étendu pour que les bibliothèques de types puissent contenir des spécifications d'attributs. Les spécifications d'attributs sont entourées de crochets droits et sont séparées par des virgules. Chaque spécification d'attribut est constituée d'un nom d'attribut suivi (le cas échéant) d'une valeur.

Le tableau suivant est la liste des noms d'attributs et des valeurs correspondantes.

Tableau 34.3 Syntaxe des attributs

Nom d'attribut	Exemple	S'applique à
aggregatable	[aggregatable]	typeinfo
appobject	[appobject]	typeinfo d'une CoClasse
bindable	[bindable]	membres, sauf membres d'une CoClasse
control	[control]	bibliothèque de types, typeinfo
custom	[custom '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}' 0]	tout
default	[default]	membres d'une CoClasse
defaultbind	[defaultbind]	membres, sauf membres d'une CoClasse
defaultcollection	[defaultcollection]	membres, sauf membres d'une CoClasse
defaultvtbl	[defaultvtbl]	membres d'une CoClasse
dispid	[dispid]	membres, sauf membres d'une CoClasse
displaybind	[displaybind]	membres, sauf membres d'une CoClasse
dllname	[dllname 'Helper.dll']	typeinfo d'un module
dual	[dual]	typeinfo d'une interface
helpfile	[helpfile 'c:\help\myhelp.hlp']	bibliothèque de types
helpstringdll	[helpstringdll 'c:\help\myhelp.dll']	bibliothèque de types
helpcontext	[helpcontext 2005]	tout sauf membres et paramètres d'une CoClasse
helpstring	[helpstring 'Interface Paye']	tout sauf membres et paramètres d'une CoClasse
helpstringcontext	[helpstringcontext $17]	tout sauf membres et paramètres d'une CoClasse
hidden	[hidden]	tout sauf des paramètres
immediatebind	[immediatebind]	membres, sauf membres d'une CoClasse
lcid	[lcid $324]	bibliothèque de types
licensed	[licensed]	bibliothèque de types, typeinfo d'une CoClasse
nonbrowsable	[nonbrowsable]	membres, sauf membres d'une CoClasse
nonextensible	[nonextensible]	typeinfo d'une interface
oleautomation	[oleautomation]	typeinfo d'une interface
predeclid	[predeclid]	typeinfo
propget	[propget]	membres, sauf membres d'une CoClasse

Tableau 34.3 Syntaxe des attributs (suite)

Nom d'attribut	Exemple	S'applique à
propput	[propput]	membres, sauf membres d'une CoClasse
propputref	[propputref]	membres, sauf membres d'une CoClasse
public	[public]	typeinfo d'un alias
readonly	[readonly]	membres, sauf membres d'une CoClasse
replaceable	[replaceable]	tout sauf membres et paramètres d'une CoClasse
requestedit	[requestedit]	membres, sauf membres d'une CoClasse
restricted	[restricted]	tout sauf des paramètres
source	[source]	tous les membres
uidefault	[uidefault]	membres, sauf membres d'une CoClasse
usesgetlasterror	[usesgetlasterror]	membres, sauf membres d'une CoClasse
uuid	[uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}']	bibliothèque de types, typeinfo (obligatoire)
vararg	[vararg]	membres, sauf membres d'une CoClasse
version	[version 1.1]	bibliothèque de types, typeinfo

Syntaxe pour une interface

La syntaxe Pascal Objet pour déclarer les informations de type d'une interface est de la forme :

```
nominterface = interface[(baseinterface)] [attributs]
listefonctions
[listeméthodepropriété]
end;
```

Par exemple, le texte suivant est la déclaration d'une interface avec deux méthodes et une propriété :

```
Interface1 = interface (IDispatch)
  [uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}', version 1.0]
  function Calculate(optional seed:Integer=0): Integer;
  procedure Reset;
  procedure PutRange(Range: Integer) [propput, dispid $00000005]; stdcall;
  function GetRange: Integer;[propget, dispid $00000005]; stdcall;
end;
```

La syntaxe Microsoft IDL correspondante est :

```
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',version 1.0]
interface Interface1 :IDispatch
{
```

```
  long Calculate([in, optional, defaultvalue(0)] long seed);
  void Reset(void);
  [propput, id(0x00000005)] void _stdcall PutRange([in] long Value);
  [propput, id(0x00000005)] void _stdcall getRange([out, retval] long *Value);
};
```

Syntaxe pour une interface de répartition

La syntaxe Pascal Objet pour déclarer les informations de type d'une dispinterface est de la forme :

```
nomdispinterface = dispinterface [attributs]
listefonctions
[listepropriétés]
end;
```

Par exemple, le texte suivant est la déclaration d'une dispinterface avec les mêmes méthodes et la même propriété que pour l'interface précédente :

```
MyDispObj = dispinterface
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
  version 1.0,
  helpstring 'dispatch interface for MyObj']
  function Calculate(seed:Integer): Integer [dispid 1];
  procedure Reset [dispid 2];
  property Range: Integer [dispid 3];
end;
```

La syntaxe Microsoft IDL correspondante est :

```
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
  version 1.0,
  helpstring "dispatch interface for MyObj"]
dispinterface Interface1
{
  methods:
    [id(1)] int Calculate([in] int seed);
    [id(2)] void Reset(void);
  properties:
    [id(3)] int Value;
};
```

Syntaxe pour une CoClasse

La syntaxe Pascal Objet pour déclarer les informations de type d'une CoClasse est de la forme :

```
nomclasse = coclass(nominterface[attributsinterface], ...); [attributs];
```

Par exemple, le texte suivant est la déclaration d'une coclasse pour l'interface *IMyInt* et la dispinterface *DmyInt* :

```
myapp = coclass(IMyInt [source], DMyInt);
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
  version 1.0,
  helpstring 'A class',
  appobject]
```

La syntaxe Microsoft IDL correspondante est :

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
  version 1.0,
  helpstring 'A class',
  appobject]
coclass myapp
{
  methods:
  [source] interface IMyInt);
  dispinterface DMyInt;
};
```

Syntaxe pour une énumération

La syntaxe Pascal Objet pour déclarer les informations de type d'une énumération est de la forme :

```
noménumération = ([attributs] listeénumération);
```

Par exemple, le texte suivant est la déclaration d'un type énuméré avec trois valeurs :

```
location = ([uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
         helpstring 'emplacement de la cabine']
  Inside = 1 [helpstring 'Dans le pavillon'];
  Outside = 2 [helpstring 'A l''extérieur du pavillon'];
  Offsite = 3 [helpstring 'Loin du pavillon'];);
```

La syntaxe Microsoft IDL correspondante est :

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
       helpstring 'emplacement de la cabine']
typedef enum
{
  [helpstring 'Dans le pavillon'] Inside = 1,
  [helpstring 'A l''extérieur du pavillon'] Outside = 2,
  [helpstring 'Loin du pavillon'] Offsite = 3
} location;
```

Syntaxe pour un alias

La syntaxe Pascal Objet pour déclarer les informations de type d'un alias est de la forme :

```
nomalias = typebase[attributs];
```

Par exemple, le texte suivant est la déclaration d'un DWORD comme alias d'un entier :

```
DWORD = Integer [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'];
```

La syntaxe Microsoft IDL correspondante est :

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'] typedef long DWORD;
```

Syntaxe pour un enregistrement

La syntaxe Pascal Objet pour déclarer les informations de type d'un enregistrement est de la forme :

```
nomenregistrement = record [attributs] listechamps end;
```

Par exemple, le texte suivant est la déclaration d'un enregistrement :

```
Tasks = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
                helpstring 'Description de la tâche']
  ID: Integer;
  StartDate: TDate;
  EndDate: TDate;
  Ownername: WideString;
  Subtasks: safearray of Integer;
end;
```

La syntaxe Microsoft IDL correspondante est :

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
       helpstring 'Description de la tâche']
typedef struct
{
  long ID;
  DATE StartDate;
  DATE EndDate;
  BSTR Ownername;
  SAFEARRAY (int) Subtasks;
} Tasks;
```

Syntaxe pour une union

La syntaxe Pascal Objet pour déclarer les informations de type d'une union est de la forme :

```
nomunion = record [attributs]
case Integer of
  0: field1;
  1: field2;
  ...
end;
```

Par exemple, le texte suivant est la déclaration d'une union :

```
MyUnion = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
                  helpstring 'description d''un élément']
case Integer of
  0: (Name: WideString);
  1: (ID: Integer);
  3: (Value: Double);
end;
```

La syntaxe Microsoft IDL correspondante est :

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
       helpstring 'description d''un élément']
typedef union
{
```

```
  BSTR Name;
  long ID;
  double Value;
  } MyUnion;
```

Syntaxe pour un module

La syntaxe Pascal Objet pour déclarer les informations de type d'un module est de la forme :

```
nommodule = module constantes pointsentrée end;
```

Par exemple, le texte suivant est la déclaration des informations de type d'un module :

```
MyModule = module [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
                   dllname 'circle.dll']
  PI: Double = 3.14159;
  function area(radius: Double): Double [ entry 1 ]; stdcall;
  function circumference(radius: Double): Double [ entry 2 ]; stdcall;
end;
```

La syntaxe Microsoft IDL correspondante est :

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
       dllname("circle.dll")]
module MyModule
{
  double PI = 3.14159;
  [entry(1)] double _stdcall area([in] double radius);
  [entry(2)] double _stdcall circumference([in] double radius);
};
```

Création d'une nouvelle bibliothèque de types

Vous pouvez souhaiter créer une bibliothèque de types qui soit indépendante d'un objet COM particulier. Vous pouvez par exemple définir une bibliothèque de types contenant des définitions de types que vous utilisez dans d'autres bibliothèques de types. Vous pouvez, pour ce faire, créer une bibliothèque de types contenant les définitions de base et l'ajouter dans la page Utilise des autres bibliothèques.

Vous pouvez également créer une bibliothèque de types pour un objet qui n'a pas encore été implémenté. Quand la bibliothèque de types contient les définitions d'interface, vous pouvez utiliser l'expert objet COM pour générer une CoClasse et l'implémentation.

Pour créer une nouvelle bibliothèque de types,

1. Choisissez Fichier | Nouveau | Autre afin d'ouvrir la boîte de dialogue Nouveaux éléments.
2. Choisissez la page ActiveX.
3. Sélectionnez l'icône Bibliothèque de types.
4. Choisissez OK.

L'éditeur de bibliothèques de types s'affiche en demandant le nom de la bibliothèque de types.

5 Entrez le nom de la bibliothèque de types. Poursuivez en ajoutant les éléments de la bibliothèque de types.

Ouverture d'une bibliothèque de types existante

Si vous utilisez les experts pour créer un contrôle ActiveX, un objet Automation, une fiche active, un objet Active Server, un objet COM, un objet transactionnel, un module de données distant ou un module de données transactionnel, une bibliothèque de types est automatiquement créée avec une unité d'implémentation. De plus, des bibliothèques de types sont associées à d'autres logiciels (serveurs) installés sur votre système.

Pour ouvrir une bibliothèque de types existante indépendant d'un projet,

1 Choisissez Fichier | Ouvrir dans le menu principal de l'EDI.

2 Dans la boîte de dialogue Ouvrir, initialisez la liste Type à Bibliothèques de types.

3 Sélectionnez la bibliothèque de types souhaitée et choisissez Ouvrir.

Pour ouvrir une bibliothèque de types associée au projet en cours,

1 Choisissez Voir | Bibliothèque de types

Remarque Si vous utilisez l'expert objet CORBA, vous pouvez également utiliser Voir | Bibliothèque de types pour modifier les interfaces d'objet CORBA. Ce que vous voyez alors n'est pas à proprement dit une bibliothèque de types, mais vous l'utilisez de la même manière.

Vous pouvez maintenant ajouter des interfaces, CoClasses et d'autres éléments à la bibliothèque de types (énumérations, propriétés ou méthodes, etc.).

Remarque Les modifications effectuées aux informations de bibliothèque de types à l'aide de l'éditeur de bibliothèque de types peuvent se refléter automatiquement dans la classe d'implémentation associée. Si vous préférez au préalable revoir les modifications, vérifiez que la boîte de dialogue est activée. Elle est activée par défaut et ce paramétrage peut être modifié avec l'option d'affichage des mises à jour avant le rafraîchissement présente sur la page Outils | Options d'environnement | Bibliothèque de types. Pour plus d'informations, voir "Boîte de dialogue Appliquer les mises à jour" à la page 34-28.

Astuce Quand vous écrivez des applications client, il n'est pas nécessaire d'ouvrir la bibliothèque de types. Vous avez juste besoin de l'unité *Projet*_TLB créée par l'éditeur de bibliothèques de types et pas de la bibliothèque de types même. Il est possible d'ajouter directement ce fichier dans un projet client ou, si la bibliothèque de types est recensée sur votre système, vous pouvez utiliser la boîte de dialogue Importation de bibliothèque de types (Projet | Importation de bibliothèque de types).

Ajout d'une interface à une bibliothèque de types

Pour ajouter une interface,

1 Dans la barre d'outils, cliquez sur l'icône Interface.

Une interface est ajoutée dans le volet liste des objets et vous pouvez en saisir le nom.

2 Entrez le nom de l'interface.

La nouvelle interface a les attributs par défaut que vous pouvez modifier selon vos besoins.

Vous pouvez ajouter des propriétés (représentées par des fonctions get/set) et des méthodes afin de définir le rôle de l'interface.

Modification d'une interface en utilisant la bibliothèque de types

Il y a plusieurs manières de modifier une interface ou une dispinterface une fois créée.

- Vous pouvez modifier les attributs de l'interface en utilisant la page des informations de type contenant les informations que vous voulez modifier. Sélectionnez l'interface dans le volet liste des objets, puis utilisez les contrôles de la page appropriée des informations de type. Par exemple, vous pouvez changer l'interface parent en utilisant la page Attributs ou utiliser la page Indicateurs pour spécifier si c'est ou non une interface double.
- Vous pouvez modifier directement la déclaration de l'interface en sélectionnant l'interface dans le volet liste des objets et en modifiant les déclarations de la page Texte.
- Vous pouvez ajouter des propriétés et méthodes à l'interface (voir la section suivante).
- Vous pouvez modifier les propriétés et méthodes déjà définies dans l'interface en modifiant leurs informations de type.
- Vous pouvez l'associer à une CoClasse en sélectionnant la CoClasse dans le volet liste des objets puis en cliquant avec le bouton droit de la souris dans la page Implémente et en choisissant Insertion d'interface.

Remarque Si vous utilisez la bibliothèque de types pour ajouter une interface CORBA, la plupart des informations de la page Attributs sont inapplicables. Vous n'avez pas non plus besoin de la page Indicateurs.

Si l'interface est associée à une CoClasse générée par un expert, vous pouvez demander à l'éditeur de bibliothèques de types d'appliquer les modifications au fichier implémentation en utilisant le bouton Rafraîchir l'implémentation de la barre d'outils.Si la boîte de dialogue Appliquer les mises à jour est activée, l'éditeur de bibliothèques de types vous prévient avant de mettre à jour les sources lorsque vous enregistrez la bibliothèque de types et vous avertit de problèmes éventuels. Par exemple, si vous renommez une interface d'événement

par erreur, vous obtenez dans votre fichier source un avertissement du type suivant :

```
En raison de la présence de variables d'instance dans votre fichier d'implémentation,
Delphi n'a pas pu mettre à jour le fichier pour refléter la modification du nom de
l'interface d'événement. Bien que Delphi ait mis à jour la
bibliothèque de types, vous devez mettre à jour le fichier d'implémentation manuellement.
```

Un commentaire TODO précède cet avertissement dans le fichier source.

Attention Si vous ignorez cet avertissement et le commentaire TODO, le code ne se compile pas.

Ajout de propriétés et méthodes à une interface ou dispinterface

Pour ajouter des membres à une interface ou une dispinterface

1. Sélectionnez l'interface et choisissez l'icône Propriété ou Méthode dans la barre d'outils. Si vous ajoutez une propriété, vous pouvez cliquer directement dans l'icône Propriété pour créer une propriété en lecture/écriture (avec des fonctions get et set), ou cliquez sur la flèche vers le bas pour afficher un menu des types de propriétés.

   Les membres méthode d'accès à la propriété ou le membre méthode sont ajoutés au volet liste des objets et vous pouvez en saisir le nom.

2. Saisissez le nom du membre.

Le nouveau membre contient les valeurs par défaut pour les pages Attributs, Paramètres et Indicateurs ; vous pouvez les modifier pour les adapter à vos besoins. Par exemple, dans le cas d'une propriété, vous utiliserez la page Attributs pour spécifier le type. Si vous ajoutez une méthode, vous utiliserez la page Paramètres pour en spécifier les paramètres.

Vous pouvez également ajouter des méthodes et propriétés directement dans la page Texte à l'aide de la syntaxe Pascal ou IDL. Par exemple, si vous utilisez la syntaxe Pascal, vous pouvez saisir les déclarations de propriétés suivantes dans la page Texte d'une interface :

```
Interface1 = interface(IDispatch)
  [ uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
    version 1.0,
    dual,
    oleautomation ]
 function AutoSelect: Integer [propget, dispid $00000002]; safecall; // Ajouter ceci
 function AutoSize: WordBool [propget, dispid $00000001]; safecall; // et ceci
 procedure AutoSize(Value: WordBool) [propput, dispid $00000001]; safecall; // Et ceci
end;
```

Si vous utilisez IDL, vous pouvez ajouter les mêmes déclarations de la manière suivante :

```
[
  uuid(5FD36EEF-70E5-11D1-AA62-00C04FB16F42),
  version(1.0),
  dual,
  oleautomation
```

```
]
interface Interface1: IDispatch
{ // Ajoutez tout ce qui se trouve entre les accolades
[propget, id(0x00000002)]
  HRESULT _stdcall AutoSelect([out, retval] long Value );
  [propget, id(0x00000003)]
  HRESULT _stdcall AutoSize([out, retval] VARIANT_BOOL Value );
  [propput, id(0x00000003)]
  HRESULT _stdcall AutoSize([in] VARIANT_BOOL Value );
};
```

Une fois des membres ajoutés à une interface en utilisant la page Texte de l'interface, les membres apparaissent comme éléments distincts dans le volet liste des objets. Chaque membre dispose de ses propres pages Attributs, Indicateurs et Paramètres. Vous pouvez modifier chaque nouvelle propriété ou méthode en la sélectionnant dans le volet liste des objets et en utilisant ses pages ou en effectuant les modifications directement dans la page Texte.

Si l'interface est associée à une CoClasse générée par un expert, vous pouvez demander à l'éditeur de bibliothèques de types d'appliquer les modifications au fichier implémentation en utilisant le bouton Rafraîchir de la barre d'outils. L'éditeur de bibliothèques de types ajoute les nouvelles méthodes à la classe d'implémentation afin de refléter les nouveaux membres. Vous pouvez ensuite rechercher les nouvelles méthodes dans le code source de l'unité d'implémentation et remplir leurs corps afin de compléter l'implémentation.

Si la boîte de dialogue Appliquer les mises à jour est activée, l'éditeur de bibliothèques de types vous prévient avant de mettre à jour les sources lorsque vous enregistrez la bibliothèque de types et vous avertit de problèmes éventuels.

Ajout d'une CoClasse à une bibliothèque de types

Le moyen le plus simple d'ajouter une CoClasse à un projet consiste à choisir Fichier|Nouveau|Autre dans le menu principal de l'EDI et d'utiliser l'expert approprié dans les pages ActiveX ou Multi-niveaux de la boîte de dialogue Nouveaux éléments. Cette approche est pratique car non seulement la CoClasse et son interface sont ajoutées à la bibliothèque de types, mais l'expert ajoute une unité d'implémentation et actualise le fichier projet pour inclure la nouvelle unité d'implémentation dans sa clause uses.

Si vous n'utilisez pas un expert, vous pouvez tout de même créer une CoClasse en choisissant l'icône CoClasse dans la barre d'outils puis en spécifier ses attributs. Généralement, vous spécifierez son nom (dans la page Attributs) et utiliserez la page Indicateurs pour spécifier si la CoClasse est un objet application, si elle représente un contrôle ActiveX, etc.

Remarque Si vous ajoutez une CoClasse à une bibliothèque de types en utilisant la barre d'outils au lieu de l'expert, vous devez générer vous-même l'implémentation pour la CoClasse et l'actualiser manuellement à chaque fois que vous modifiez un élément de l'une des interfaces de la CoClasse. Il n'est pas possible d'ajouter directement des membres à une CoClasse. C'est implicitement que vous ajoutez des membres quand vous ajoutez une interface à la CoClasse.

Ajout d'une interface à une CoClasse

Les CoClasses sont définies par les interfaces qu'elles présentent aux clients. Même si vous pouvez ajouter un nombre quelconque de propriétés et de méthodes à la classe d'implémentation d'une CoClasse, les clients ne voient que les propriétés et méthodes qui sont exposées par les interfaces associées à la CoClasse.

Pour associer une interface à une CoClasse, cliquez avec le bouton droit de la souris dans la page Implémente de la classe et choisissez Insérer une interface pour afficher une liste des interfaces dans laquelle choisir. Cette liste contient les interfaces définies dans la bibliothèque de types en cours et celles définies dans toutes les bibliothèques de types référencées par celle-ci. Choisissez une interface que la classe doit implémenter. L'interface est ajoutée à la page avec son GUID et d'autres attributs.

Si la CoClasse a été générée par un expert, l'éditeur de bibliothèques de types actualise automatiquement la classe d'implémentation pour y placer le squelette des méthodes (y compris les méthodes d'accès aux propriétés) pour toutes les interfaces ajoutées de cette manière.Si la boîte de dialogue Appliquer les mises à jour est activée, l'éditeur de bibliothèques de types vous prévient avant de mettre à jour les sources lorsque vous enregistrez la bibliothèque de types et vous avertit de problèmes éventuels.

Ajout d'une énumération à une bibliothèque de types

Pour ajouter une énumération à une bibliothèque de types,

1 Dans la barre d'outils, choisissez l'icône Enumération.

   Un type énumération dont vous pouvez saisir le nom est ajouté dans le volet liste des objets.

2 Saisissez le nom du membre.

   La nouvelle énumération est vide et sa page Attributs contient les attributs par défaut que vous pouvez modifier.

Ajoutez des valeurs à l'énumération en cliquant sur le bouton Nouvelle constante. Sélectionnez ensuite chaque valeur énumérée et assignez-lui un nom (et éventuellement une valeur) à l'aide de la page Attributs.

Une fois une énumération ajoutée, le nouveau type peut être utilisé par la bibliothèque de types ou par toute bibliothèque de types qui y fait référence dans sa page Utilise. Vous pouvez, par exemple, utiliser l'énumération comme type d'une propriété ou d'un paramètre.

Ajout d'un alias à une bibliothèque de types

Pour ajouter un alias à une bibliothèque de types :

1 Dans la barre d'outils, choisissez l'icône Alias.

   Un type alias dont vous pouvez saisir le nom est ajouté dans le volet liste des objets.

2 Saisissez le nom de l'alias.

   Par défaut, le nouvel alias correspond au type Integer. Utilisez la page Attributs pour choisir à la place le type que l'alias doit représenter.

Une fois un alias ajouté, le nouveau type peut être utilisé par la bibliothèque de types ou par toute bibliothèque de types qui y fait référence dans sa page Utilise. Vous pouvez, par exemple, utiliser l'alias comme type d'une propriété ou d'un paramètre.

Ajout d'un enregistrement ou d'une union à une bibliothèque de types

Pour ajouter un enregistrement ou une union à une bibliothèque de types :

1 Dans la barre d'outils, choisissez le bouton Enregistrement ou le bouton Union.

   Le type d'élément choisi dont vous pouvez saisir le nom est ajouté dans le volet liste des objets.

2 Saisissez le nom de l'enregistrement ou de l'union.

   A ce stade, l'enregistrement ou l'union ne contient pas de champ.

3 Ayant sélectionné l'enregistrement ou l'union dans le volet liste des objets, choisissez le bouton Champ dans la barre d'outils. Spécifiez le nom et le type du champ en utilisant la page Attributs.

4 Répétez l'étape 3 pour chaque champ nécessaire.

Une fois l'enregistrement ou l'union défini, le nouveau type peut être utilisé par la bibliothèque de types ou par toute bibliothèque de types qui y fait référence dans sa page Utilise. Vous pouvez, par exemple, utiliser l'enregistrement ou l'union comme type d'une propriété ou d'un paramètre.

Ajout d'un module à une bibliothèque de types

Pour ajouter un module à une bibliothèque de types :

1 Dans la barre d'outils, choisissez l'icône Module.

   Le module sélectionné dont vous pouvez saisir le nom est ajouté dans le volet liste des objets.

2 Saisissez le nom du module

3 Dans la page Attributs, spécifiez le nom de la DLL dont le module représente les points d'entrée.

4 Ajoutez des méthodes de la DLL spécifiée à l'étape 3 en choisissant le bouton Méthode dans la barre d'outils puis en utilisant la page Attributs pour décrire la méthode.

5 Ajoutez les constantes que le module doit définir en choisissant le bouton Constante de la barre d'outils. Pour chaque constante, spécifiez le nom, le type et une valeur.

Enregistrement et recensement des informations d'une bibliothèque de types

Après avoir modifié une bibliothèque de types, vous devez enregistrer et recenser les informations de la bibliothèque de types.

L'enregistrement de la bibliothèque de types actualise :

- Le fichier bibliothèque de types binaire (d'extension .tlb).
- Le fichier unité *Projet*_TLB qui représente son contenu.
- Le code d'implémentation de toutes les CoClasses générées par un expert.

Remarque La bibliothèque de types est stockée dans un fichier binaire distinct (.TLB), mais elle est également liée au serveur (.EXE, DLL ou .OCX).

Remarque Si vous utilisez l'éditeur de bibliothèques de types pour des interfaces CORBA, l'unité *Projet*_TLB.pas définit les objets stub et squelette nécessaires à l'application CORBA.

L'éditeur de bibliothèques de types vous propose plusieurs options pour stocker les informations de la bibliothèque de types. L'option utilisée dépend de l'étape en cours de l'implémentation de la bibliothèque de types :

- Enregistrer pour enregistrer le fichier .TLB et l'unité *Projet*_TLB sur disque.
- Rafraîchir pour actualiser uniquement en mémoire les unités de la bibliothèque de types.
- Recenser pour ajouter une entrée au registre Windows de votre système pour la bibliothèque de types. Cela est effectué automatiquement lors du recensement du serveur auquel le fichier .TLB est associé.
- Exporter pour enregistrer un fichier .IDL contenant les définitions de types et d'interfaces avec la syntaxe IDL.

Toutes les méthodes décrites ci-dessus effectuent une vérification de la syntaxe. Quand vous rafraîchissez, recensez ou enregistrez la bibliothèque de types, Delphi actualise automatiquement l'unité d'implémentation de toutes les CoClasses créées en utilisant un expert. Vous pouvez, de manière facultative, vérifier les modifications avant leur application, si l'option Appliquer les modifications de l'éditeur de bibliothèques de types est activée.

Boîte de dialogue Appliquer les mises à jour

La boîte de dialogue Appliquer les mises à jour apparaît lorsque vous actualisez, recensez ou enregistrez la bibliothèque de types si vous avez sélectionné l'option d'affichage des mises à jour avant le rafraichissement dans la page Outils | Options d'environnement | Bibliothèque de types (option activée par défaut).

Si cette option n'est pas activée, l'éditeur de bibliothèques de types met automatiquement à jour les sources de l'objet associé lorsque vous apportez des modifications dans l'éditeur. Si elle est activée, vous pouvez vous opposer aux modifications proposées lorsque vous actualisez, enregistrez ou recensez la bibliothèque de types.

La boîte de dialogue Appliquer les mises à jour vous avertit d'erreurs éventuelles et insère des commentaires TODO dans le fichier source. Par

exemple, si vous renommez un événement par erreur, vous obtenez dans votre fichier source un avertissement du type suivant :

```
En raison de la présence de variables d'instance dans votre fichier d'implémentation,
Delphi n'a pas pu mettre à jour le fichier pour refléter la modification du nom de
l'interface d'événement. Bien que Delphi ait mis à jour la
bibliothèque de types, vous devez mettre à jour le fichier d'implémentation manuellement.
```

Un commentaire TODO précède cet avertissement dans le fichier source.

Remarque Si vous ignorez cet avertissement et le commentaire TODO, le code n'est pas compilé.

Enregistrement d'une bibliothèque de types

L'enregistrement d'une bibliothèque de types effectue les actions suivantes :

- Vérification de la syntaxe.
- Enregistrement des informations dans un fichier .TLB.
- Enregistrement des informations dans une unité *Projet*_TLB.
- Demande au gestionnaire de module de l'EDI d'actualiser l'implémentation si la bibliothèque de types est associée à une CoClasse générée par un expert.

Pour enregistrer la bibliothèque de types, choisissez Fichier | Enregistrer dans le menu principal de Delphi.

Rafraîchissement de la bibliothèque de types

Le rafraîchissement d'une bibliothèque de types effectue les actions suivantes :

- Vérification de la syntaxe.
- Génération, uniquement en mémoire, des unités Delphi de la bibliothèque de types. L'unité n'est pas enregistrée sur disque.
- Demande au gestionnaire de module de l'EDI d'actualiser l'implémentation si la bibliothèque de types est associée à une CoClasse générée par un expert.

Pour rafraîchir la bibliothèque de types, choisissez l'icône Rafraîchir dans la barre d'outils de l'éditeur de bibliothèques de types.

Remarque Si vous avez renommé ou supprimé des éléments de la bibliothèque de types, le rafraîchissement de l'implémentation peut créer des entrées en double. Dans ce cas, vous devez déplacer le code pour corriger l'entrée et supprimer les doublons. De même, si vous supprimez des éléments de la bibliothèque de types, le rafraîchissement de l'implémentation ne les retire pas des CoClasses (partant du principe que vous voulez juste les rendre invisibles aux clients). Vous devez supprimer ces éléments manuellement dans l'unité d'implémentation quand ils ne sont plus nécessaires.

Recensement d'une bibliothèque de types

Il n'est généralement pas nécessaire de recenser explicitement une bibliothèque de types car elle est automatiquement recensée quand vous recensez votre application serveur COM (voir "Recensement d'un objet COM" à la page 36-18). Cependant, si vous créez une bibliothèque de types en utilisant l'expert

bibliothèque de types, elle n'est pas associée à un objet serveur. Dans ce cas, vous pouvez la recenser directement en utilisant la barre d'outils.

Le recensement d'une bibliothèque de types effectue les actions suivantes :

- Vérification de la syntaxe.
- Ajout aux registres Windows d'une entrée pour la bibliothèque de types.

Pour recenser la bibliothèque de types, choisissez l'icône Recenser de la barre d'outils de l'éditeur de bibliothèques de types.

Exportation d'un fichier IDL

L'exportation d'une bibliothèque de types effectue les actions suivantes :

- Vérification de la syntaxe.
- Création d'un fichier IDL contenant les informations de déclaration de type. Ce fichier décrit les informations de type avec le langage IDL CORBA ou en IDL Microsoft.

Pour exporter une bibliothèque de types, choisissez l'icône Exporter dans la barre d'outils de l'éditeur de bibliothèques de types.

Déploiement des bibliothèques de types

Par défaut, quand une bibliothèque de types a été créée comme partie d'un projet de serveur ActiveX ou Automation, elle est automatiquement liée comme ressource dans le fichier .DLL, .OCX ou EXE.

Vous pouvez cependant, si vous le préférez, déployer votre application avec la bibliothèque de types sous la forme d'un fichier .TLB distinct, car Delphi gère la bibliothèque de types

Historiquement, les bibliothèques de types des applications Automation étaient stockées dans un fichier distinct d'extension .TLB. Actuellement, dans les applications Automation standard, les bibliothèques de types sont compilées directement dans le fichier .OCX ou .EXE. Le système d'exploitation suppose que la bibliothèque de types est la première ressource dans le fichier exécutable (.OCX ou .EXE).

Quand vous souhaitez proposer à d'autres développeurs d'applications l'accès à la bibliothèque de types, elle peut se présenter sous les formes suivantes :

- Une ressource. Cette ressource doit avoir le type TYPELIB et un identificateur entier. Si vous choisissez de générer une bibliothèque de types avec un compilateur de ressources, elle doit être déclarée de la manière suivante dans le fichier ressource (.RC) :

  ```
  1 typelib mylib1.tlb
  2 typelib mylib2.tlb
  ```

  Il peut y avoir plusieurs ressources bibliothèque de types dans une bibliothèque ActiveX. Les développeurs d'applications utilisent un compilateur

de ressource pour ajouter le fichier .TLB à leurs propres bibliothèques ActiveX.

- Un fichier binaire autonome. Le fichier .TLB généré par l'éditeur de bibliothèques de types est un fichier binaire.

Chapitre

35

Création de clients COM

Les clients COM sont des applications qui utilisent un objet COM implémenté par une autre application ou bibliothèque. Les types les plus courants sont les applications qui contrôlent des serveurs Automation (ce sont les contrôleurs Automation) et les applications qui accueillent un contrôle ActiveX (ce sont les conteneurs ActiveX).

Au premier abord, ces deux types de clients COM semblent très différents : un contrôleur Automation standard lance un serveur EXE externe et émet des commandes que le serveur traite pour lui. Le serveur Automation est généralement non visuel et hors processus. D'un autre côté, le client ActiveX standard accueille un contrôle visuel en l'utilisant comme vous pouvez utiliser les contrôles de la palette des composants. Les serveurs ActiveX sont toujours des serveurs en processus.

Cependant, la réalisation de ces deux types de clients COM est étonnamment similaire : l'application client obtient une interface pour l'objet serveur et en utilise les propriétés et méthodes. Delphi simplifie beaucoup cela en vous permettant d'envelopper la CoClasse serveur dans un composant du client que vous pouvez même installer dans la palette des composants. Des exemples de tels composants enveloppe apparaissent dans deux pages de la palette des composants : des enveloppes ActiveX exemple apparaissent dans la page ActiveX et des objets Automation exemple apparaissent dans la page Serveurs.

Quand vous écrivez un client COM, vous devez comprendre l'interface que le serveur expose aux clients, exactement comme vous devez comprendre les propriétés et méthodes d'un composant de la palette des composants pour pouvoir l'utiliser dans votre application. Cette interface (ou cet ensemble d'interfaces) est déterminée par l'application serveur et se trouve généralement publiée dans une bibliothèque de types. Pour des informations spécifiques sur les interfaces publiées par une application serveur spécifique, vous devez consulter la documentation de cette application.

Même si vous ne choisissez pas d'envelopper un objet serveur et de l'installer dans la palette des composants, vous devez rendre sa définition d'interface

accessible à votre application. Pour ce faire, vous pouvez importer les informations de la bibliothèque de types du serveur.

Remarque Vous pouvez également interroger directement les informations de type en utilisant l'API COM, mais Delphi ne propose pas de gestion particulière de ce processus.

Certaines anciennes technologies COM comme OLE ne proposent pas les informations de type dans une bibliothèque de types. Elles reposent à la place sur un ensemble standard d'interfaces prédéfinies. Elles sont présentées dans "Création de clients pour les serveurs n'ayant pas une bibliothèque de types" à la page 35-17.

Importation des informations d'une bibliothèque de types

Pour que votre application client dispose des informations sur le serveur COM, vous devez importer les informations décrivant le serveur qui sont stockées dans la bibliothèque de types du serveur. Votre application peut ensuite utiliser les classes générées pour contrôler l'objet serveur.

Il y a deux manières d'importer les informations d'une bibliothèque de types :

- Vous pouvez utiliser la boîte de dialogue Importation de bibliothèque de types pour importer toutes les informations disponibles sur le type de serveur, les objets et les interfaces. C'est la méthode la plus générale, car elle vous permet d'importer des informations de toute bibliothèque de types et peut, facultativement, créer des composants enveloppe pour toutes les CoClasses de la bibliothèque de types pouvant être créées et qui n'ont pas les indicateurs Hidden, Restricted ou PreDeclID.
- Vous pouvez utiliser la boîte de dialogue Importation d'ActiveX si vous importez la bibliothèque de types d'un contrôle ActiveX. Cela importe les mêmes informations de type, mais ne crée des composants enveloppe que pour les CoClasses qui représentent des contrôles ActiveX.
- Vous pouvez utiliser l'utilitaire en ligne de commande tlibimp.exe qui fournit des options de configuration supplémentaires non disponibles à partir de l'EDI.
- Une bibliothèque de types générée à l'aide d'un expert est automatiquement importée en utilisant le mécanisme de l'élément de menu Importer une bibliothèque de types.

Indépendamment de la méthode utilisée pour importer les informations de la bibliothèque de types, la boîte de dialogue crée une unité de nom *NomBibTypes*_TLB, où *NomBibTypes* est le nom de la bibliothèque de types. Ce fichier contient des déclarations pour les classes, types et interfaces définis dans la bibliothèque de types. En l'incluant dans votre projet, ces définitions deviennent accessibles à votre application afin de pouvoir créer des objets et appeler leurs interfaces. Ce fichier peut être recréé par l'EDI de temps en temps ; de ce fait, des modifications manuelles apportées au fichier ne sont pas recommandées.

Outre l'ajout des définitions de types à l'unité *NomBibTypes*_TLB, la boîte de dialogue peut également créer des classes enveloppe VCL pour toutes les CoClasses définies dans la bibliothèque de types. Quand vous utilisez la boîte de dialogue Importation de bibliothèque de types, ces classes enveloppe sont facultatives. Quand vous utilisez la boîte de dialogue Importation d'ActiveX, elles sont toujours générées pour toutes les CoClasses représentant des contrôles.

Les classes enveloppe générées représentent les CoClasses de votre application et exposent les propriétés et méthodes de ses interfaces. Si une CoClasse gère les interfaces de création d'événements (*IConnectionPointContainer* et *IConnectionPoint*), la classe enveloppe VCL crée un collecteur d'événements afin que vous puissiez affecter des gestionnaires d'événements aux événements aussi facilement que pour les autres composants. Si vous demandez dans la boîte de dialogue d'installer les classes VCL générées dans la palette des composants, vous pouvez utiliser l'inspecteur d'objets pour affecter des valeurs aux propriétés et aux gestionnaires d'événements.

Remarque La boîte de dialogue Importation de bibliothèque de types ne crée par de classe enveloppe pour les objets événement COM+. Pour écrire un client répondant aux événements générés par un objet événement COM+, vous devez créer le collecteur d'événements par code. Ce processus est décrit dans "Gestion des événements COM+" à la page 35-16.

Pour davantage de détails sur le code généré quand vous importez une bibliothèque de types, voir "Code généré par l'importation des informations d'une bibliothèque de types" à la page 35-5.

Utilisation de la boîte de dialogue Importation de bibliothèque de types

Pour importer une bibliothèque de types :

1. Choisissez Projet | Importer une bibliothèque de types.
2. Sélectionnez la bibliothèque de types dans la liste.

   La boîte de dialogue répertorie toutes les bibliothèques recensées sur ce système. Si la bibliothèque de types n'est pas dans la liste, choisissez le bouton Ajouter, trouvez et sélectionnez le fichier de la bibliothèque de types et choisissez OK et répétez l'étape 2. Remarquez que la bibliothèque de types peut être un fichier bibliothèque de types autonome (.tlb, .olb) ou un serveur fournissant une bibliothèque de types (.dll, .ocx, .exe).
3. Si vous voulez générer un composant VCL qui encapsule une CoClasse de la bibliothèque de types, cochez la case Générer le Wrapper de composant. Si vous ne générez pas le composant, vous pouvez toujours utiliser la CoClasse en utilisant les définitions de l'unité *NomBibTypes*_TLB. Cependant vous avez alors à écrire vos propres appels pour créer l'objet serveur et, si nécessaire, configurer un collecteur d'événements.

   La boîte de dialogue Importation de bibliothèque de types n'importe que les CoClasses ayant l'indicateur CanCreate activé et n'ayant pas les indicateurs

Hidden, Restricted ou PreDeclID activés. Ces indicateurs peuvent être redéfinis au moyen de l'utilitaire en ligne de commande tlibimp.exe.

4. Si vous ne voulez pas installer le composant généré dans la palette des composants, choisissez Créer l'unité. Cela génère l'unité *NomBibTypes*_TLB et, si vous avez coché l'option Générer le Wrapper de composant à l'étape 3, y ajoute les déclarations du composant enveloppe. Cela entraîne la sortie de la boîte de dialogue Importation de bibliothèque de types.
5. Si vous voulez installer dans la palette des composants le composant enveloppe généré, sélectionnez la page de la palette dans laquelle vous voulez placer le composant et choisissez Installer. Cela génère l'unité *NomBibTypes*_TLB, comme le bouton Créer l'unité, puis cela affiche la boîte de dialogue Installation de composant qui vous permet de spécifier le paquet (existant ou nouveau) dans lequel placer le composant. Ce bouton est grisé si aucun composant ne peut être créé pour la bibliothèque de types.

Quand vous sortez de la boîte de dialogue Importation de bibliothèque de types, l'unité *NomBibTypes*_TLB apparaît dans le répertoire spécifié par l'option Répertoire unité. Ce fichier contient les déclarations pour les éléments définis dans la bibliothèque de types, ainsi que pour les composants enveloppe générés si vous avez coché l'option Générer le Wrapper de composant.

De plus, si vous avez installé le composant enveloppe généré, l'objet serveur défini par la bibliothèque de types a été ajouté à la palette des composants. Vous pouvez utiliser l'inspecteur d'objets pour définir les propriétés du serveur ou écrire un gestionnaire d'événement. Si vous ajoutez le composant dans une fiche ou un module de données, vous pouvez cliquer dessus à la conception avec le bouton droit de la souris pour voir sa page de propriétés (s'il en a une).

Remarque La page Serveurs de la palette de composants contient un certain nombre d'exemples de serveurs Automation qui ont été importés de cette manière.

Utilisation de la boîte de dialogue Importation d'ActiveX

Pour importer un contrôle ActiveX :

1. Choisissez Composant | Importer un contrôle ActiveX.
2. Sélectionnez la bibliothèque de types dans la liste.

   La boîte de dialogue répertorie toutes les bibliothèques recensées définissant un contrôle ActiveX (c'est un sous-ensemble de la liste de bibliothèques de types affichée par la boîte de dialogue Importation de bibliothèque de types). Si la bibliothèque de types n'est pas dans la liste, choisissez le bouton Ajouter, trouvez et sélectionnez le fichier de la bibliothèque de types, choisissez OK et répétez l'étape 2. Remarquez que la bibliothèque de types peut être un fichier bibliothèque de types autonome (.tlb, .olb) ou un serveur ActiveX (.dll, .ocx).
3. Si vous ne voulez pas installer le contrôle ActiveX dans la palette des composants, choisissez Créer l'unité. Cela génère l'unité *NomBibTypes*_TLB et y ajoute la déclaration de ses envelopppes de composants. Cela entraîne la sortie de la boîte de dialogue Importation d'ActiveX.

4 Si vous voulez installer le contrôle dans la palette des composants, sélectionnez la page de la palette dans laquelle vous voulez placer le composant et choisissez Installer. Cela génère l'unité *NomBibTypes*_TLB, comme le bouton Créer l'unité, puis cela affiche la boîte de dialogue Installation de composant qui vous permet de spécifier le paquet (existant ou nouveau) dans lequel placer le composant

Quand vous sortez de la boîte de dialogue Importation d'ActiveX, l'unité *NomBibTypes*_TLB apparaît dans le répertoire spécifié par l'option Répertoire unité. Ce fichier contient les déclarations pour les éléments définis dans la bibliothèque de types, ainsi que pour le composant enveloppe généré pour le contrôle ActiveX.

Remarque A la différence de la boîte de dialogue Importation de bibliothèque de types où c'est facultatif, la boîte de dialogue Importation d'ActiveX génère toujours un composant enveloppe. En effet, comme c'est un contrôle visuel, un contrôle ActiveX a besoin de l'assistance supplémentaire du composant enveloppe pour pouvoir s'adapter aux fiches VCL.

Si vous installez le composant enveloppe généré, il y a un contrôle ActiveX dans la palette des composants. Vous pouvez utiliser l'inspecteur d'objets pour définir ses propriétés ou écrire des gestionnaires d'événements. Si vous ajoutez le contrôle dans une fiche ou un module de données, vous pouvez cliquer dessus à la conception avec le bouton droit de la souris pour voir sa page de propriétés (s'il en a une).

Remarque La page ActiveX de la palette des composants contient un certain nombre d'exemples de contrôles ActiveX importés de cette manière.

Code généré par l'importation des informations d'une bibliothèque de types

Une fois une bibliothèque de types importée, vous pouvez visualiser l'unité *NomBibTypes*_TLB générée. En haut, vous trouverez les éléments suivants :

- Les déclarations des constantes donnant des noms symboliques aux GUIDS de la bibliothèque de types, ses interfaces et ses CoClasses. Le nom de ces constantes est généré comme suit :
  - Le GUID de la bibliothèque de types de la forme LBID_*NomBibTypes*, où *NomBibTypes* est le nom de la bibliothèque de types.
  - Le GUID d'une interface a la forme IID_*NomInterface*, où *NomInterface* est le nom de l'interface.
  - Le GUID d'une dispinterface a la forme DIID_*NomInterface*, où *NomInterface* est le nom de la dispinterface.
  - Le GUID d'une CoClasse a la forme CLASS_*NomClasse*, où *NomClasse* est le nom de la CoClasse.
- Des déclarations des CoClasses de la bibliothèque de types. Elles correspondent à chaque CoClasse de son interface par défaut.

- Des déclarations pour les interfaces et dispinterfaces de la bibliothèque de types.
- Des déclarations pour une classe créateur pour chaque CoClasse dont l'interface par défaut gère les liaisons par Vtable. La classe créateur a deux méthodes de classe, *Create* et *CreateRemote*, qui peuvent s'utiliser pour instancier la CoClasse localement (*Create*) ou à distance (*CreateRemote*). Ces méthodes renvoient l'interface par défaut de la CoClasse.

Ces déclarations fournissent ce qu'il faut pour créer des instances de la CoClasse et accéder à son interface. Il vous faut ajouter le fichier *NomBibTypes*_TLB.pas généré à la clause uses de l'unité dans laquelle vous voulez effectuer le lien avec une CoClasse et appeler ses interfaces.

Remarque Cette portion de l'unité *NomBibTypes*_TLB est également générée quand vous utilisez l'éditeur de bibliothèques de types ou l'utilitaire en ligne de commande TLIBIMP.

Si vous voulez utiliser un contrôle ActiveX, vous avez également besoin de la classe enveloppe VCL générée en plus des déclarations décrites ci-dessus. L'enveloppe VCL gère les problèmes de gestion de fenêtre du contrôle. Vous pouvez également avoir d'autres classes enveloppe VCL générées pour d'autres CoClasses avec la boîte de dialogue Importation de bibliothèque de types. Ces enveloppes VCL simplifient la création d'objets serveur et l'appel de leurs méthodes. Elles sont particulièrement recommandées si vous voulez que votre application client réponde aux événements.

Les déclarations pour les enveloppes VCL générées apparaissent en bas de la section interface. Les composants enveloppe des contrôles ActiveX sont des descendants de *TOleControl*. Les composants enveloppe pour les objets Automation descendent de *TOleServer*. Le composant enveloppe généré ajoute les propriétés, méthodes et événements exposés par l'interface de la CoClasse. Vous pouvez utiliser ce composant comme vous utilisez tous les autres composants VCL.

Attention Vous ne devez pas modifier l'unité générée *NomBibTypes*_TLB. Elle est régénérée à chaque fois que la bibliothèque de types est rafraîchie, ce qui écrase toutes les modifications.

Remarque Pour des informations les plus à jour, voir les commentaires placés dans l'unité générée automatiquement *NomBibTypes*_TLB.

Contrôle d'un objet importé

Après avoir importé les informations de la bibliothèque de types, vous êtes prêts à commencer à programmer avec les objets importés. La manière de procéder dépend pour partie des objets, et d'autre part du choix de créer des composants enveloppe.

Utilisation des composants enveloppe

Si vous avez généré un composant enveloppe pour l'objet serveur, l'écriture de l'application COM n'est pas très différente de celle d'une application contenant des composants VCL. Les événements de l'objet serveur sont déjà encapsulés dans le composant VCL. Il vous suffit juste d'affecter des gestionnaires d'événements, de définir des valeurs de propriétés et d'appeler des méthodes.

Pour utiliser les propriétés, méthodes et événements de l'objet, voir la documentation de l'objet serveur. Le composant enveloppe fournit automatiquement une interface double quand c'est possible. Delphi détermine la forme de la VTable à partir des informations de la bibliothèque de types.

De plus, le nouveau composant hérite de sa classe de base des propriétés et méthodes importantes.

Enveloppes ActiveX

Vous devez toujours utiliser un composant enveloppe pour accueillir des contrôles ActiveX car le composant enveloppe intègre la fenêtre du contrôle dans le modèle VCL.

Le contrôle ActiveX hérite de *TOleControl* des propriétés et méthodes qui vous permettent d'accéder à l'interface sous-jacente ou d'obtenir des informations sur le contrôle. Cependant, la plupart des applications n'en n'ont pas besoin. Vous utilisez à la place le contrôle importé comme vous utiliseriez tout autre contrôle VCL.

Généralement, les contrôles ActiveX proposent une page de propriétés qui vous permet de définir leurs propriétés. Les pages de propriétés sont similaires aux éditeurs que certains composants affichent quand vous double-cliquez dessus dans le concepteur de fiche. Pour afficher la page de propriétés d'un contrôle ActiveX, cliquez dessus avec le bouton droit de la souris et choisissez Propriétés.

La manière dont vous utilisez les contrôles ActiveX importés est déterminée par l'application serveur. Cependant, les contrôles ActiveX utilisent un ensemble standard de notifications quand ils représentent les données d'un champ de base de données. Pour davantage d'informations sur la manière d'accueillir des contrôles ActiveX orientés données, voir "Utilisation de contrôles ActiveX orientés données" à la page 35-9.

Enveloppes des serveurs Automation

Les enveloppes des objets Automation vous permettent de contrôler comment vous voulez constituer la connexion avec l'objet serveur :

- La propriété *ConnectKind* indique si le serveur est local ou distant et si vous voulez vous connecter avec un serveur en cours d'exécution ou si une nouvelle instance doit être lancée. Pour se connecter avec un serveur distant, vous devez spécifier le nom de la machine en utilisant la propriété *RemoteMachineName*.

- Une fois la valeur de *ConnectKind* spécifiée, il y a trois manières de connecter le composant au serveur :
  - Une connexion explicite au serveur en appelant la méthode *Connect* du composant.
  - Vous pouvez indiquer au composant de se connecter automatiquement avec le serveur au démarrage de votre application en initialisant la propriété *AutoConnect* à *True*.
  - Vous n'avez pas besoin de vous connecter explicitement au serveur. Le composant crée automatiquement une connexion quand vous utilisez l'une des propriétés et méthodes du serveur par le biais du composant.

Vous appelez les méthodes ou utilisez les propriétés comme pour tout autre composant :

```
TServerComponent1.FaireUneChose;
```

Il est facile de gérer les événements car vous pouvez utiliser l'inspecteur d'objets pour écrire des gestionnaires d'événements. Toutefois, vous devez faire attention car les gestionnaires d'événements de votre composant ont des paramètres légèrement différents de ceux définis pour les événements de la bibliothèque de types. Plus précisément, les types pointeurs (paramètres var et pointeur d'interface) sont changés en Variants. Vous devez explicitement transtyper les paramètres var dans le type sous-jacent avant de leur affecter une valeur. Les pointeurs d'interface peuvent être transtypés dans le type d'interface approprié en utilisant l'opérateur **as**.Par exemple, l'exemple de code suivant est le gestionnaire d'événement pour l'événement OnNewWorkBook de ExcelApplication. Le gestionnaire d'événement a un paramètre qui fournit l'interface d'une autre CoClasse (ExcelWorkbook). Cependant, l'interface n'est pas transmise comme un pointeur d'interface ExcelWorkBook, mais comme un OleVariant.

```
procedure TForm1.XLappNewWorkbook(Sender: TObject; var Wb:OleVariant);
begin
  { Remarque : le OleVariant de l'interface est transtypé dans le type adéquat }
  ExcelWorkbook1.ConnectTo((iUnknown(wb) as ExcelWorkbook));
end;
```

Dans cet exemple, le gestionnaire d'événement affecte le classeur à un composant ExcelWorkbook (ExcelWorkbook1). Cela illustre la manière de connecter un composant enveloppe à une interface existante en utilisant la méthode *ConnectTo*. La méthode *ConnectTo* est ajoutée au code généré du composant enveloppe.

Les serveurs qui ont un objet application exposent une méthode Quit pour cet objet afin de permettre aux clients de terminer la connexion. Généralement Quit expose des fonctionnalités équivalentes à l'utilisation du menu Fichier pour la sortie de l'application. Du code pour l'appel de la méthode Quit est généré dans la méthode *Disconnect* de votre composant. Il est possible d'appeler la méthode Quit sans paramètre. Le composant enveloppe a également une propriété *AutoQuit*. *AutoQuit* oblige le contrôleur à appeler Quit quand le composant est libéré. Si vous voulez déconnecter à un autre moment, ou si la méthode Quit nécessite des paramètres, vous devez l'appeler explicitement. Quit apparaît comme méthode publique du composant généré.

Utilisation de contrôles ActiveX orientés données

Quand vous utilisez un contrôle ActiveX orienté données dans une application Delphi, vous devez l'associer à la base de données dont il représente les données. Pour ce faire, vous avez besoin d'un composant source de données tout comme pour les contrôles orientés données VCL.

Une fois le contrôle ActiveX orienté données placé dans le concepteur de fiche, affectez à sa propriété *DataSource* la source de données qui représente l'ensemble de données souhaité. Une fois la source de données spécifiée, vous pouvez utiliser l'éditeur de liaisons de données de contrôle ActiveX pour lier la propriété de liaison aux données du contrôle à un champ de l'ensemble de données.

Pour afficher l'éditeur de liaisons de données de contrôle ActiveX, cliquez avec le bouton droit de la souris dans le contrôle ActiveX orienté données pour afficher le menu contextuel. Outre les habituelles options du menu Fiche, la commande Liaison de données apparaît. Sélectionnez cette option pour afficher l'éditeur de liaisons de données qui énumère tous les champs de la source de données et les propriétés pouvant être liées du contrôle ActiveX.

Pour lier un champ à une propriété :

1. Dans l'éditeur de liaisons de données de contrôle ActiveX, sélectionnez un champ et un nom de propriété.

   Nom de champ énumère les champs de la base de données et Nom de propriété énumère les propriétés du contrôle ActiveX qui peuvent être liées à un champ de base de données. Le dispID de la propriété est entre parenthèses, par exemple Value(12).

2. Choisissez Lier et OK.

Remarque Aucune propriété n'apparaît dans la boîte de dialogue si le contrôle ActiveX ne contient pas de propriétés orientés données. Pour activer une liaison de données simple pour une propriété d'un contrôle ActiveX, utilisez la bibliothèque de types comme indiqué dans "Activation de la liaison de données simple avec la bibliothèque de types" à la page 38-12.

L'exemple suivant parcourt les étapes de l'utilisation d'un contrôle ActiveX orienté données dans le conteneur Delphi. Cet exemple utilise le contrôle calendrier Microsoft disponible si Microsoft Office 97 est installé sur votre système.

1. Dans le menu principal Delphi, choisissez Composant | Importer un contrôle ActiveX.
2. Sélectionnez un contrôle ActiveX orienté données comme Microsoft Calendar control 8.0, changez son nom de classe en *TCalendarAXControl* puis choisissez Installer.
3. Dans la boîte de dialogue Installation de composant, choisissez OK pour ajouter le contrôle au paquet utilisateur par défaut, cela rend le contrôle accessible depuis la palette.

4. Choisissez Tout fermer puis Fichier | Nouveau | Application pour commencer une nouvelle application.
5. Déposez dans la fiche un objet *TCalendarAXControl* de la page ActiveX que vous venez d'ajouter à la palette.
6. Depuis la page AccèsBD, déposez dans la fiche des objets *DataSource* et *Table*.
7. Sélectionnez l'objet *DataSource* et initialisez sa propriété *DataSet* à *Table1*.
8. Sélectionnez l'objet *Table* et effectuez les opérations suivantes :
   - Initialisez la propriété *DatabaseName* à DBDEMOS
   - Initialisez la propriété *TableName* à EMPLOYEE.DB
   - Initialisez la propriété *Active* à *True*
9. Sélectionnez l'objet *TCalendarAXControl* et initialisez sa propriété *DataSource* à *DataSource1*.
10. Sélectionnez l'objet *TCalendarAXControl*, cliquez avec le bouton droit de la souris et choisissez Liaisons de données pour ouvrir l'éditeur de liaisons de données de contrôle ActiveX.

    Nom de champ énumère les champs de la base de données et Nom de propriété énumère les propriétés du contrôle ActiveX qui peuvent être liées à un champ de base de données. Le dispID de la propriété est entre parenthèses.
11. Sélectionnez le champ *HireDate* et la propriété *Value*, choisissez Lier puis OK.

    Le nom de champ et la propriété sont maintenant liés.
12. Depuis l'onglet ContrôleBD, déposez un objet *DBGrid* dans la fiche et initialisez sa propriété *DataSource* à *DataSource1*.
13. Depuis l'onglet ContrôleBD, déposez un objet *DBNavigator* dans la fiche et initialisez sa propriété *DataSource* à *DataSource1*.
14. Exécutez l'application.
15. Testez l'application comme suit :

    Le champ *HireDate* étant affiché dans l'objet *DBGrid*, parcourez la base de données en utilisant le navigateur de données. Les dates changent dans le contrôle ActiveX quand vous vous déplacez dans la base de données.

Exemple : impression d'un document avec Microsoft Word

Les étapes suivantes illustrent la manière de créer un contrôleur Automation imprimant un document avec Microsoft Word 8 d'Office 97.

Avant de commencer : Créez un nouveau projet composé d'une fiche, d'un bouton et d'une boîte de dialogue Ouvrir (*TOpenDialog*). Ces contrôles constituent le contrôleur Automation.

Etape 1 : Préparation de Delphi pour cet exemple

Pour vous faciliter la tâche, Delphi fournit dans la palette de composants de nombreux serveurs courants, comme Word, Excel et PowerPoint. Pour illustrer l'importation d'un serveur, nous utilisons Word. Puisqu'il existe déjà sur la palette de composants, la première étape consiste à supprimer le paquet contenant Word afin de l'installer sur la palette. L'étape 4 décrit comment rétablir la palette de composants dans son état ordinaire.

Pour supprimer Word de la palette de composants,

1. Choisissez Composant | Installer des paquets.
2. Cliquez sur les composants serveur Automation exemple de Borland et choisissez Supprimer.

   La page Serveurs de la palette de composants ne contient plus aucun serveur fourni avec Delphi (si aucun autre serveur n'a été importé, la page Serveurs disparaît également).

Etape 2 : importation de la bibliothèque de types Word

Pour importer la bibliothèque de types Word,

1. Choisissez Projet | Importer une bibliothèque de types.
2. Dans la boîte de dialogue d'importation de bibliothèque de types,
   1. Sélectionnez la bibliothèque d'objets Microsoft Office 8.0.

      Si Word (version 8) n'est pas dans la liste, choisissez le bouton Ajouter, puis dans le dossier Program Files\Microsoft Office\Office, sélectionnez le fichier de la bibliothèque de types de Word, MSWord8.olb, choisissez Ajouter et sélectionnez Word (Version 8) dans la liste.
   2. Dans Page de palette, choisissez Serveurs.
   3. Choisissez Installer.

      La boîte de dialogue d'installation apparaît. Sélectionnez l'onglet Dans nouveaux paquets et tapez WordExample pour créer un nouveau paquet contenant cette bibliothèque de types.
3. Retournez à Serveurs dans Page de palette, sélectionnez WordApplication et placez-le sur une fiche.
4. Ecrivez un gestionnaire d'événement pour l'objet bouton, comme décrit à l'étape suivante.

Etape 3 : utilisation d'un objet interface VTable ou de répartition pour contrôler Microsoft Word

Vous pouvez employer un objet interface VTable ou de répartition pour contrôler Microsoft Word.

Utilisation d'un objet interface VTable

En déposant une instance de l'objet WordApplication dans la fiche, vous pouvez accéder facilement au contrôle en utilisant un objet interface VTable. Pour ce faire, vous devez simplement appeler les méthodes de la classe que vous venez de créer. Pour Word, il s'agit de la classe *TWordApplication*.

1 Sélectionnez le bouton, double-cliquez sur son gestionnaire d'événement *OnClick* et spécifiez le code de gestion d'événement suivant :

```
procedure TForm1.Button1Click(Sender: TObject);

var
   FileName: OleVariant;
begin
 if OpenDialog1.Execute then
  begin
     FileName := OpenDialog1.FileName;

WordApplication1.Documents.Open(FileName,
   EmptyParam,EmptyParam,EmptyParam,
   EmptyParam,EmptyParam,EmptyParam,
   EmptyParam,EmptyParam,EmptyParam);

WordApplication1.ActiveDocument.PrintOut(
   EmptyParam,EmptyParam,EmptyParam,
   EmptyParam, EmptyParam,EmptyParam,
   EmptyParam,EmptyParam,EmptyParam,
   EmptyParam,EmptyParam,EmptyParam,
   EmptyParam,EmptyParam);
   end;

end;
```

2 Générez et exécutez le programme, en cliquant sur le bouton, Word vous demande le fichier à imprimer.

Utilisation d'un objet interface de répartition

Vous pouvez également utiliser une interface de répartition avec liaison tardive. Pour utiliser un objet interface de répartition, il faut créer et initialiser l'objet Application en utilisant la classe enveloppe de répartition _ApplicationDisp comme suit. Remarquez que les méthodes de l'interface de répartition sont "décrites" dans le source comme renvoyant des interfaces Vtable, alors qu'en fait vous devez les transtyper en interfaces de répartition.

1 Sélectionnez le bouton, double-cliquez sur son gestionnaire d'événement *OnQuit* et spécifiez le code de gestion d'événement suivant :

```
procedure TForm1.Button1Click(Sender: TObject);
var
MyWord : _ApplicationDisp;
FileName : OleVariant;
begin
  if OpenDialog1.Execute then
  begin
    FileName := OpenDialog1.FileName;
```

```
    MyWord := CoWordApplication.Create as
        _ApplicationDisp;
    (MyWord.Documents as DocumentsDisp).Open(FileName,EmptyParam,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam,EmptyParam,EmptyParam);
    (MyWord.ActiveDocument as _DocumentDisp).PrintOut(EmptyParam,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam,EmptyParam,EmptyParam);
    MyWord.Quit(EmptyParam,EmptyParam,EmptyParam);
  end;
end;
```

2 Générez et exécutez le programme, en cliquant sur le bouton, Word vous demande le fichier à imprimer.

Etape 4 : nettoyage de l'exemple

A la fin de cet exemple, vous pouvez rétablir Delphi dans son état initial.

1 Supprimez les objets de cette page Serveurs :

1. Choisissez Composant | Installer des paquets.
2. Dans la liste, sélectionnez le paquet WordExample et choisissez Supprimer.
3. Confirmez en choisissant Oui.
4. Sortez de la boîte de dialogue Installer des paquets en choisissant OK.

2 Retournez dans le paquet Composants Serveur automation exemple Borland :

1. Choisissez Composant | Installer des paquets.
2. Cliquez sur Ajouter.
3. Dans la boîte de dialogue affichée, choisissez dclaxserver50.bpl.
4. Sortez de la boîte de dialogue Installer des paquets en choisissant OK.

Ecriture de code client basé sur les définitions de la bibliothèque de types

Si vous êtes obligé d'utiliser un composant enveloppe pour accueillir un contrôle ActiveX, vous pouvez par contre écrire un contrôleur Automation en utilisant uniquement les définitions de la bibliothèque de types qui apparaissent dans l'unité *NomBibTypes*_TLB. Ce processus est un tout petit peu plus compliqué que si vous laissez un composant enveloppe faire le travail, en particulier si vous avez besoin de répondre aux événements.

Connexion à un serveur

Avant de pouvoir piloter un serveur Automation depuis votre application contrôleur, vous devez obtenir une référence sur une interface qu'il gère. Généralement, vous vous connectez au serveur via son interface principale. Par exemple vous vous connectez à Microsoft Word via le composant WordApplication.

Si l'interface principale est une interface double, vous pouvez utiliser les objets créateur du fichier *NomBibTypes*_TLB.pas. Les classes de créateur portent le même nom que la CoClasse sans le préfixe "Co". Vous pouvez vous connecter à un serveur situé sur la même machine en appelant la méthode *Create* ou à un serveur situé sur une autre machine en utilisant la méthode *CreateRemote*. Comme *Create* et *CreateRemote* sont des méthodes de classe, vous n'avez pas besoin d'une instance de la classe de créateur pour les appeler.

```
MyInterface := CoServerClassName.Create;
MyInterface := CoServerClassName.CreateRemote('Machine1');
```

Create et *CreateRemote* renvoient l'interface par défaut de la CoClasse.

Si l'interface par défaut est une interface de répartition, il n'y a pas de classe créateur générée pour la CoClasse. Vous devez à la place, appeler la fonction globale *CreateOleObject*, en lui transmettant le GUID de la CoClasse (il y a une constante pour ce GUID définie en haut de l'unité _TLB). *CreateOleObject* renvoie un pointeur IDispatch pour l'interface par défaut.

Contrôle d'un serveur Automation en utilisant une interface double

Après avoir utilisé la classe créateur générée automatiquement pour vous connecter au serveur, appelez les méthodes de l'interface. Par exemple :

```
var
  MyInterface : _Application;
begin
  MyInterface := CoWordApplication.Create;
  MyInterface.DoSomething;
```

L'interface et la classe créateur sont définies dans l'unité *NomBibTypes*_TLB générée automatiquement quand vous importez une bibliothèque de types.

Pour des informations sur les interfaces doubles, voir "Interfaces doubles" à la page 36-14.

Contrôle d'un serveur Automation en utilisant une interface de répartition

Généralement, vous utilisez l'interface double pour contrôler le serveur Automation, comme décrit ci-dessus. Toutefois, il se peut que vous deviez contrôler un serveur Automation avec un objet interface de répartition s'il n'y a pas d'interface double disponible.

Pour appeler les méthodes d'une interface de répartition :

1. Connectez-vous au serveur, en utilisant la fonction globale *CreateOleObject*.
2. Utilisez l'opérateur **as** pour transtyper l'interface *IDispatch* renvoyée par *CreateOleObject* en une interface de répartition pour la CoClasse. Ce type dispinterface est déclaré dans l'unité *NomBibTypes*_TLB.
3. Contrôlez le serveur Automation en appelant les méthodes de la dispinterface. Vous pouvez utiliser autrement les interfaces de répartition en les affectant à un *Variant*. En affectant l'interface renvoyée par *CreateOleObject* à un *Variant*, vous pouvez tirer profit de la gestion intégrée des interfaces que propose le type *Variant*. Appelez simplement les méthodes de l'interface et c'est le *Variant* qui

gère automatiquement les appels *IDispatch*, la récupération de l'identificateur de répartition et les appels de méthode appropriés. Le type Variant possède une gestion intégrée permettant l'appel des interfaces de répartition, via ses méthodes

```
var.
  V: Variant;
begin
  V:= CreateOleObject('TheServerObject');
  V.NomMéthode; { appelle la méthode spécifiée }
  ...
```

Un avantage de l'utilisation de *Variants* est qu'il n'est pas nécessaire d'importer la bibliothèque de types car les *Variants* utilisent uniquement les méthodes standard de *IDispatch* pour appeler le serveur. La contrepartie, c'est que les *Variants* sont plus lents car ils utilisent une liaison dynamique à l'exécution.

Pour davantage d'informations sur les interfaces de répartition, voir "Interfaces d'Automation" à la page 36-13.

Gestion des événements dans un contrôleur Automation

Lorsque vous générez un composant enveloppe pour un objet dont vous importez la bibliothèque de types, vous pouvez répondre aux événements en utilisant simplement les événements ajoutés au composant généré. Néanmoins, si vous n'utilisez pas un composant enveloppe (ou si le serveur utilise des événements COM+), vous devez écrire vous-même le code du collecteur d'événements.

Gestion des événements Automation par code

Avant de pouvoir gérer les événements, vous devez définir un collecteur d'événements. C'est une classe qui implémente l'interface de répartition des événements qui est définie dans la bibliothèque de types du serveur.

Pour écrire le collecteur d'événements, créez un objet qui implémente l'interface de répartition des événements :

```
TServerEventsSink = class(TObject, _TheServerEvents)
...{ déclarez ici les méthodes de _TheServerEvents }
end;
```

Quand vous disposez d'une instance de votre collecteur d'événements, vous devez informer l'objet serveur de son existence afin que le serveur puisse l'appeler. Pour ce faire, appelez la procédure globale *InterfaceConnect* en lui transmettant :

- L'interface du serveur qui génère les événements.
- Le GUID de l'interface événement que gère votre collecteur d'événements.
- Une interface IUnknown pour votre collecteur d'événements.
- Une variable qui reçoit un Longint représentant la connexion entre le serveur et votre collecteur d'événements.

```
{MyInterface est l'interface serveur que vous obtenez quand vous vous connectez au serveur
}
InterfaceConnect(MyInterface, DIID_TheServerEvents,
                 MyEventSinkObject as IUnknown, cookievar);
```

Après avoir appelé *InterfaceConnect*, votre collecteur d'événements est connecté et reçoit les appels du serveur quand des événements se produisent.

Vous devez terminer la connexion avant de libérer le collecteur d'événements. Pour ce faire, appelez la procédure globale *InterfaceDisconnect* en lui transmettant les mêmes paramètres, sauf l'interface de votre collecteur d'événements (le paramètre final étant en entrée et non plus en sortie) :

```
InterfaceDisconnect(MyInterface, DIID_TheServerEvents, cookievar);
```

Remarque Vous devez être certain que le serveur a libéré sa connexion avec votre collecteur d'événements avant de libérer le collecteur. Comme vous ne savez pas comment le serveur répond à la notification de déconnexion initiée par *InterfaceDisconnect*, cela peut entraîner une condition si vous libérez votre collecteur d'événements immédiatement après l'appel. Le moyen le plus simple de vous prémunir contre cela est que votre collecteur d'événements gère son propre compteur de références qui n'est pas décrémenté tant que le serveur n'a pas libéré l'interface du collecteur d'événements.

Gestion des événements COM+

Avec COM+, les serveurs utilisent un objet utilitaire spécial pour générer des événements à la place d'un groupe d'interfaces spéciales (*IConnectionPointContainer* et *IConnectionPoint*). Donc, vous ne pouvez pas utiliser un collecteur d'événements qui descend de *TEventDispatcher*. *TEventDispatcher* est conçue pour fonctionner avec des interfaces, pas avec des objets événement COM+.

Au lieu de définir un collecteur d'événements, votre application client définit un objet abonné. Comme les collecteurs d'événements, les objets abonné fournissent l'implémentation de l'interface événement. Ils sont différents des collecteurs d'événements car ils décrivent un objet événement particulier et non une connexion avec un point de connexion d'un serveur.

Pour définir un objet abonné, utilisez l'expert objet COM en sélectionnant l'interface de l'objet événement que vous voulez implémenter. L'expert génère une unité d'implémentation avec des squelettes de méthodes que vous pouvez renseigner pour créer vos gestionnaires d'événements. Pour davantage d'informations sur l'utilisation de l'expert objet COM pour implémenter une interface existante, voir "Utilisation de l'expert objet COM" à la page 36-2.

Remarque Vous pouvez avoir à ajouter l'interface de l'objet événement dans le registre en utilisant l'expert si elle n'apparaît pas dans la liste des interfaces que vous pouvez implémenter.

Une fois l'objet abonné créé, vous devez vous abonner à l'interface de l'objet événement ou individuellement aux méthodes (événements) de cette interface. Vous pouvez souscrire à trois types d'abonnements :

- **Abonnements temporaires.** Comme les collecteurs d'événements habituels, les abonnements temporaires sont liés à la durée de vie de l'instance d'objet. Quand l'objet abonné est libéré, l'abonnement s'arrête et COM+ ne lui transmet plus les événements.

- **Abonnements permanents.** Ils sont liés à la classe de l'objet et non à une instance d'objet spécifique. Quand l'événement se produit, COM recherche ou démarre un processus de l'objet abonné et appelle son gestionnaire d'événement. Les objets en processus (DLL) utilisent ce type d'abonnement.
- **Abonnements par utilisateur.** Ces abonnements offrent une version plus sécurisée des abonnements temporaires. L'objet abonné et l'objet serveur qui déclenchent les événements doivent être exécutés avec le même compte utilisateur sur la même machine.

Remarque Les objets qui s'abonnent à des événements COM+ doivent être installés dans une application COM+.

Création de clients pour les serveurs n'ayant pas une bibliothèque de types

Certaines technologies COM plus anciennes comme OLE (Object Linking and Embedding) ne fournissent pas les informations de type dans une bibliothèque de types. Elles utilisent à la place un jeu standard d'interfaces prédéfinies. Pour écrire des clients accueillant de tels objets, vous pouvez utiliser les composants *TOleContainer*. Ce composant apparaît dans la page Système de la palette des composants.

TOleContainer se comporte comme le site d'accueil d'un objet Ole2. Il implémente l'interface *IOleClientSite* et, de manière facultative, *IOleDocumentSite*. La communication est gérée en utilisant des verbes OLE.

Pour utiliser *TOleContainer* :

1. Placez un composant *TOleContainer* dans votre fiche.
2. Initialisez la propriété *AllowActiveDoc* à *True* si vous voulez pouvoir accueillir un document actif.
3. Initialisez la propriété *AllowInPlace* pour indiquer si l'objet accueilli doit apparaître dans le *TOleContainer* ou dans une fenêtre séparée.
4. Ecrivez des gestionnaires d'événements pour répondre quand l'objet est activé, désactivé, déplacé ou redimensionné.
5. Pour lier l'objet *TOleContainer* à la conception, cliquez avec le bouton droit de la souris et choisissez Insérer un objet. Dans la boîte de dialogue Insertion d'objet, choisissez l'objet serveur à accueillir.
6. Vous avez plusieurs moyens de lier l'objet *TOleContainer* à l'exécution, selon la manière dont vous voulez identifier l'objet serveur. La méthode *CreateObject* attend un identificateur de programme, *CreateObjectFromFile* attend le nom d'un fichier dans lequel l'objet a été enregistré, *CreateObjectFromInfo* attend un record contenant des informations sur la manière de créer l'objet ou *CreateLinkToFile* qui attend le nom d'un fichier dans lequel l'objet a été enregistré et effectue une liaison et non une incorporation.

7. Une fois l'objet lié, vous pouvez accéder à son interface en utilisant la propriété *OleObjectInterface*. Cependant, comme la communication avec les objets Ole2 était basée sur les verbes OLE, vous préférerez probablement envoyer des commandes au serveur en utilisant la méthode *DoVerb*.
8. Quand vous voulez libérer l'objet serveur, appelez la méthode *DestroyObject*.

Chapitre

36

Création de serveurs COM simples

Delphi dispose d'experts pour vous aider à créer divers objets COM. Les objets COM les plus simples sont des serveurs qui exposent des propriétés et des méthodes (et éventuellement des événements) via une interface par défaut que leurs clients peuvent appeler.

Remarque Automation et les serveurs COM ne sont pas disponibles dans les applications CLX. Cette technologie n'est utilisée que sur Windows et n'est pas multiplate-forme.

Deux experts, en particulier, simplifient le processus de création d'objets COM simples :

- L' expert objet COM construit un objet COM léger dont l'interface par défaut descend de *IUnknown* ou qui implémente une interface déjà recensée dans votre système. Cet expert offre la plus grande flexibilité dans les types d'objets COM qu'il est possible de créer.
- L' expert objet Automation crée un objet Automation simple dont l'interface par défaut descend de *IDispatch*. *IDispatch* introduit un mécanisme de marshaling standard et gère la liaison tardive des appels d'interfaces.

Remarque COM définit de nombreuses interfaces standard et des mécanismes pour gérer des situations spécifiques. Les experts Delphi automatisent les opérations les plus courantes. Cependant, certaines tâches, comme le marshaling personnalisé ne sont gérées par aucun expert Delphi. Pour des informations à ce propos et sur d'autres technologies qui ne sont pas explicitement gérées par Delphi, reportez-vous à la documentation Microsoft Developer's Network (MSDN). Le site Web Microsoft propose également des informations à jour sur la gestion COM.

Présentation de la création d'un objet COM

Que vous utilisiez l'expert objet Automation pour créer un nouveau serveur Automation ou l'expert objet COM pour créer d'autres types d'objets COM, le processus à suivre est le même.

Il implique les étapes suivantes :

1. Conception de l'objet COM.
2. Utilisation de l'expert objet COM ou de l'expert objet Automation pour créer l'objet serveur.
3. Définition de l'interface que l'objet expose aux clients.
4. Recensement de l'objet COM.
5. Test et débogage de l'application.

Conception d'un objet COM

Quand vous concevez un objet COM, vous devez choisir les interfaces COM que vous voulez implémenter. Vous pouvez écrire des objets COM pour implémenter une interface qui a déjà été définie ou vous pouvez définir une nouvelle interface que votre objet doit implémenter. De plus, votre objet peut gérer plusieurs interfaces. Pour des informations sur les interfaces COM standard que vous pouvez gérer, voir la documentation MSDN.

- Pour créer un objet COM qui implémente une interface existante, utilisez l'expert objet COM.
- Pour créer un objet COM qui implémente une nouvelle interface que vous définissez, utilisez l'expert objet COM ou l'expert objet Automation. L'expert objet COM peut générer une nouvelle interface par défaut qui descend de *IUnknown*, l'expert objet Automation attribue à l'objet une interface par défaut qui descend de *IDispatch*. Indépendamment de l'expert utilisé, vous pouvez toujours employer ultérieurement l'éditeur de bibliothèques de types pour modifier l'interface parent de l'interface par défaut que l'expert a généré.

Outre le choix des interfaces à gérer, vous devez décider si l'objet COM est un serveur en ou hors processus ou un serveur distant. Pour les serveurs en processus, les serveurs hors processus et les serveurs distants qui utilisent une bibliothèque de types, COM vous délègue les données. Sinon, vous devez choisir comment transmettre les données aux serveurs hors processus. Pour des informations sur les types de serveurs, voir "Serveurs en processus, hors processus et distants" à la page 33-7.

Utilisation de l'expert objet COM

L'expert objet COM effectue les opérations suivantes :

- Création d'une nouvelle unité.
- Définition d'une nouvelle classe qui descend de *TCOMObject* et configure le constructeur du fabricant de classe. Pour davantage d'informations sur la classe de base, voir "Code généré par les experts" à la page 33-23.

- Facultativement, ajout au projet d'une bibliothèque de types et ajout de l'objet et de son interface à la bibliothèque de types.

Avant de créer un objet COM, créez ou ouvrez le projet de l'application contenant les fonctionnalités à implémenter. Le projet peut être, selon vos besoins, une application ou une bibliothèque ActiveX.

Pour afficher l'expert objet COM :

1. Choisissez Fichier | Nouveau | Autre pour ouvrir la boîte de dialogue Nouveaux éléments.
2. Sélectionnez la page ActiveX.
3. Double-cliquez sur l'icône Objet COM.

Dans l'expert, vous devez spécifier les informations suivantes :

- **Nom de CoClasse.** C'est le nom de l'objet tel qu'il apparaît aux clients. La classe créée pour implémenter votre objet porte ce nom préfixé par un 'T'. Si vous choisissez de ne pas implémenter une interface existante, l'expert attribue à votre CoClasse une interface par défaut portant ce nom préfixé par un 'I'.
- **Interface implémentée.** Par défaut, l'expert attribue à votre objet une interface par défaut qui descend de *IUnknown*. Après être sorti de l'expert, vous devez utiliser l'éditeur de bibliothèques de types pour ajouter des propriétés et méthodes à cette interface. Cependant, vous pouvez aussi sélectionner une interface prédéfinie que votre objet va implémenter. Choisissez le bouton Liste dans l'expert objet COM pour afficher l'expert de sélection d'interface qui vous permet de sélectionner parmi toutes les interfaces doubles ou personnalisées définies dans une bibliothèque de types recensée sur votre système. L'interface que vous sélectionnez devient l'interface par défaut de votre nouvelle CoClasse. L'expert ajoute toutes les méthodes de cette interface à la classe d'implémentation générée, ainsi il ne vous reste plus qu'à remplir le corps des méthodes dans l'unité d'implémentation. Si vous sélectionnez une interface existante, l'interface n'est pas ajoutée à la bibliothèque de types de votre projet. Cela signifie que lors du déploiement de l'objet, vous devez également déployer la bibliothèque de types qui définit l'interface.
- Instancie. Sauf si vous créez un serveur en processus, vous devez indiquer comment COM lance l'application qui accueille votre objet COM. Si votre application implémente plusieurs objets COM, vous devez utiliser la même méthode d'instanciation pour tous. Pour des informations sur les valeurs possibles, voir "Types d'instanciation des objets COM" à la page 36-5.
- **Modèle de thread**. Généralement, le client demande à votre objet de participer à différents threads d'exécution. Vous pouvez spécifier comment COM sérialise ces threads quand il appelle votre objet. Le choix du modèle de thread détermine comment l'objet est recensé. C'est à vous d'assurer la gestion de thread impliquée par le modèle choisi. Pour des informations sur les valeurs possibles, voir "Choix d'un modèle de thread" à la page 36-6. Pour des informations sur la manière de gérer les threads dans votre application, voir chapitre 9, "Ecriture d'applications multithreads".

- **Bibliothèque de types**. Vous pouvez décider si vous voulez inclure une bibliothèque de types pour votre objet. Cela est recommandé pour deux raisons : elle vous permet d'utiliser l'éditeur de bibliothèques de types pour définir les interfaces, et donc actualiser l'essentiel de l'implémentation, et cela donne aux clients un moyen simple d'obtenir des informations sur votre objet et ses interfaces. Si vous implémentez une interface existante, Delphi requiert l'utilisation d'une bibliothèque de types par votre projet. C'est la seule façon de fournir l'accès à la déclaration d'interface originale. Pour des informations sur les bibliothèques de types, voir "Bibliothèques de types" à la page 33-16 et chapitre 34, "Utilisation des bibliothèques de types".
- **Marshaling.** Si vous voulez vous confiner aux types compatibles Automation et si vous choisissez de créer une bibliothèque de types, vous pouvez laisser COM gérer pour vous le marshaling quand vous ne générez pas un serveur en processus. En indiquant l'interface de votre objet comme étant OleAutomation dans la bibliothèque de types, vous permettez à COM de définir pour vous les proxys et stubs et de gérer le passage de paramètres entre les frontières de processus. Pour davantage d'informations, voir "Le mécanisme du marshaling" à la page 33-8. Vous ne pouvez spécifier que l'interface est compatible Automation que si vous créez une nouvelle interface. Si vous sélectionnez une interface existante, ses attributs sont déjà spécifiés dans sa bibliothèque de types. Si l'interface de votre objet n'est pas marquée comme OleAutomation, vous devez soit créer un serveur en processus ou écrire votre propre code de marshaling.

Vous pouvez, si vous le souhaitez, ajouter une description de votre objet COM. Cette description apparaît pour l'objet dans la bibliothèques de types si vous en créez une.

Utilisation de l'expert objet Automation

L'expert objet Automation effectue les opérations suivantes :

- Création d'un nouvelle unité.
- Définition d'une nouvelle classe qui descend de *TCOMObject* et configure le constructeur du fabricant de classe. Pour davantage d'informations sur la classe de base, voir "Code généré par les experts" à la page 33-23.
- Ajout au projet d'une bibliothèque de types et ajout de l'objet et de son interface à la bibliothèque de types.

Avant de créer un objet Automation, créez ou ouvrez le projet de l'application contenant les fonctionnalités à implémenter. Le projet peut être, selon vos besoins, une application ou une bibliothèque ActiveX.

Pour afficher l'expert objet Automation :

1. Choisissez Fichier | Nouveau | Autre.
2. Sélectionnez la page ActiveX.
3. Double-cliquez sur l'icône Objet Automation.

Dans l'expert, vous devez spécifier les informations suivantes :

- **Nom de CoClasse.** C'est le nom de l'objet tel qu'il apparaît aux clients. La classe créée pour implémenter votre objet porte ce un nom préfixé par un 'T'. Si vous choisissez de ne pas implémenter une interface existante, l'expert attribue à votre CoClasse une interface par défaut portant ce nom préfixé par un 'I'.
- **Instancie**. Sauf si vous créez un serveur en processus, vous devez indiquer comment COM lance l'application qui accueille votre objet COM. Si votre application implémente plusieurs objets COM, vous devez utiliser la même méthode d'instanciation pour tous. Pour des informations sur les valeurs possibles, voir "Types d'instanciation des objets COM" à la page 36-5.
- **Modèle de thread**. Généralement, le client demande à votre objet de participer à différents threads d'exécution. Vous pouvez spécifier comment COM sérialise ces threads quand il appelle votre objet. Le choix du modèle de thread détermine comment l'objet est recensé. C'est à vous d'assurer la gestion de thread impliquée par le modèle choisi. Pour des informations sur les valeurs possibles, voir "Choix d'un modèle de thread" à la page 36-6. Pour des informations sur la manière de gérer les threads dans votre application, voir chapitre 9, "Ecriture d'applications multithreads".
- **Générer le code de support d'événement** Vous devez indiquer si vous voulez que votre objet génère des événements pour les clients pouvant y répondre. L'expert peut fournir la gestion des interfaces nécessaires pour générer des événements et la distribution des appels aux gestionnaires d'événements des clients. Pour des informations sur le fonctionnement des événements et la manière de les implémenter, voir "Exposition d'événements aux clients" à la page 36-11.

Vous pouvez, si vous le souhaitez, ajouter une description de votre objet COM. Cette description apparaît pour l'objet dans la bibliothèques de types.

L'objet Automation implémente une **interface double** qui gère la liaison précoce (à la compilation) via la VTable et la liaison tardive (à l'exécution) via l'interface *IDispatch*. Pour davantage d'informations, voir "Interfaces doubles" à la page 36-14.

Types d'instanciation des objets COM

La plupart des experts COM attendent que vous spécifiez le mode d'instanciation de l'objet. L'instanciation détermine le nombre d'instances de votre objet que les clients peuvent créer dans un seul exécutable. Par exemple, si vous spécifiez le modèle Instance unique, une fois qu'un client a instancié votre objet, COM rend l'application invisible ; ainsi d'autres clients doivent lancer leurs propres instances de l'application. Comme cela affecte la visibilité globale de votre application, le mode d'instanciation doit être le même pour tous les objets de votre application qui peuvent être instanciés par les clients. Cela signifie que vous ne pouvez pas créer un objet de l'application en utilisant le mode Instance unique alors qu'un autre de la même application utilise le mode Instance multiple.

Remarque Le mode d'instanciation est ignoré lorsque votre objet COM est uniquement utilisé en tant que serveur en processus.

Lorsque votre application COM crée un nouvel objet COM, il peut avoir un des types d'instanciation suivants :

Instanciation	Signification
Interne	L'objet ne peut être créé que de manière interne. Une application externe ne peut pas créer d'instance de l'objet directement. Une application externe ne peut pas créer directement une instance de l'objet même si votre application peut le créer et en transmettre une interface aux clients.
Instance unique	Permet aux clients de ne créer qu'une seule instance de l'objet pour chaque exécutable (application), ainsi la création de plusieurs instances entraîne le lancement de plusieurs instances de l'application. Chaque client dispose de sa propre instance dédiée de l'application serveur. Cette option est souvent utilisée pour les applications MDI (interface à documents multiples).
Instances multiples	Spécifie que plusieurs client peuvent créer des instances de l'objet dans le même espace de processus. Chaque fois qu'un client demande un service, une instance séparée du serveur est créée : c'est-à-dire qu'il peut y avoir plusieurs instances dans un seul exécutable.

Choix d'un modèle de thread

Lors de la création d'un objet à l'aide de l'expert, sélectionnez le modèle de thread que votre objet peut gérer. En ajoutant la gestion des threads à votre objet COM, vous pouvez augmenter ses performances, car plusieurs clients peuvent accéder simultanément à votre application.

Le tableau suivant énumère les différents modèles de thread que vous pouvez spécifier.

Tableau 36.1 Modèles de thread pour les objets COM

Modèle de thread	Description	Avantages et inconvénients
Single	Le serveur ne gère pas les threads. COM sérialise les demandes des clients afin que l'application ne reçoive qu'une seule demande à la fois.	Les clients sont traités un par un, il n'est donc pas nécessaire de gérer les threads. Pas de gains en performances.
Apartment (ou apartment à thread unique)	COM s'assure qu'un seul thread client peut appeler l'objet à la fois. Tous les appels des clients utilisent le thread dans lequel l'objet a été créé.	Les objets peuvent accéder en toute sécurité à leurs propres données d'instance mais les données globales doivent être protégées en utilisant des sections critiques ou une autre forme de sérialisation. Les variables locales du thread sont fiables entre plusieurs appels. Certains gains en performances.

Tableau 36.1 Modèles de thread pour les objets COM (suite)

Modèle de thread	Description	Avantages et inconvénients
Libre (également appelé apartment à threads multiples)	Les objets peuvent recevoir des appels d'un nombre quelconque de threads à tout moment.	Les objets doivent protéger toutes les données de l'instance et les données globales en utilisant des sections critiques ou une autre forme de sérialisation. Les variables locales des threads *ne* sont *pas* fiables entre plusieurs appels.
Les deux	Identique au modèle Libre mais les appels sortants (par exemple, les callbacks) sont assurés de s'exécuter dans le même thread.	Maximum de performances et de souplesse. N'impose pas à l'application de gérer les threads pour les paramètres spécifiés pour les appels sortants.
Neutre	Plusieurs clients peuvent appeler l'objet dans différents threads en même temps, mais COM s'assure que deux appels n'entrent pas en conflit.	Vous devez vous protéger des conflits de thread impliquant des données globales ou les données d'instance sont accédées par plusieurs méthodes. Ce modèle ne doit pas être utilisé avec les objets ayant une interface utilisateur (contrôles visuels). Ce modèle est disponible uniquement avec COM+. Dans COM, il est remplacé par le modèle Apartment.

Remarque Les variables locales sont toujours fiables (sauf celles des callbacks) indépendamment du modèle de thread. En effet, les variables locales sont stockées dans la pile et chaque thread dispose de sa propre pile. Les variables locales des callbacks peuvent n'être pas fiables si vous utilisez le modèle libre.

Le modèle de thread que vous choisissez dans l'expert détermine comment l'objet est publié dans le registre. Vous devez être certain que l'implémentation de l'objet est conforme au modèle de thread choisi. Pour des informations générales sur l'écriture de code adapté aux threads, voir chapitre 9, "Ecriture d'applications multithreads".

Pour les serveurs en processus, l'initialisation du modèle de thread dans l'expert définit la clé du modèle de thread dans l'entrée de registre CLSID.

Les serveurs hors processus sont recensés comme EXE, et Delphi initialise COM pour le modèle de thread le plus élevé nécessaire. Par exemple, si un EXE contient un objet à thread libre, il est initialisé pour les threads libres, ce qui signifie qu'il peut fournir la gestion attendue pour tous les objets à thread libre ou apartment contenus dans l'EXE. Pour redéfinir manuellement le comportement de thread d'un EXE, utilisez la variable*CoInitFlags* décrite dans l'aide en ligne.

Ecriture d'un objet gérant le modèle de thread libre

Utilisez le modèle de thread libre au lieu du modèle apartment chaque fois que l'objet doit être accédé par plusieurs threads. Considérons, par exemple, une application client connectée à un objet sur une machine distante. Lorsque le client distant appelle une méthode sur cet objet, le serveur reçoit l'appel sur un thread appartenant au groupe de threads de la machine serveur. Le thread recevant effectue l'appel localement sur l'objet réel et, l'objet supportant le modèle de thread libre, le thread peut effectuer un appel direct dans l'objet.

Si, au contraire, l'objet avait supporté le modèle de thread apartment, l'appel aurait dû être transféré au thread dans lequel l'objet a été créé et le résultat aurait dû être transféré en retour dans le thread recevant avant de revenir au client. Cette approche nécessite des mécanismes de marshaling supplémentaires.

Pour supporter le modèle de thread libre, vous devez considérer la façon dont *chaque* méthode accédera aux données d'instance. Si la méthode est écrite pour des données d'instance, vous devez utiliser les sections critiques, ou tout autre forme de sérialisation, pour protéger les données d'instance. En outre, la sérialisation des appels critiques est moins lourde que l'exécution du code de marshaling de COM.

Remarquez que si les données d'instance sont accessibles en lecture seulement, la sérialisation n'est pas nécessaire.

Les serveurs en processus à thread libre peuvent améliorer leurs performances en se comportant comme objet extérieur d'un agrégat avec le marshaler à thread libre. Le marshaler à thread libre propose un raccourci pour la gestion standard COM du thread quand une DLL à thread libre est appelée par un hôte (un client) qui n'utilise pas de thread libre.

Pour effectuer l'agrégation avec le marshaler à thread libre, vous devez :

- Appeler *CoCreateFreeThreadedMarshaler*, en lui transmettant l'interface *IUnknown* de votre objet qui doit utiliser le marshaler à thread libre résultant :

  ```
  CoCreateFreeThreadedMarshaler(self as IUnknown, FMarshaler);
  ```

  Cette ligne affecte l'interface du marshaler à thread libre à un membre de classe, *FMarshaler*.

- En utilisant l'éditeur de bibliothèques de types, ajoutez l'interface *IMarshal* à l'ensemble des interfaces que votre CoClasse implémente.

- Dans la méthode *QueryInterface* de votre objet, déléguez tous les appels de IDD_IMarshal au marshaler à thread libre (stocké dans *FMarshaler* ci-dessus).

Attention Le marshaler à thread libre viole les règles normales du marshaling COM pour permettre une meilleure efficacité. Il doit être utilisé avec prudence. En particulier, il ne doit être agrégé qu'avec des objets à thread libre dans des serveurs en processus et ne doit être instancié que par l'objet qui l'utilise (et pas par un autre thread).

Ecriture d'un objet supportant le modèle de thread apartment

Pour implémenter le modèle de thread apartment (à thread unique), il est nécessaire de suivre certaines règles :

- Le premier thread de l'application qui a été créé est le thread principal de COM. C'est habituellement le thread sur lequel WinMain a été appelé. Ce doit être également le dernier thread à désinitialiser COM.
- Chaque thread du modèle de thread apartment doit posséder une boucle de messages et la file des messages doit être vérifiée fréquemment.
- Lorsqu'un thread obtient un pointeur sur une interface COM, les méthodes de cette interface ne peuvent être appelées que depuis ce thread.

Le modèle apartment à thread unique est un moyen terme entre la solution qui consiste à ne fournir aucun support des threads et celle qui assure le support complet du multi-thread dans le modèle libre. Un serveur conforme au modèle apartment est la garantie que le serveur sérialise les accès à toutes ses données globales (par exemple, son nombre d'objets). En effet, différents objets peuvent tenter d'accéder aux données globales depuis différents threads. Cependant, les données des instances de l'objet sont sécurisées, car les méthodes sont toujours appelées sur le même thread.

Habituellement, les contrôles destinés aux navigateurs Web utilisent le modèle de thread apartment, car les applications navigateur initialisent toujours leurs threads en tant qu'apartment.

Ecriture d'un objet supportant le modèle de thread neutre

Avec COM+, vous pouvez utiliser un autre modèle de thread, à mi-chemin entre thread libre et thread apartment : le modèle neutre. Comme le modèle de thread libre, ce modèle permet à plusieurs threads d'accéder simultanément à votre objet. Il n'y a pas de marshaling supplémentaire pour transférer au thread dans lequel l'objet a été créé. Cependant votre objet est assuré de ne pas recevoir d'appels contradictoires.

L'écriture d'un objet utilisant le modèle de thread neutre respecte sensiblement les même règles que l'écriture d'un objet gérant le modèle de thread apartment, sauf qu'il n'est pas nécessaire de protéger les données de l'instance des conflits de threads, elles peuvent être accédées par différentes méthodes de l'interface de l'objet. Toutes les données d'instance qui ne peuvent être accédées que par une seule méthode d'interface sont automatiquement adaptées aux threads.

Définition de l'interface d'un objet COM

Quand vous utilisez un expert pour créer un objet COM, l'expert crée automatiquement une bibliothèque de types (sauf si vous avez demandé le contraire dans l'expert objet COM). La bibliothèque de types permet aux applications hôte de déterminer ce que l'objet peut faire. Elle vous permet également de définir l'interface de l'objet en utilisant l'éditeur de bibliothèques de types. Les interfaces que vous définissez dans l'éditeur de bibliothèques de

types définissent les propriétés, méthodes et événements que votre objet expose aux clients.

Remarque Si vous sélectionnez une interface existante dans l'expert objet COM, vous n'avez pas besoin d'ajouter des propriétés ou des méthodes. La définition de l'interface est importée depuis la bibliothèque de types dans laquelle l'interface est définie. Il vous suffit alors de trouver les méthodes de l'interface importée dans l'unité d'implémentation et de remplir leur corps.

Ajout d'une propriété à l'interface de l'objet

Quand vous ajoutez une propriété à l'interface de votre objet en utilisant l'éditeur de bibliothèques de types, il ajoute automatiquement une méthode pour lire la valeur de la propriété et/ou une méthode pour définir la valeur de la propriété. L'éditeur de bibliothèques de types, ajoute à son tour ces méthodes à votre classe d'implémentation et crée dans l'unité d'implémentation des squelettes de méthodes que vous devez compléter.

Pour ajouter une propriété à l'interface de votre objet

1. Dans l'éditeur de bibliothèques de types, sélectionnez l'interface par défaut de l'objet.

   L'interface par défaut porte le même nom que l'objet mais préfixé par la lettre "I". Pour déterminer l'interface par défaut, dans l'éditeur de bibliothèques de types, choisissez la CoClasse, sélectionnez la page Implémente et recherchez dans la liste des interfaces implémentées celle indiquée comme "Défaut".

2. Pour exposer une propriété en lecture/écriture, cliquez sur le bouton Propriété de la barre d'outils ; vous pouvez aussi cliquer sur la flèche à côté de ce bouton dans la barre d'outils, puis sur le type de propriété à exposer.

3. Dans la page Attributs, spécifiez le nom et le type de la propriété.

4. Dans la barre d'outils, choisissez le bouton Rafraîchir.

   Une définition et les squelettes d'implémentation des méthodes d'accès à la propriété sont insérés dans l'unité d'implémentation de l'objet.

5. Dans l'unité d'implémentation, recherchez les méthodes d'accès de la propriété. Elles ont un nom de la forme Get_NomPropriété et Set_NomPropriété. Ajoutez du code qui récupère ou définit la valeur de la propriété de votre objet. Ce code peut simplement appeler une fonction existante de l'application, accéder à une donnée membre ajoutée à la définition de l'objet ou implémenter autrement la propriété.

Ajout d'une méthode à l'interface de l'objet

Quand vous ajoutez une méthode à l'interface de votre objet en utilisant l'éditeur de bibliothèques de types, il ajoute les méthodes à la classe d'implémentation et crée dans l'unité d'implémentation des squelettes de méthode à compléter.

Pour exposer une méthode via l'interface de votre objet :

1. Dans l'éditeur de bibliothèques de types, sélectionnez l'interface par défaut de l'objet.

   L'interface par défaut porte le même nom que l'objet mais préfixé par la lettre "I". Pour déterminer l'interface par défaut, dans l'éditeur de bibliothèques de types, choisissez la CoClasse, sélectionnez la page Implémente et recherchez dans la liste des interfaces implémentées celle indiquée comme "Défaut."
2. Choisissez le bouton Méthode.
3. Dans la page Attributs, spécifiez le nom de la méthode.
4. Dans la page Paramètres, spécifiez le type de valeur renvoyée par la méthode et ajoutez les paramètres appropriés.
5. Dans la barre d'outils, choisissez le bouton Rafraîchir.
6. Une définition et le squelette de l'implémentation de la méthode sont insérés dans l'unité d'implémentation de l'objet.
7. Dans l'unité d'implémentation, trouvez la méthode qui vient d'être insérée. La méthode est entièrement vide. Remplissez le corps de la méthode pour effectuer l'opération que représente la méthode.

Exposition d'événements aux clients

Il y a deux types d'événements qu'un objet COM peut générer : les événements classiques et les événements COM+.

- Les événements COM+ nécessitent la création d'un objet événement séparé en utilisant l'expert objet événement et l'ajout de code pour appeler cet objet événement depuis votre objet serveur. Pour davantage d'informations sur la génération d'objets événement, voir "Génération d'événements dans COM+" à la page 39-20.
- Vous pouvez utiliser l'expert pour gérer l'essentiel du travail de création des événements classiques. Ce processus est décrit ci-dessous.

Remarque L'expert objet COM ne génère pas de code de gestion des événements. Si vous voulez que votre objet génère des événements classiques, utilisez l'expert objet Automation.

Pour que votre objet génère des événements, vous devez effectuer les opérations suivantes :

1. Dans l'expert Automation, cochez la case Insérer le code de support d'événement.

L'expert crée un objet qui contient une interface Events en plus de l'interface par défaut. Cette interface Events a un nom de la forme I*NomCoClasse*Events. C'est une interface en sortie (source) : cela signifie que cette interface ne doit pas être implémentée par votre objet mais par les clients, elle est appelée par votre objet (vous pouvez le constater en sélectionnant la CoClasse, allez sur la page Implémente, vous constaterez que la colonne Source indique *True* pour l'interface Events).

Outre l'interface Events, l'expert ajoute l'interface *IConnectionPointContainer* à la déclaration de votre classe d'implémentation et plusieurs membres de classe pour gérer les événements. Parmi ces nouveaux membres de classe, les plus importants sont *FConnectionPoint* et *FConnectionPoints*. Ils implémentent les interfaces *IConnectionPoint* et *IConnectionPointContainer* en utilisant des classes VCL prédéfinies. *FConnectionPoint* est gérée par une autre méthode que l'expert ajoute, *EventSinkChanged*.

2. Dans l'éditeur de bibliothèques de types, sélectionnez l'interface en sortie Events de votre objet (elle a un nom de la forme I*NomCoClasse*Events)
3. Cliquez sur le bouton Méthode de la barre d'outils de l'éditeur de bibliothèques de types. Chaque méthode ajoutée à l'interface Events représente un gestionnaire d'événement que le client doit implémenter.
4. Dans la page Attributs, spécifiez le nom du gestionnaire d'événement, par exemple MonEvenement.
5. Dans la barre d'outils, choisissez le bouton Rafraîchir.

   L'implémentation de votre objet contient maintenant tout ce qu'il faut pour accepter les récepteurs d'événements des clients et gérer une liste des interfaces à appeler quand l'événement se produit. Pour appeler ces interfaces, vous pouvez créer une méthode pour générer chaque événement dans les clients.
6. Dans l'éditeur de code, ajoutez une méthode à votre objet pour déclencher chaque événement. Par exemple :

```
unit ev;
interface
uses
   ComObj, AxCtrls, ActiveX, Project1_TLB;
type
   TMyAutoObject = class (TAutoObject,IConnectionPointContainer, IMyAutoObject)
private
   .
   .
   .
public
   procedure Initialize; override;
   procedure Fire_MonEvenement; { Ajoutez une méthode pour déclencher l'événement}
```

7 Codez la méthode ajoutée à l'étape précédente afin qu'elle parcourt tous les récepteurs d'événements gérés dans le membre *FConnectionPoint* de votre objet :

```
procedure TMyAutoObject.Fire_MonEvenement;
var
  I: Integer;
  EventSinkList: TList;
  EventSink: IMyAutoObjectEvents;
begin
  if FConnectionPoint <> nil then
  begin
    EventSinkList :=FConnectionPoint.SinkList; {obtenir la liste des récepteurs client }
    for I := 0 to EventSinkList.Count - 1 do
    begin
      EventSink := IUnknown(FEvents[I]) as IMyAutoObjectEvents;
      EventSink.MyEvent;
    end;
  end;
end;
```

8 A chaque fois que vous avez besoin de déclencher l'événement pour que les clients soient informés de son occurrence, appelez la méthode qui distribue l'événement à tous les récepteurs d'événements :

```
if EventOccurs then Fire_MonEvenement;
    { Appelez la méthode créée pour déclencher des événements.}
```

Gestion des événements dans un objet Automation

Pour qu'un serveur gère les événements COM classiques, il doit proposer une définition d'une interface de sortie implémentée par un client. Cette interface en sortie contient tous les gestionnaires d'événements que le client doit implémenter pour répondre aux événements du serveur.

Quand un client a implémenté l'interface événement en sortie, il informe de son souhait d'être notifié des événements en interrogeant l'interface *IConnectionPointContainer* du serveur. L'interface *IConnectionPointContainer* renvoie l'interface *IConnectionPoint* du serveur qui est utilisée par le client comme pointeur pour son implémentation des gestionnaires d'événements (c'est que l'on appelle un récepteur).

Le serveur gère une liste de tous les récepteurs des clients et appelle leurs méthodes quand un événement se produit, comme décrit ci-dessous.

Interfaces d'Automation

L'expert objet Automation implémente par défaut une interface double ce qui signifie que l'objet Automation gère :

- La liaison tardive à l'exécution, via son interface *IDispatch*. Elle est implémentée comme une interface de répartition ou **dispinterface**.

- La liaison précoce à la compilation, ce qui s'effectue directement en appelant l'une des fonctions membre de la table de fonction virtuelle (VTable). C'est ce que l'on appelle une **interface personnalisée**.

Remarque Les interfaces générées par l'expert objet COM qui ne descendent pas de *IDispatch* ne gèrent que les appels de la VTable.

Interfaces doubles

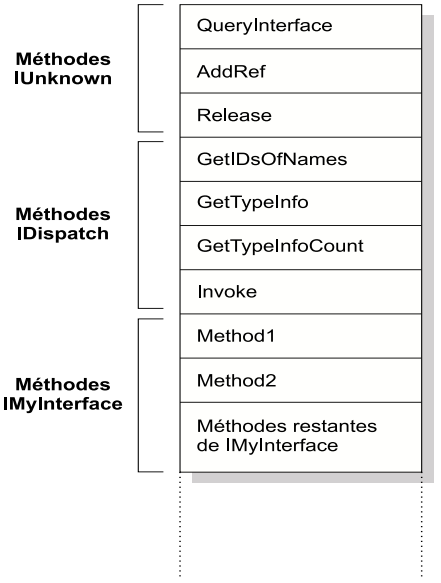
Une interface double est à la fois une interface personnalisée et une dispinterface. Elle est implémentée en tant qu'interface VTable COM qui dérive de *IDispatch*. Pour les contrôleurs qui accèdent à l'objet uniquement à l'exécution, la dispinterface est disponible. Pour les objets qui peuvent tirer profit de la liaison à la compilation, c'est l'interface VTable, la plus efficace, qui est utilisée.

Les interfaces doubles offrent les avantages combinés des interfaces VTable et des dispinterfaces :

- Pour les interfaces VTable, le compilateur fait une vérification de type et fournit des messages d'erreurs plus informatifs.
- Pour les contrôleurs Automation qui ne peuvent pas obtenir d'information de type, la dispinterface fournit l'accès à l'objet à l'exécution.
- Pour les serveurs en processus, vous bénéficiez d'un accès rapide via les interfaces VTable.
- Pour les serveurs hors processus, COM effectue le marshaling des données à la fois pour les interfaces VTable et pour les dispinterfaces. COM fournit une implémentation proxy/stub générique qui peut réaliser le marshaling de l'interface en fonction des informations contenues dans une bibliothèque de types. Pour davantage d'informations sur le marshaling, voir "Marshaling des données" à la page 36-16.

Le diagramme page suivante décrit l'interface *IMyInterface* d'un objet gérant une interface double nommée *IMyInterface.* Les trois premières entrées de la VTable d'une interface double font référence à l'interface *IUnknown*, les quatre suivantes font référence à l'interface *IDispatch*, les autres sont des entrées COM pour l'accès direct aux membres de l'interface personnalisée.

Figure 36.1 VTable d'une interface double



Interfaces de répartition

Les contrôleurs Automation sont des clients qui utilisent l'interface COM *IDispatch* pour accéder aux objets du serveur COM. Le contrôleur doit d'abord créer l'objet, puis demander à l'interface *IUnknown* de l'objet un pointeur sur son interface *IDispatch*. *IDispatch* garde en interne la trace des méthodes et des propriétés par le biais d'un identificateur de répartition (dispID), qui est un numéro d'identification propre à chaque membre interface. Grâce à *IDispatch*, un contrôleur récupère les informations de type de l'objet pour l'interface de répartition, puis associe les noms des membres interface aux dispID spécifiques. Ces dispID sont accessibles à l'exécution et les contrôleurs y accèdent en appelant la méthode *GetIDsOfNames* de IDispatch.

Lorsqu'il connaît le dispID, le contrôleur peut appeler la méthode *Invoke* de *IDispatch* pour exécuter le code requis (propriété ou méthode), en regroupant les paramètres de la propriété ou de la méthode dans l'un des paramètres de la méthode *Invoke*. *Invoke* a une signature fixe définie lors de la compilation, qui lui permet d'accepter des arguments variés lors de l'appel d'une méthode d'interface.

L'implémentation de l'objet automation de *Invoke* doit alors dégrouper les paramètres, appeler la propriété ou la méthode et gérer les éventuelles erreurs. Lorsque la propriété ou la méthode revient, l'objet retransmet la valeur renvoyée au contrôleur.

Cette procédure est appelée liaison différée car le contrôleur se lie à la propriété ou à la méthode lors de l'exécution de l'application plutôt que lors de sa compilation.

Remarque Lors de l'importation d'une bibliothèque de types, Delphi requiert des dispID lors de la génération du code, permettant de cette façon aux classes enveloppe générées d'appeler *Invoke* sans appeler *GetIDsOfNames*. Cela peut augmenter de manière significative les performances d'exécution des contrôleurs.

Interfaces personnalisées

Les interfaces personnalisées sont des interfaces définies par l'utilisateur qui permettent aux clients d'appeler les méthodes de l'interface en fonction de leur ordre dans la VTable et du type des arguments. La VTable contient les adresses de toutes les propriétés et méthodes qui sont membres de l'objet, y compris les fonctions membre des interfaces qu'il supporte. Si l'objet ne supporte pas *IDispatch*, les entrées correspondant aux membres des interfaces personnalisées de l'objet suivent immédiatement celles des membres de *IUnknown*.

Si l'objet possède une bibliothèque de types, vous pouvez accéder à l'interface personnalisée via sa disposition VTable, que vous pouvez obtenir à l'aide de l'éditeur de bibliothèques de types. Si l'objet possède une bibliothèque de types et gère *IDispatch*, un client peut aussi obtenir les dispID de l'interface *IDispatch* et se lier directement à un déplacement de la VTable. L'importateur de la bibliothèque de types de Delphi (TLIBIMP) extrait les dispIDs à l'importation, ce qui évite aux clients qui utilisent les enveloppes de dispinterfaces d'appeler *GetIDsOfNames*; ces informations figurent déjà dans le fichier _TLB. Toutefois, les clients doivent quand même appeler *Invoke*.

Marshaling des données

Pour les serveurs hors processus et distants, vous devez prendre en compte la façon dont COM effectue le marshaling des données hors du processus en cours. Vous pouvez effectuer le marshaling :

- Automatiquement, en utilisant l'interface *IDispatch*.
- Automatiquement, en créant une bibliothèque de types avec votre serveur et en marquant l'interface avec le drapeau d'Automation OLE. COM sait comment réaliser le marshaling de tous les types **compatibles Automation** de la bibliothèque de types et peut implémenter les proxys et les stubs pour vous. Certaines restrictions de types sont nécessaires pour permettre le marshaling automatique.
- Manuellement, en implémentant toutes les méthodes de l'interface IMarshal. Cela s'appelle le **marshaling personnalisé**.

Remarque La première méthode, l'utilisation de *IDispatch* est uniquement disponible sur les serveurs Automation. La seconde méthode est automatiquement disponible sur tous les objets créés par les experts et utilisant une bibliothèque de types.

Types compatibles avec l'Automation

Les résultats de fonctions et les types des paramètres des méthodes déclarés dans les interfaces duales et de répartition doivent être des types *compatibles Automation*. Les types suivants sont compatibles Automation OLE :

- Les types autorisés prédéfinis, comme *Smallint*, *Integer*, *Single*, *Double et WideString*. Pour la liste complète, voir "Types autorisés" à la page 34-13.
- Les types énumération définis dans une bibliothèque de types. Les types énumération compatibles Automation OLE sont stockés dans des valeurs 32 bits et sont traités comme des valeurs de type *Integer* pour la transmission des paramètres.
- Les types interface définis dans une bibliothèque de types qui sont sécurisés Automation OLE, c'est-à-dire dérivés de *IDispatch* et contenant seulement des types compatibles Automation OLE.
- Les types dispinterface définis dans une bibliothèque de types.
- Tout type enregistrement personnalisé défini dans la bibliothèque de types.
- *IFont*, *IStrings* et *IPicture*. Les objets utilitaires doivent être instanciés pour que correspondent
  - un *IFont* à un *TFont*
  - un *IStrings* à un *TStrings*
  - un *IPicture* à un *TPicture*

  Les experts contrôle ActiveX et ActiveForm créent automatiquement ces objets utilitaires lorsque c'est nécessaire. Pour utiliser les objets utilitaires, appelez respectivement les routines globales GetOleFont, GetOleStrings, GetOlePicture.

Restrictions de type pour le marshaling automatique

Pour qu'une interface supporte le marshaling automatique (aussi appelé marshaling Automation ou marshaling de bibliothèque de types), les restrictions suivantes sont nécessaires. Lorsque vous modifiez votre objet Automation en utilisant l'éditeur de bibliothèques de types, celui-ci applique les restrictions suivantes :

- Les types doivent être compatibles pour la communication interplate-forme. Par exemple, vous ne pouvez pas utiliser de structure de données (autrement qu'en implémentant un autre objet propriété), ni d'argument non signé, ni d'AnsiStrings, etc.
- Les types chaîne doivent être transférés en tant que chaînes larges (BSTR). PChar et AnsiString ne peuvent pas subir de marshaling en toute sécurité.
- Tous les membres d'une interface double doivent passer un HRESULT comme valeur de retour de fonction. Si la méthode est déclarée par le biais de la convention d'appel safecall, cette condition est imposée automatiquement, avec le type de retour déclaré converti en paramètre de sortie.

- Les membres d'une interface duale qui ont besoin de renvoyer d'autres valeurs doivent spécifier ces paramètres en tant que **var** ou **out**, en indiquant un paramètre de sortie pour renvoyer la valeur de la fonction.

Remarque Une façon d'outrepasser les restrictions des types Automation est d'implémenter une interface *IDispatch* distincte et une interface personnalisée. Vous pouvez ainsi utiliser tous les types d'arguments. Cela signifie que les clients COM ont la possibilité d'utiliser l'interface personnalisée, à laquelle les contrôleurs Automation peuvent encore accéder. Mais, dans ce cas, il faut implémenter manuellement le code de marshaling.

Marshaling personnalisé

Généralement, vous utiliserez le marshaling automatique dans les serveurs hors processus et distants parce que c'est le plus simple : COM fait le travail à votre place. Cependant, vous pouvez décider de fournir un marshaling personnalisé si vous pensez que ses performances seront supérieures. Si sous implémentez votre propre marshaling personnalisé, vous devez gérer l'interface *IMarshal*. Pour davantage d'informations sur cette approche, voir la documentation Microsoft.

Recensement d'un objet COM

Un objet serveur peut être recensé comme serveur en ou hors processus. Pour davantage d'informations sur ces types de serveur, voir "Serveurs en processus, hors processus et distants" à la page 33-7.

Remarque Pour retirer l'objet COM de votre système, vous devez commencer par annuler son recensement.

Recensement d'un serveur en processus

Pour recenser un serveur en processus (DLL ou OCX) :

- Choisissez Exécuter | Recenser le serveur ActiveX.

Pour annuler le recensement d'un serveur en processus :

- Choisissez Exécuter | Dérecenser le serveur ActiveX.

Recensement d'un serveur hors processus

Pour recenser un serveur hors processus :

- Exécutez le serveur avec l'option de ligne de commande **/regserver**.

  Pour spécifier les options de la ligne de commande, utilisez la boîte de dialogue Exécuter | Paramètres.

  Vous pouvez également recenser le serveur en l'exécutant.

Pour annuler le recensement d'un serveur hors processus :

- Exécutez le serveur avec l'option de ligne de commande /**unregserver**.

Vous pouvez également utiliser la commande **tregsvr** depuis la ligne de commande ou exécuter regsvr32.exe depuis le système d'exploitation.

Remarque Si le serveur COM doit fonctionner dans COM+, vous devez l'installer dans une application COM+ au lieu de le recenser (l'installation de l'objet dans une application COM+ se charge automatiquement du recensement). Pour des informations sur la manière d'installer un objet dans une application COM+, voir "Installation d'objets transactionnels" à la page 39-24.

Test et débogage de l'application

Pour tester et déboguer une application serveur COM :

1. Si nécessaire, activez les informations de débogage en utilisant la page Compilateur de la boîte de dialogue Projet | Options. Activez aussi Débogage intégré dans le dialogue Outils | Options du débogueur.
2. Pour un serveur en processus, choisissez Exécuter | Paramètres, entrez le nom du contrôleur Automation dans la zone Application hôte, puis choisissez OK.
3. Choisissez Exécuter | Exécuter.
4. Définissez des points d'arrêts dans le serveur Automation.
5. Utilisez le contrôleur Automation pour interagir avec le serveur Automation.

Le serveur Automation fait une pause quand les points d'arrêts sont rencontrés.

Remarque Comme autre approche, si vous écrivez aussi le contrôleur Automation, vous pouvez déboguer dans un serveur en processus en activant le support inter-processus COM. Utilisez la page Général de la boîte de dialogue obtenue par Outils | Options du débogueur pour activer le support inter-processus.

Chapitre

37

Création d'une page Active Server

Si vous utilisez l'environnement Microsoft Internet Information Server (IIS) pour proposer vos pages Web, vous pouvez utiliser ASP (Active Server Pages) pour créer des applications client-serveur Web dynamiques. ASP vous permet d'écrire un script qui est appelé à chaque fois que le serveur charge la page Web. Ce script peut, à son tour, appeler des objets Automation pour obtenir des informations qu'il peut inclure dans une page HTML générée. Vous pouvez, par exemple, écrire un serveur Automation Delphi pour se connecter à une base de données et utiliser ce contrôle pour accéder aux données qui sont ainsi actualisées à chaque fois que le serveur charge la page Web.

Pour le client, ASP apparaît comme un document standard HTML qui peut être visualisé par les utilisateurs de toute plate-forme disposant d'un navigateur Web.

Les applications ASP sont similaires aux applications écrites en utilisant la technologie courtier Web Delphi. Pour davantage d'informations sur la technologie courtier Web, voir chapitre 27, "Création d'applications Internet". Cependant ASP est différent car il sépare la conception de l'interface utilisateur de l'implémentation des règles de gestion et de la logique complexe d'une application.

- La conception de l'interface utilisateur est gérée par la page Active Server. C'est essentiellement un document HTML, mais elle peut contenir des scripts incorporés qui appellent des objets ASP pour fournir au document le contenu qui reflète les règles de gestion ou la logique de l'application.
- La logique de l'application est encapsulée par des objets Active Server qui exposent des méthodes simples à la page Active Server en lui fournissant le contenu dont elle a besoin.

Remarque Même si ASP présente l'avantage de séparer la conception de l'interface utilisateur de la logique de l'application, à grande échelle ses performances sont limitées. Pour des sites Web devant satisfaire un très grand nombre de clients, il est recommandé d'utiliser à la place une approche exploitant la technologie courtier Web.

Les scripts des pages Active Server et les objets Automation incorporés dans une page peuvent utiliser les éléments intrinsèques ASP (des objets prédéfinis qui fournissent des informations sur l'application en cours, les messages HTTP du navigateur, etc).

Ce chapitre décrit comment créer un objet Active Server en utilisant l'expert objet Active Server Delphi. Ce contrôle Automation spécial peut ensuite être appelé par une page Active Server et lui fournir son contenu.

Voici les étapes de la création d'un objet Active Server :

- Création d'un objet Active Server pour l'application.
- Définition de l'interface d'un objet Active Server.
- Recensement d'un objet Active Server.
- Test et débogage d'une application ASP.

Création d'un objet Active Server

Un objet Active Server est un objet Automation qui accède à toutes les informations de l'application ASP et aux messages HTTP qu'elle utilise pour communiquer avec les navigateurs. Il descend de *TASPObject* ou de *TASPMTSObject*(qui à son tour descend de *TAutoObject*), et il gère les protocoles d'Automation, en s'exposant pour être utilisé par d'autres applications ou par le script dans la page Active Server. Pour créer un objet Active Server, utilisez l'expert Objet Active Server.

Votre projet d'objet Active Server peut prendre, selon vos besoins, la forme d'un exécutable (exe) ou d'une bibliothèque (dll). Cependant vous devez tenir compte des inconvénients liés à l'utilisation d'un serveur hors processus. Ces inconvénients sont présentés dans la section "Création d'ASP pour des serveurs en et hors processus" à la page 37-8.

Pour afficher l'expert d'objet Active Server :

1 Choisissez Fichier | Nouveau | Autre.

2 Sélectionnez l'onglet ActiveX.

3 Double-cliquez sur l'icône Objet Active Server.

Dans l'expert, spécifiez le nom du nouvel objet Active Server et spécifiez les modèles et de thread qu'il faut gérer. Ces paramètres déterminent comment l'objet peut être appelé. Vous devez écrire l'implémentation de telle manière qu'elle soit conforme au modèle (par exemple, en évitant les conflits de thread). Les modèles d'instanciation et de thread impliquent les mêmes choix que pour les autres objets COM. Pour davantage d'informations, voir "Types d'instanciation des objets COM" à la page 36-5 et "Choix d'un modèle de thread" à la page 36-6.

Ce qui rend unique l'objet Active Server est sa capacité à accéder aux informations relatives à l'application ASP et aux messages HTTP échangés entre la page Active Server et les navigateurs client Web. L'accès à ces informations se fait via les éléments intrinsèques ASP.

Vous pouvez, dans l'expert, spécifier comment votre objet y accède en spécifiant l'option Type d'Active Server :

- Si vous utilisez IIS 3 ou IIS 4, vous pouvez utiliser Méthodes d'événement de niveau page. Avec ce modèle, votre objet implémente les méthodes *OnStartPage* et *OnEndPage* appelées au chargement/déchargement de la page Active Server. Quand l'objet est chargé, il obtient automatiquement une interface *IScriptingContext* qu'il utilise pour accéder aux éléments intrinsèques ASP. Ces interfaces sont à leur tour reflétées sous la forme de propriétés héritées de la classe de base (*TASPObject*).
- Si vous utilisez IIS5 ou plus, utilisez le type Contexte d'objet. Avec ce modèle, votre objet obtient une interface *IObjectContext* qu'il utilise pour accéder aux éléments intrinsèques ASP. Là encore, ces interfaces sont à leur tour reflétées sous la forme de propriétés héritées de la classe de base (TASPMTSObject). L'avantage de cette approche tient à ce que l'objet accède à tous les autres services proposés par *IObjectContext*. Pour accéder à l'interface *IObjectContext*, appelez simplement *GetObjectContext* (définie dans l'unité mtx) de la manière suivante :

  ```
  ObjectContext := GetObjectContext;
  ```

  Pour davantage d'informations sur les services accessibles via *IObjectContext*, voir chapitre 39, "Création d'objets MTS ou COM+".

Vous pouvez demander à l'expert de générer une page .ASP simple pour accueillir le nouvel objet Active Server. La page générée contient un script minimaliste (écrit en VBScript) qui crée votre objet Active Server à partir de son ProgID et indique où vous pouvez appeler ses méthodes. Ce script appelle **Server.CreateObject** pour démarrer votre objet Active Server.

Remarque Même si le script de test généré utilise VBScript, une page Active Server peut également contenir du code Jscript.

Lorsque vous sortez de l'expert, une nouvelle unité est ajoutée au projet en cours ; elle contient la définition de l'objet Active Server. De plus, l'expert ajoute un projet de bibliothèque de types et ouvre l'éditeur de bibliothèque de types. Vous pouvez alors exposer les propriétés et méthodes de l'interface via la bibliothèque de type comme indiqué dans "Définition de l'interface d'un objet COM" à la page 36-9. Quand vous écrivez l'implémentation des propriétés et méthodes de votre objet, vous pouvez tirer profit des éléments intrinsèques ASP (décrit ci-dessous) pour obtenir des informations sur l'application ASP et les messages HTTP qu'elle utilise pour communiquer avec les navigateurs.

Comme tout autre objet Automation, l'objet Active Server implémente une **interface double**, qui gère aussi bien la liaison précoce (à la compilation) via la VTable et la liaison tardive (à l'exécution) via l'interface *IDispatch*. Pour davantage d'informations sur les interfaces doubles, voir "Interfaces doubles" à la page 36-14.

Utilisation des éléments intrinsèques ASP

Les objets intrinsèques ASP sont un ensemble d'objets COM fournis par ASP aux objets qui s'exécutent dans une page Active Server. Ils permettent à votre objet Active Server d'accéder aux informations reflétant les messages échangés entre votre application et le navigateur Web ainsi qu'un moyen de stocker des informations partagées par tous les objets Active Server appartenant à une même application ASP.

Pour simplifier l'accès à ces objets, la classe de base de votre objet Active Server les présente sous la forme de propriétés. Pour une description complète de ces objets, reportez-vous à la documentation Microsoft. Les rubriques suivantes vous en donnent une présentation rapide.

Application

On accède à l'objet Application via une interface *IApplicationObject*. Il représente toute l'application ASP, définie comme étant l'ensemble de tous les fichiers .asp d'un même répertoire virtuel et de ses sous-répertoires. L'objet Application peut être partagé par plusieurs clients, il propose donc une gestion du verrouillage que vous devez utiliser pour empêcher des conflits de threads.

IApplicationObject propose les membres suivants :

Tableau 37.1 Membres de l'interface IApplicationObject

Propriété, méthode ou événement	Signification
Propriété Contents	Enumère tous les objets ajoutés à l'application en utilisant les commandes de script. Cette interface a deux méthodes *Remove* et *RemoveAll* que vous pouvez utiliser pour supprimer un objet de la liste ou bien tous les objets.
Propriété StaticObjects	Enumère tous les objets ajoutés à l'application avec la balise <OBJECT>.
Méthode Lock	Empêche d'autres clients de verrouiller l'objet Application jusqu'à l'appel de Unlock. Tous les clients doivent appeler Lock avant d'accéder à la mémoire partagée (par exemple, les propriétés).
Méthode Unlock	Libère le verrou placé en utilisant la méthode Lock.
Evénement Application_OnEnd	Se produit à la sortie de l'application, après l'événement Session_OnEnd. Les seuls éléments intrinsèques alors disponibles sont Application et Server. Ce gestionnaire d'événement doit être écrit en VBScript ou en JScript.
Evénement Application_OnStart	Se produit avant la création d'une nouvelle session (avant Session_OnStart). Les seuls éléments intrinsèques alors disponibles sont Application et Server. Ce gestionnaire d'événement doit être écrit en VBScript ou en JScript.

Request

On accède à l'objet Request via une interface *IRequest*. Il donne des informations sur le message de requête HTTP qui a entraîné l'ouverture de la page Active Server.

IRequest propose les membres suivants :

Tableau 37.2 Membres de l'interface IRequest

Propriété, méthode ou événement	Signification
Propriété ClientCertificate	Indique la valeur de tous les champs du certificat client qui a été envoyé avec le message HTTP.
Propriété Cookies	Indique la valeur de tous les en-têtes de cookie du message HTTP.
Propriété Form	Indique la valeur des éléments de fiche dans le corps HTTP. Il est possible d'y accéder par nom.
Propriété QueryString	Indique la valeur de toutes les variables de la chaîne de requête de l'en-tête HTTP.
Propriété ServerVariables	Indique la valeur de diverses variables d'environnement. Ces variables représentent la plupart des variables courantes d'un en-tête HTTP.
Propriété TotalBytes	Indique le nombre d'octets dans le corps de la requête. C'est la limite supérieure du nombre d'octets renvoyés par la méthode BinaryRead.
Méthode BinaryRead	Récupère le contenu d'un message Post. Appelez la méthode en spécifiant le nombre maximum d'octets à lire. Le contenu résultant est renvoyé dans un tableau Variant d'octets. Après avoir appelé BinaryRead vous ne pouvez plus utiliser la propriété Form.

Response

On accède à l'objet Response via une interface *IResponse*. Il vous permet de spécifier des informations sur le message de réponse HTTP renvoyé au navigateur client.

IResponse propose les membres suivants :

Tableau 37.3 Membres de l'interface IResponse

Propriété, méthode ou événement	Signification
Propriété Cookies	Détermine la valeur des tous les en-têtes de cookie du message HTTP.
Propriété Buffer	Indique si la sortie de la page est placée dans un tampon. Si c'est le cas, le serveur n'envoie la réponse au client qu'après avoir traité tous les scripts serveur de la page en cours.
Propriété CacheControl	Détermine si des serveurs proxy peuvent placer dans un cache la sortie de la réponse.
Propriété Charset	Ajoute le nom du jeu de caractères à l'en-tête de type de contenu.

Tableau 37.3 Membres de l'interface IResponse (suite)

Propriété, méthode ou événement	Signification
Propriété ContentType	Spécifie le type de contenu HTTP du corps du message de réponse.
Propriété Expires	Spécifie combien de temps la réponse peut rester dans le cache d'un navigateur avant d'expirer.
Propriété ExpiresAbsolute	Spécifie l'heure et la date d'expiration de la réponse.
Propriété IsClientConnected	Indique si le client n'est plus connecté au serveur.
Propriété Pics	Définit la valeur du champ pics-label de l'en-tête de réponse.
Propriété Status	Indique l'état de la réponse. C'est la valeur d'un en-tête de status HTTP.
Méthode AddHeader	Ajoute un en-tête HTTP de nom et de valeur spécifiés.
Méthode AppendToLog	Ajoute une chaîne à la fin de l'entrée de l'historique du serveur Web pour cette requête.
Méthode BinaryWrite	Ecrit des informations brutes (non interprétées) dans le corps du message de réponse.
Méthode Clear	Efface tout le HTML placé dans le tampon.
Méthode End	Arrête le traitement du fichier .asp et renvoie le résultat en cours.
Méthode Flush	Envoie immédiatement tout ce que contient le tampon de sortie.
Méthode Redirect	Envoie un message de réponse de redirection qui renvoie le navigateur client sur une autre URL.
Méthode Write	Ecrit, sous forme de chaîne, une variable dans la sortie HTTP en cours.

Session

On accède à l'objet Session via une interface *ISessionObject*. Il vous permet de stocker des variables qui subsistent durant toute l'interaction d'un client avec l'application ASP. Ces variables ne sont donc pas libérées quand le client passe d'une page à une autre dans l'application ASP mais uniquement quand le client sort de l'application.

ISessionObject propose les membres suivants :

Tableau 37.4 Membres de l'interface ISessionObject

Propriété, méthode ou événement	Signification
Propriété Contents	Enumère tous les objets ajoutés à la session en utilisant la balise <OBJECT>. Vous pouvez accéder à toute variable de la liste par son nom ou appeler les méthodes *Remove* ou *RemoveAll* de l'objet Contents pour supprimer des valeurs.
Propriété StaticObjects	Enumère tous les objets ajoutés à la session avec la balise <OBJECT>.

Tableau 37.4 Membres de l'interface ISessionObject (suite)

Propriété, méthode ou événement	Signification
Propriété CodePage	Spécifie le code de page à utiliser pour mettre en correspondance les symboles. Différentes localisations peuvent utiliser des pages de code différentes.
Propriété LCID	Spécifie l'identificateur de localisation à utiliser pour interpréter des contenus chaîne.
Propriété SessionID	Indique l'identificateur de session du client en cours.
Propriété Timeout	Spécifie la durée (en minutes) durant laquelle la session subsiste sans une requête (ou une réactualisation) du client avant la fin de l'application.
Méthode Abandon	Détruit la session et libère ses ressources.
Evénement Session_OnEnd	Se produit lors de l'abandon ou du dépassement de délai d'une session. Les seuls éléments intrinsèques alors disponibles sont Application, Server et Session. Ce gestionnaire d'événement doit être écrit en VBScript ou en JScript.
Evénement Session_OnStart	Se produit lors de la création par le serveur d'une nouvelle session (après Application_OnStart mais avant l'exécution du script de la page Active Server). Tous les éléments intrinsèques sont alors disponibles. Ce gestionnaire d'événement doit être écrit en VBScript ou en JScript.

Server

On accède à l'objet Server via une interface *IServer*. Il propose divers utilitaires pour écrire votre application ASP.

IServer propose les membres suivants :

Tableau 37.5 Membres de l'interface IServer

Propriété, méthode ou événement	Signification
Propriété ScriptTimeout	Identique à la propriété Timeout de l'objet Session.
Méthode CreateObject	Instancie l'objet Active Server spécifié.
Méthode Execute	Exécute le script d'un fichier .asp spécifié.
Méthode GetLastError	Renvoie un objet ASPError qui décrit la condition d'erreur.
Méthode HTMLEncode	Code une chaîne en vue de son utilisation dans un en-tête HTML, en remplaçant les caractères réservés par les constantes symboliques appropriées.
Méthode MapPath	Associe un répertoire virtuel spécifié (un répertoire absolu dans le serveur en cours ou un chemin relatif à la page en cours) à un chemin d'accès physique.
Méthode Transfer	Envoie toutes les informations d'état en cours à une autre page Active Server pour traitement.
Méthode URLEncode	Applique les règles de codage d'une URL, y compris les caractères d'échappement, à la chaîne spécifiée

Création d'ASP pour des serveurs en et hors processus

Vous pouvez utiliser **Server.CreateObject** dans une page ASP pour démarrer un serveur en ou hors processus en fonction de vos besoins. Généralement les serveurs en processus sont plus couramment utilisés.

A la différence de la plupart des serveurs en processus, un objet Active Server ne s'exécute pas dans l'espace de processus du client mais dans celui de IIS. Cela signifie que le client n'a pas besoin de télécharger votre application (comme c'est, par exemple, le cas avec des objets ActiveX). Les composants DLL en processus sont plus rapides et plus fiables que des serveurs hors processus, il est donc préférable de les utiliser du côté serveur.

Comme les serveurs hors processus sont moins fiables, IIS est fréquemment configuré pour interdire l'utilisation d'exécutables hors processus. Dans ce cas, la création d'un serveur hors processus par votre objet Active Server produit une erreur de la forme suivante :

```
Server object error 'ASP 0196'
Cannot launch out of process component
/path/outofprocess_exe.asp, line 11
```

De plus, comme les composants hors processus créent souvent des processus de serveur individuels pour chaque instance d'objet, ils sont plus lents que les applications CGI. Ils ne supportent pas aussi bien les redéploiements que les composants DLL.

Si les performances et le redéploiement sont des priorités, il est fortement conseillé d'utiliser des composants en processus. Cependant les sites Intranet qui ont un trafic faible ou modéré peuvent utiliser un composant hors processus sans affecter les performances globales du site.

Pour des informations générales sur les serveurs en et hors processus, voir "Serveurs en processus, hors processus et distants" à la page 33-7.

Recensement d'un objet Active Server

Vous pouvez recenser l'objet ASP comme serveur en ou hors processus, mais les serveurs en processus sont plus fréquemment utilisés.

Remarque Si vous voulez retirer l'objet ASP de votre système, vous devez commencer par annuler son recensement, puis retirer ses entrées dans le registre Windows.

Recensement d'un serveur en processus

Pour recenser un serveur en processus (DLL ou OCX) :

- Choisissez Exécuter | Recenser le serveur ActiveX.

Pour annuler le recensement d'un serveur en processus :

- Choisissez Exécuter | Dérecenser le serveur ActiveX.

Recensement d'un serveur hors processus

Pour recenser un serveur hors processus :

- Exécutez le serveur avec l'option de ligne de commande /regserver. Pour spécifier les options de la ligne de commande, vous pouvez utiliser la boîte de dialogue Exécuter|Paramètres.

  Vous pouvez également recenser le serveur en l'exécutant.

Pour annuler le recensement d'un serveur hors processus :

- Exécutez le serveur avec l'option de ligne de commande /unregserver.

Test et débogage d'une application ASP

Le débogage d'un serveur en processus comme un objet Active Server se fait comme celui d'une DLL. Vous choisissez une application hôte qui charge la DLL et débogez de manière habituelle. Pour tester et déboguer un objet Active Server :

1. Activez, si nécessaire, les informations de débogage en utilisant la page Compilateur de la boîte de dialogue Projet|Options. Activez également le débogage intégré dans la boîte de dialogue Outils|Options du débogueur.
2. Choisissez Exécuter|Paramètres, entrez le nom de votre serveur Web dans la zone Application hôte et choisissez OK.
3. Choisissez Exécuter|Exécuter.
4. Définissez des points d'arrêt dans l'implémentation de l'objet Active Server.
5. Utilisez le navigateur Web pour interagir avec l'objet ASP.

Le débogueur s'arrête quand les points d'arrêt sont atteints.

Chapitre

38

Création d'un contrôle ActiveX

Un contrôle ActiveX est un composant logiciel qui peut s'intégrer à toute application hôte acceptant les contrôles ActiveX (comme C++Builder, Delphi, Visual Basic, Internet Explorer et, avec un complément, Netscape Navigator) et en étendre les fonctionnalités. Les contrôles ActiveX implémentent un ensemble particulier d'interfaces qui permettent cette intégration.

Ainsi, Delphi est fourni avec plusieurs contrôles ActiveX : graphes, tableurs ou graphiques. Vous pouvez ajouter ces contrôles à la palette des composants de l'EDI et les utiliser comme tout composant VCL standard en les déposant dans des fichiers et en définissant leurs propriétés avec l'inspecteur d'objets.

Vous pouvez également déployer sur le Web un contrôle ActiveX, en le référençant dans des documents HTML et en le visualisant dans un navigateur Web gérant ActiveX.

Delphi propose des experts qui permettent de créer deux types de contrôles ActiveX :

- **Des contrôles ActiveX qui encapsulent des classes VCL.** En encapsulant une classe VCL, vous pouvez convertir les composants existants en contrôles ActiveX ou en créer de nouveaux, les tester localement puis les convertir en contrôles ActiveX. Généralement, les contrôles ActiveX sont conçus pour s'intégrer dans une application plus importante.
- **Des fiches actives.** Les fiches actives (ActiveForms) vous permettent d'utiliser le concepteur de fiche pour créer un contrôle plus élaboré qui se comporte comme une boîte de dialogue ou même comme une application complète. Vous développez une fiche active de la même manière qu'une application Delphi normale. Les fiches actives sont généralement conçues en vue d'un déploiement sur le Web.

Ce chapitre est une présentation générale de la façon de créer un contrôle ActiveX dans l'environnement Delphi. Vous n'y trouverez pas les détails complets de l'écriture des contrôles ActiveX sans utiliser d'experts. Pour cela,

consultez la documentation Microsoft Developer's Network (MSDN) ou recherchez sur le site Web de Microsoft des informations sur ActiveX.

Présentation de la création d'un contrôle ActiveX

Créer des contrôles ActiveX avec Delphi ressemble beaucoup à la création de contrôles ou de fiches ordinaires. Par contre, c'est très différent de la création des autres objets COM où vous commencez par définir l'interface de l'objet, puis complétez ensuite l'implémentation. Pour créer des contrôles ActiveX (autres que les fiches actives), il faut inverser ce processus, er partir de l'implémentation d'un contrôle VCL puis générer l'interface et la bibliothèque de types une fois que le contrôle est codé. Pour la création de fiches actives, l'interface et la bibliothèque de types sont créés en même temps que la fiche, puis vous utilisez le concepteur de fiche pour implémenter la fiche.

Une fois achevé, un contrôle ActiveX est composé d'un contrôle VCL qui fournit l'implémentation sous-jacente, un objet COM qui encapsule le contrôle VCL et une bibliothèque de types qui énumère les propriétés, méthodes et événements de l'objet COM.

Pour créer un nouveau contrôle ActiveX (autre qu'une fiche active), effectuez les étapes suivantes :

1. Concevez et créez le contrôle VCL personnalisé qui sera la base de votre contrôle ActiveX.
2. Utilisez l'expert contrôle ActiveXpour créer un contrôle ActiveX à partir du contrôle VCL créé à l'étape 1.
3. Utilisez l'expert de page propriétés ActiveXpour créer une ou plusieurs pages de propriétés pour le contrôle (facultatif).
4. Associez une page de propriétés à un contrôle ActiveX (facultatif).
5. Recensez le contrôle ActiveX.
6. Testez le contrôle ActiveX avec toutes les applications cible possibles.
7. Déployez le contrôle ActiveX sur le Web (facultatif).

Pour créer une nouvelle fiche active, effectuez les étapes suivantes :

1. Utilisez l'expert ActiveForm pour créer une fiche active, qui apparaît comme une fiche vide dans l'EDI et l'enveloppe ActiveX associée à cette fiche.
2. Utilisez le concepteur de fiche pour ajouter des composants à votre fiche active et implémenter son comportement comme vous l'utiliseriez pour une fiche ordinaire.
3. Suivez les étapes 3-7 précédentes pour donner à la fiche active une page de propriétés, la recenser et la déployer sur le Web.

Eléments d'un contrôle ActiveX

Un contrôle ActiveX implique plusieurs éléments qui ont chacun une fonction spécifique. Ces éléments sont un contrôle VCL, un objet enveloppe COM correspondant qui expose des propriétés, des méthodes, des événements, et une ou plusieurs bibliothèques de types associées.

Contrôle VCL

Dans Delphi, l'implémentation sous-jacente à un contrôle ActiveX est un contrôle VCL. Quand vous créez un contrôle ActiveX, vous devez commencer par concevoir ou choisir le contrôle VCL à partir duquel vous aller construire votre contrôle ActiveX.

Le contrôle VCL sous-jacent doit être un descendant de *TWinControl* car il doit disposer d'une fenêtre qui peut être accueillie par l'application hôte. Quand vous créez une fiche active, cet objet est un descendant de *TActiveForm*.

Remarque L'expert contrôle ActiveX énumère les descendants *TWinControl* dans lesquels vous pouvez choisir pour créer un contrôle ActiveX. Néanmoins, cette liste ne propose pas tous les descendants *TWinControl*. Certains contrôles sont répertoriés comme incompatibles avec ActiveX (en utilisant la procédure *RegisterNonActiveX*) et n'apparaissent pas dans cette liste.

Enveloppe ActiveX

Le véritable objet COM est un objet ActiveX qui encaspule le contrôle VCL. Pour les fiches actives, cette classe est toujours *TActiveFormControl*. Pour les autres contrôles ActiveX, elle a un nom de la forme *TVCLClassX*, où *TVCLClass* est le nom de la classe du contrôle VCL. Ainsi, par exemple, l'enveloppe ActiveX de *TButton* se nommerait *TButtonX*.

La classe enveloppe est un descendant de *TActiveXControl*, qui gère les interfaces ActiveX. L'enveloppe ActiveX en hérite, ce qui lui permet de retransmettre les messages Windows au contrôle VCL et d'accueillir sa fenêtre dans l'application hôte.

L'enveloppe ActiveX expose les propriétés et méthodes du contrôle VCL aux clients en utilisant son interface par défaut. L'expert implémente automatiquement la plupart des propriétés et méthodes de classe enveloppe et délègue les appels de méthode au contrôle VCL sous-jacent. L'expert fournit également à la classe enveloppe les méthodes qui déclenchent les événements du contrôle VCL dans les clients et affecte ces méthodes aux gestionnaires d'événements du contrôle VCL.

Bibliothèque de types

L'expert contrôle ActiveX génère automatiquement une bibliothèque de types qui contient les définitions de type pour la classe enveloppe, son interface par défaut et toutes les définitions de type nécessaires. Ces informations de type permettent au contrôle d'informer les applications hôtes de ses services. Vous pouvez visualiser et modifier ces informations en utilisant l'éditeur de bibliothèques de

types. Même si ces informations sont stockées dans un fichier bibliothèque binaire séparé (d'extension .TLB), elles sont également compilées automatiquement sous forme de ressources dans la DLL du contrôle ActiveX.

Page de propriétés

Vous pouvez éventuellement attribuer à votre contrôle ActiveX une page de propriétés. La page de propriétés permet à l'utilisateur de l'application hôte (le client) de visualiser et de modifier les propriétés du contrôle. Vous pouvez regrouper plusieurs propriétés dans une page ou utiliser une page pour proposer une interface de type boîte de dialogue à une propriété. Pour davantage d'informations sur la manière de créer des pages de propriétés, voir "Création d'une page de propriétés pour un contrôle ActiveX" à la page 38-13.

Conception d'un contrôle ActiveX

Lors de la conception d'un contrôle ActiveX, vous commencez par créer un contrôle VCL personnalisé. Il sert de base à votre contrôle ActiveX. Pour des informations sur la création de contrôles personnalisés, voir la partie V, "Création de composants personnalisés".

Quand vous concevez le contrôle VCL, n'oubliez pas qu'il va être incorporé dans une autre application : ce contrôle n'est pas une application en lui-même. De ce fait, vous devez éviter de concevoir des boîte de dialogue trop sophistiquées. Généralement, votre but est de concevoir un contrôle simple qui travaille à l'intérieur d'une application principale et en respecte les règles.

De plus, vous devez vous assurer que les types de toutes les propriétés et méthodes que votre contrôle expose aux clients sont compatibles avec l'Automation, car l'interface du contrôle ActiveX doit gérer *IDispatch*. L'expert n'ajoute à l'interface de la classe enveloppe que les méthodes ayant des paramètres dont le type est compatible avec l'Automation. Pour une liste des types compatibles avec l'Automation, voir "Types autorisés" à la page 34-13.

Les experts implémentent toutes les interfaces ActiveX nécessaires en utilisant la classe enveloppe COM. Ils représentent également toutes les propriétés, méthodes et événements compatibles avec l'Automation via l'interface par défaut de la classe enveloppe. Quand l'expert a généré la classe COM enveloppe et son interface, vous pouvez utiliser l'éditeur de bibliothèques de types pour modifier l'interface par défaut ou augmenter la classe enveloppe en implémentant d'autres interfaces.

Génération d'un contrôle ActiveX à partir d'un contrôle VCL

Pour générer un contrôle ActiveX à partir d'un contrôle VCL, utilisez l'expert contrôle ActiveX. Les propriétés, méthodes et événements du contrôle VCL deviennent les propriétés, méthodes et événements du contrôle ActiveX.

Avant d'utiliser l'expert contrôle ActiveX, vous devez choisir le contrôle VCL qui fournit l'implémentation sous-jacente du contrôle ActiveX généré.

Pour afficher l'expert contrôle ActiveX :

1. Choisissez Fichier|Nouveau pour ouvrir la boîte de dialogue Nouveaux éléments.
2. Sélectionnez l'onglet ActiveX.
3. Double-cliquez sur l'icône Contrôle ActiveX.

Dans l'expert, sélectionnez le nom du contrôle VCL encapsulé par le nouveau contrôle ActiveX. La boîte de dialogue énumère tous les contrôles disponibles, c'est-à-dire les descendants *TWinControl* qui ne sont pas recensés comme incompatibles avec ActiveX en utilisant la procédure *RegisterNonActiveX*.

Astuce Si vous ne voyez pas le contrôle souhaité dans la liste déroulante, vérifiez si vous l'avez installé dans l'EDI ou ajoutez son unité à votre projet.

Une fois le contrôle sélectionné, l'expert génère automatiquement un nom pour la coclasse, l'unité d'implémentation de l'enveloppe ActiveX et le projet de bibliothèque ActiveX. Si un projet de bibliothèque ActiveX est déjà ouvert et qu'il ne contient pas d'objet événement COM+, ce projet est automatiquement utilisé. Vous pouvez modifier ces noms dans l'expert sauf si vous avez déjà un projet de bibliothèque ActiveX ouvert, auquel cas le nom du projet n'est pas modifiable.

L'expert utilise toujours le modèle de thread Apartment. Cela ne pose pas de problème si votre projet ActiveX ne contient qu'un seul contrôle. Cependant, si vous ajoutez d'autres objets à votre projet, c'est à vous d'assurer la gestion des threads.

L'expert vous permet également de configurer diverses options du contrôle ActiveX :

- **Licence de conception :** Vous pouvez faire recenser votre contrôle pour vous assurer que les utilisateurs du contrôle ne peuvent pas l'ouvrir en conception ou à l'exécution, sauf s'ils disposent d'une clé de licence pour le contrôle.
- **Informations de version :** Vous pouvez inclure dans le contrôle ActiveX des informations de version, comme le copyright ou une description du fichier. Ces informations peuvent être visualisées par un navigateur. Certains clients hôtes comme Visual Basic 4.0 exigent des informations de version pour accueillir le contrôle ActiveX. Spécifiez les informations de version en choisissant Projet|Options et en sélectionnant la page Informations de version.
- **Boîte A propos :** Vous pouvez demander à l'expert de générer une fiche séparée qui implémente une boîte de dialogue A propos pour le contrôle. Les utilisateurs de l'application hôte peuvent afficher cette boîte de dialogue A propos dans un environnement de développement. Par défaut, la boîte de dialogue A propos contient le nom du contrôle ActiveX, une image, les informations de copyright et un bouton OK. Vous pouvez modifier cette fiche que l'expert ajoute à votre projet.

Quand vous sortez de l'expert, il génère les éléments suivants :

- Un fichier projet de bibliothèque ActiveX qui contient le code nécessaire pour démarrer un contrôle ActiveX. Généralement, vous ne modifierez pas ce fichier.
- Une bibliothèque de types qui définit une CoClasse pour le contrôle, l'interface qu'elle expose aux clients et les définitions de types qui leurs sont nécessaires. Pour davantage d'informations sur la bibliothèque de types, voir chapitre 34, "Utilisation des bibliothèques de types".
- Une unité d'implémentation ActiveX qui définit et implémente le contrôle ActiveX, un descendant de *TActiveXControl*. Ce contrôle ActiveX est une implémentation entièrement fonctionnelle qui ne nécessite aucun travail de votre part. Vous pouvez cependant modifier cette classe si vous voulez personnaliser les propriétés, méthodes et événements que le contrôle ActiveX expose aux clients.
- Une unité et une fiche de boîte de dialogue A propos, si vous l'avez demandé.
- Un fichier .LIC, si vous avez demandé une licence.

Génération d'un contrôle ActiveX basé sur une fiche VCL

A la différence des autres contrôles ActiveX, les fiches actives ne sont pas d'abord conçues puis encapsulées dans une classe enveloppe ActiveX. A la place, l'expert ActiveForm génère une fiche vide que vous concevez après coup quand l'expert vous laisse dans le concepteur de fiche.

Lorsqu'une fiche ActiveForm est déployée sur le Web, Delphi crée une page HTML pour contenir la référence à la fiche ActiveForm et spécifier son emplacement dans la page. La fiche ActiveForm peut ensuite être affichée et exécutée depuis un navigateur Web. Dans le navigateur Web, la fiche se comporte comme une fiche autonome Delphi. Elle peut contenir n'importe quel composant VCL ou ActiveX, y compris des contrôles VCL personnalisés.

Pour démarrer l'expert ActiveForm :

1. Choisissez Fichier|Nouveau pour ouvrir la boîte de dialogue Nouveaux éléments.
2. Sélectionnez l'onglet ActiveX.
3. Double-cliquez sur l'icône ActiveForm.

L'expert ActiveForm ressemble à l'expert contrôle ActiveX mais vous ne spécifiez pas le nom de la classe VCL à encapsuler. En effet, les fiches actives sont toujours basées sur *TActiveForm*.

Comme dans l'expert contrôle ActiveX, vous pouvez changer les noms par défaut de la CoClasse, de l'unité d'implémentation et du projet de bibliothèque ActiveX. De même l'expert vous permet d'indiquer si vous voulez que votre

fiche active nécessite une licence, si elle doit contenir des informations de version et si vous voulez une fiche A propos.

Quand vous sortez de l'expert, il génère :

- Un fichier projet de bibliothèque ActiveX qui contient le code nécessaire pour démarrer un contrôle ActiveX. Généralement, vous ne modifierez pas ce fichier.
- Une bibliothèque de types qui définit une CoClasse pour le contrôle, l'interface qu'elle expose aux clients et les définitions de types qui leurs sont nécessaires. Pour davantage d'informations sur la bibliothèque de types, voir chapitre 34, "Utilisation des bibliothèques de types".
- Une fiche qui dérive de *TActiveForm*. Cette fiche apparaît dans le concepteur de fiche, ce qui vous permet de concevoir visuellement comment la fiche active apparaîtra aux clients. Son implémentation apparaît dans l'unité d'implémentation générée. Dans la section initialisation de l'unité d'implémentation, un fabricant de classe est créé, et il configure *TActiveFormControl* comme l'enveloppe ActiveX de cette fiche.
- Une unité et une fiche de boîte de dialogue A propos, si vous l'avez demandé.
- Un fichier .LIC, si vous avez demandé une licence.

Arrivé là, vous pouvez ajouter des contrôles à la fiche et la concevoir à votre guise.

Une fois le projet ActiveForm conçu et compilé dans une bibliothèque ActiveX (d'extension OCX), vous pouvez déployer le projet sur votre serveur Web et Delphi crée une page HTML test contenant une référence à la fiche active.

Licences des contrôles ActiveX

Attribuer une licence à un contrôle ActiveX consiste à fournir une clé de licence à la conception et à gérer la création dynamique de licences pour les contrôles créés à l'exécution.

Pour fournir des licences de conception, l'expert ActiveX crée une clé pour le contrôle, stockée dans un fichier d'extension LIC portant le même nom que le projet. Ce fichier .LIC est ajouté au projet. L'utilisateur du contrôle doit disposer d'une copie du fichier .LIC pour pouvoir utiliser le contrôle dans son environnement de développement. Chaque contrôle d'un projet pour lequel la case Licence de conception est cochée dispose d'une entrée distincte dans le fichier LIC.

Pour gérer les licences à l'exécution, la classe enveloppe implémente deux méthodes, *GetLicenseString* et *GetLicenseFilename*. Elles renvoient, respectivement, la chaîne de licence du contrôle et le nom du fichier .LIC. Quand une application hôte tente de créer un contrôle ActiveX, le fabricant de classe du contrôle appelle ces méthodes et compare la chaîne renvoyée par *GetLicenseString* à la chaîne stockée dans le fichier .LIC.

Les licences d'exécution pour Internet Explorer nécessitent un niveau supplémentaire d'indirection, car l'utilisateur peut visualiser le code source HTML de toutes les pages Web et un contrôle ActiveX est copié sur son ordinateur avant qu'il ne soit affiché. Pour créer des licences d'exécution pour les contrôles utilisés sous Internet Explorer, vous devez d'abord générer un fichier paquet de licence (fichier LPK) et l'incorporer dans la page HTML qui contient le contrôle. Le fichier LPK est essentiellement un tableau de CLSID et de clés de licence.

Remarque Pour générer le fichier LPK, utilisez l'utilitaire LPK_TOOL.EXE, que vous pouvez télécharger à partir du site Web de Microsoft (www.microsoft.com).

Pour incorporer le fichier LPK dans une page Web, utilisez les objets HTML, <OBJECT> et <PARAM>, comme suit :

```
<OBJECT CLASSID="clsid:6980CB99-f75D-84cf-B254-55CA55A69452">
    <PARAM NAME="LPKPath" VALUE="ctrllic.lpk">
</OBJECT>
```

Le CLSID identifie l'objet en tant que paquet de licence et PARAM spécifie l'emplacement relatif du fichier paquet de licence par rapport à la page HTML.

Lorsque Internet Explorer essaie d'afficher la page Web contenant le contrôle, il analyse le fichier LPK, extrait la clé de licence et, si celle-ci correspond à la licence du contrôle (renvoyée par *GetLicenseString*), il restitue le contrôle dans la page. Si plusieurs fichiers LPK sont inclus dans une page Web, Internet Explorer ne prend en compte que le premier.

Pour plus d'informations, effectuez des recherches sur la définition de licences de contrôles ActiveX sur le site Web de Microsoft.

Personnalisation de l'interface du contrôle ActiveX

Les experts contrôle ActiveX et ActiveForm génèrent une interface par défaut pour la classe enveloppe du contrôle ActiveX. Cette interface par défaut expose simplement les propriétés, méthodes et événements du contrôle VCL d'origine ou de la fiche, avec les exceptions suivantes :

- Les propriétés orientées données n'apparaissent pas. Comme les contrôles ActiveX ont une autre manière d'associer des contrôles aux données que les contrôles VCL, l'expert ne convertit pas les propriétés relatives aux données. Pour des informations sur la manière d'obtenir des contrôles ActiveX orientés données, voir "Activation de la liaison de données simple avec la bibliothèque de types" à la page 38-12.
- Les propriétés, méthodes ou événements dont le type n'est pas compatible avec l'Automation n'apparaissent pas. Vous pouvez les ajouter à l'interface du contrôle ActiveX une fois que l'expert a terminé.

Vous pouvez ajouter, modifier ou supprimer les propriétés, méthodes et événements d'un contrôle ActiveX en modifiant la bibliothèque de types. Vous pouvez utiliser l'éditeur de bibliothèques de types comme décrit dans le chapitre 34, "Utilisation des bibliothèques de types".N'oubliez pas que si vous

ajoutez des événements, ils doivent être ajoutés à l'interface Events et non à l'interface par défaut du contrôle ActiveX.

Remarque Vous pouvez ajouter des propriétés non publiées à l'interface de votre contrôle ActiveX. De telles propriétés peuvent être définies à l'exécution et apparaissent dans l'environnement de développement, mais les modifications effectuées ne sont pas persistantes. Donc, si l'utilisateur du contrôle modifie la valeur de la propriété à la conception, les modifications ne sont pas prises en compte dans le contrôle à l'exécution. Si la source est un objet VCL alors que la propriété n'est pas déjà publiée, vous pouvez rendre la propriété persistente en créant un descendant de l'objet VCL et en publiant la propriété dans le descendant.

Vous n'êtes pas obligé d'exposer toutes les propriétés, méthodes et événements du contrôle VCL aux applications hôtes. Vous pouvez utiliser l'éditeur de bibliothèques de types pour les retirer des interfaces que l'expert a généré. Quand vous retirez des propriétés et méthodes d'une interface en utilisant l'éditeur de bibliothèques de types, l'éditeur ne les retire pas de la classe d'implémentation correspondante. Editez la classe enveloppe ActiveX dans l'unité d'implémentation pour les en retirer une fois que vous avez modifié l'interface dans l'éditeur de bibliothèques de types.

Attention Toutes les modifications effectuées dans la bibliothèque de types sont perdues si vous régénérez le contrôle ActiveX depuis le contrôle ou la fiche VCL d'origine.

Astuce Il est judicieux de tester les méthodes que l'expert ajoute à votre classe enveloppe ActiveX. Cela vous donne l'opportunité de vérifier où l'expert a omis des propriétés orientées données ou des méthodes incompatibles avec l'Automation ; mais cela vous donne également l'occasion de détecter les méthodes pour lesquelles l'expert n'a pas pu générer une implémentation. Ces méthodes apparaissent dans l'implémentation avec un commentaire indiquant le problème.

Ajout de propriétés, méthodes et événements supplémentaires

Vous pouvez ajouter au contrôle des propriétés, méthodes et événements supplémentaires par le biais de l'éditeur de bibliothèques de types. La déclaration est automatiquement ajoutée à l'unité d'implémentation, au fichier de bibliothèque de types (TLB) et à l'unité de bibliothèque de types du contrôle. Ce que Delphi ajoute réellement dépend de ce que vous avez ajouté : propriété ou méthode ou si vous avez ajouté un événement.

Ajout de propriétés et de méthodes

La classe enveloppe ActiveX implémente les propriétés dans son interface en utilisant des méthodes d'accès en lecture et en écriture. C'est-à-dire que la classe enveloppe a des propriétés COM qui apparaissent dans une interface comme des méthodes get et/ou set. A la différence des propriétés VCL, vous ne voyez pas une déclaration de propriété dans l'interface des propriétés COM. Vous ne voyez que les méthodes qui sont indiquées comme méthodes d'accès à la propriété. Quand vous ajoutez une propriété à l'interface par défaut du contrôle ActiveX, la définition de la classe enveloppe (qui apparaît dans le fichier unité _TLB

actualisé par l'éditeur de bibliothèques de types) est augmentée d'une ou deux nouvelles méthodes (une méthode get et/ou une méthode set). Vous devez implémenter comme si vous aviez ajouté une méthode à l'interface, la classe enveloppe est augmentée d'une méthode correspondante que vous devez implémenter. L'ajout de propriétés à l'interface de la classe enveloppe revient donc à ajouter des méthodes : le squelette de nouvelles implémentations de méthodes à compléter est ajouté à la définition de la classe enveloppe.

Remarque Pour davantage d'informations sur ce qui apparaît dans le fichier unité _TLB généré, voir "Code généré par l'importation des informations d'une bibliothèque de types" à la page 35-5.

Par exemple, soit une propriété *Caption* de type TCaption dans l'objet VCL sous-jacent. Pour ajouter cette propriété à l'interface de l'objet, entrez ce qui suit quand vous ajoutez une propriété à l'interface via l'éditeur de bibliothèques de types :

```
property Caption: TCaption read Get_Caption write Set_Caption;
```

Delphi ajoute les déclarations suivantes à la classe enveloppe :

```
function Get_Caption: WideString; safecall;
procedure Set_Caption(const Value: WideString); safecall;
```

De plus, il ajoute les squelettes de méthodes d'implémentation suivants que vous devez compléter :

```
function TButtonX.Get_Caption: WideString;
begin
end;

procedure TButtonX.Set_Caption(Value: WideString);
begin
end;
```

Généralement, vous pouvez implémenter ces méthodes par simple délégation au contrôle VCL associé auquel vous avez accès en utilisant le membre *FDelphiControl* de la classe enveloppe :

```
function TButtonX.Get_Caption: WideString;
begin
  Result := WideString(FDelphiControl.Caption);
end;

procedure TButtonX.Set_Caption(const Value: WideString);
begin
  FDelphiControl.Caption := TCaption(Value);
end;
```

Dans certains cas, vous avez besoin d'ajouter du code pour convertir les types de données COM en types natifs Pascal Objet. L'exemple précédent le fait en utilisant le transtypage.

Remarque Comme les méthodes d'une interface Automation sont déclarées **safecall**, vous n'avez pas besoin d'implémenter le code d'exception COM pour ces méthodes. Le compilateur Delphi gère cela pour vous en générant autour du corps des méthodes **safecall** le code permettant de capturer les exceptions Delphi et de les convertir en structures d'information et en codes de retour des erreurs COM.

Ajout d'événement

Le contrôle ActiveX peut déclencher des événements dans son conteneur de la même manière qu'un objet automation déclenche des événements dans ses clients. Ce mécanisme est décrit dans "Exposition d'événements aux clients" à la page 36-11.

Si le contrôle VCL que vous utilisez comme base pour votre contrôle ActiveX a des événements publiés, l'expert ajoute automatiquement le traitement de la liste d'événements client à la classe enveloppe ActiveX et définit la dispinterface de sortie que les clients doivent implémenter pour répondre aux événements.

Pour ajouter un événement dans l'éditeur de bibliothèques de types, sélectionnez l'interface d'événement et cliquez sur l'icône de méthode. Ajoutez ensuite manuellement la liste de paramètres que vous voulez inclure par le biais de la page des paramètres.

Déclarez ensuite une méthode dans la classe d'implémentation qui soit de même type que le gestionnaire de l'événement dans le contrôle VCL sous-jacent. Cela n'est pas généré automatiquement car Delphi ne sait pas quel gestionnaire d'événement vous utilisez :

```
procedure KeyPressEvent(Sender: TObject; var Key: Char);
```

Implémentez cette méthode pour utiliser le récepteur d'événements de l'application hôte stocké dans le membre FEvents de la classe enveloppe :

```
procedure TButtonX.KeyPressEvent(Sender: TObject; var Key: Char);
var
  TempKey: Smallint;
begin
    TempKey := Smallint(Key); {transtypage dans un type compatible OleAutomation }
    if FEvents <> nil then
      FEvents.OnKeyPress(TempKey)
    Key := Char(TempKey);
  end;
```

Remarque Si vous déclenchez des événements dans un contrôle ActiveX, vous n'avez pas besoin de parcourir une liste de récepteurs d'événements car le contrôle n'a qu'une seule application hôte. Le processus est plus simple qu'avec la plupart des serveurs Automation.

Enfin, vous devez attribuer ce gestionnaire d'événement au contrôle VCL sous-jacent afin qu'il soit appelé quand l'événement se produit. Vous effectuez cette affectation dans la méthode *InitializeControl* :

```
procedure TButtonX.InitializeControl;
begin
  FDelphiControl := Control as TButton;
  FDelphiControl.OnClick := ClickEvent;
  FDelphiControl.OnKeyPress := KeyPressEvent;
end;
```

Activation de la liaison de données simple avec la bibliothèque de types

Avec la liaison de données simple, vous pouvez lier une propriété de votre contrôle ActiveX à un champ d'une base de données. Pour ce faire, le contrôle ActiveX doit communiquer à son application hôte la valeur que représente le champ de données et quand elle est modifiée. Vous activez cette communication en définissant les indicateurs de liaison de la propriété avec l'éditeur de bibliothèques de types.

Si vous spécifiez qu'une propriété est liée, quand un utilisateur modifie la propriété (par exemple, un champ d'une base de données), le contrôle notifie son conteneur (l'application client hôte) que la valeur a été modifiée et demande que l'enregistrement de la base de données soit actualisé. Le conteneur interagit avec la base de données, puis notifie le contrôle de la réussite ou de l'échec de l'actualisation de l'enregistrement.

Remarque C'est l'application conteneur accueillant votre contrôle ActiveX qui est responsable de la connexion avec la base de données des propriétés spécifiées comme orientés données dans la bibliothèque de types. Pour des informations sur l'écriture d'un tel conteneur avec Delphi, voir "Utilisation de contrôles ActiveX orientés données" à la page 35-9.

Utilisez la bibliothèque de types pour activer la liaison de données simple,

1. Dans la barre d'outils, cliquez sur la propriété à lier.
2. Choisissez la page Indicateurs.
3. Sélectionnez les attributs de liaison suivants :

Attributs de liaison	Description
Bindable	Indique que la propriété supporte la liaison de données. Si elle est marquée "bindable", la propriété notifie à son conteneur quand sa valeur a été modifiée.
Request Edit	Indique que la propriété supporte la notification OnRequestEdit. Cela permet au contrôle de demander au conteneur si sa valeur peut être modifiée par l'utilisateur.
Display Bindable	Indique que le conteneur peut montrer aux utilisateurs que cette propriété est "bindable".
Default Bindable	Indique la propriété "bindable" qui représente le mieux l'objet. Les propriétés qui ont cet attribut de liaison par défaut doivent avoir aussi l'attribut bindable. On ne peut pas spécifier plusieurs propriétés de ce type dans une dispinterface.
Immediate Bindable	Chacune des propriétés "bindable" d'une fiche peut bénéficier de ce comportement. Quand ce bit est défini, toutes les modifications sont notifiées. Les bits Bindable et Request Edit doivent être définis pour que cet attribut prenne effet.

4. Cliquez sur le bouton Rafraîchir de la barre d'outils pour actualiser la bibliothèque de types.

Pour pouvoir tester un contrôle dont la liaison de données est activée, vous devez d'abord le recenser.

Par exemple, pour lier un contrôle *TEdit* aux données d'un contrôle ActiveX, créez le contrôle ActiveX à partir d'un *TEdit*, puis modifiez les indicateurs de la propriété Text en Bindable, Display Bindable, Default Bindable et Immediate Bindable.

Une fois le contrôle recensé et importé, il peut être utilisé pour afficher les données.

Création d'une page de propriétés pour un contrôle ActiveX

Une page de propriétés est une boîte de dialogue similaire à l'inspecteur d'objets Delphi qui permet aux utilisateurs de modifier les propriétés d'un contrôle ActiveX. Un dialogue de page de propriétés sert à regrouper plusieurs propriétés d'un contrôle afin de les modifier ensemble ou à gérer des propriétés complexes.

En général, les utilisateurs accèdent à la page des propriétés en cliquant sur le contrôle ActiveX avec le bouton droit de la souris et en choisissant Propriétés.

Le processus de création d'une page de propriétés est similaire à celui d'une fiche :

1. Création d'une nouvelle page de propriétés.
2. Ajout de contrôles à la page de propriétés.
3. Association des contrôles de la page de propriétés aux propriétés d'un contrôle ActiveX.
4. Connexion de la page de propriétés au contrôle ActiveX.

Remarque Quand des propriétés sont ajoutées à un contrôle ActiveX ou à une fiche ActiveForm, vous devez publier les propriétés dont vous voulez qu'elles persistent. Si ces propriétés ne sont pas publiées dans le contrôle VCL sous-jacent, vous devez créer un descendant personnalisé du contrôle VCL qui redéclare les propriétés comme publiées, et utiliser l'expert contrôle ActiveX pour créer le contrôle ActiveX à partir de cette classe dérivée.

Création d'une nouvelle page de propriétés

Pour créer une nouvelle page de propriétés, utilisez l'expert Page propriétés.

Pour créer une nouvelle page de propriétés,

1. Choisissez Fichier | Nouveau.
2. Sélectionnez l'onglet ActiveX.
3. Double-cliquez sur l'icône Page propriétés.

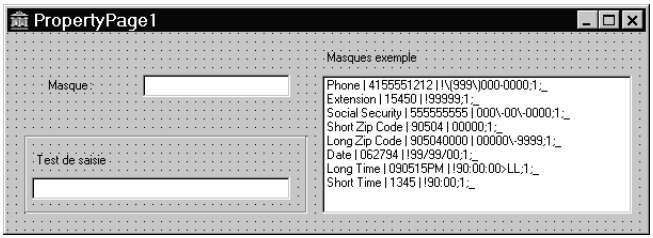
L'expert crée une nouvelle fiche et l'unité d'implémentation de la page de propriétés. La fiche est un descendant de *TPropertyPage* qui vous permet d'associer la fiche avec le contrôle ActiveX dont il modifie les propriétés.

Ajout de contrôles à une page de propriétés

Vous devez ajouter à la page de propriétés un contrôle pour chaque propriété du contrôle ActiveX auquel l'utilisateur doit avoir accès.

Par exemple, la figure suivante illustre la configuration de la page de propriétés pour un contrôle ActiveX MaskEdit.

Figure 38.1 Page de propriétés MaskEdit en mode conception



La boîte liste permet à l'utilisateur de sélectionner parmi une liste de modèles exemple. Les contrôles de saisie permettent à l'utilisateur de tester le masque avant de l'appliquer au contrôle ActiveX.

Association des contrôles de la page de propriétés aux propriétés du contrôle ActiveX

Une fois tous les contrôles nécessaires ajoutés à la page de propriétés, il faut associer chaque contrôle à la propriété correspondante. Cette association s'effectue en ajoutant du code à l'unité implémentant la page de propriétés. Précisément, il faut ajouter du code aux méthodes *UpdatePropertyPage* et *UpdateObject*.

Actualisation de la page de propriétés

Ajoutez du code à la méthode *UpdatePropertyPage* pour mettre à jour le contrôle de la page de propriétés quand les propriétés du contrôle ActiveX changent. Vous devez ajouter du code à la méthode *UpdatePropertyPage* pour actualiser la page de propriétés avec les valeurs en cours des propriétés du contrôle ActiveX.

Vous pouvez accéder au contrôle ActiveX en utilisant la propriété *OleObject* de la page de propriétés qui est un *OleVariant* contenant l'interface du contrôle ActiveX.

Par exemple, le code suivant actualise le contrôle boîte de saisie (InputMask) de la page de propriétés avec la valeur en cours de la propriété *EditMask* du contrôle ActiveX:

```
procedure TPropertyPage1.UpdatePropertyPage;
begin
   { Update your controls from OleObject }
   InputMask.Text := OleObject.EditMask;
end;
```

Remarque Il est également possible d'écrire une page de propriétés qui représente plusieurs contrôles ActiveX. Dans ce cas, n'utilisez pas la propriété *OleObject*. Vous devez, à la place, parcourir la liste des interfaces qui est gérée par la propriété *OleObjects*.

Actualisation de l'objet

Ajoutez du code à la méthode *UpdateObject* pour mettre à jour la propriété quand l'utilisateur modifie les contrôles de la page de propriétés. Vous devez ajouter du code à la méthode *UpdateObject* afin de définir la nouvelle valeur des propriétés du contrôle ActiveX.

Utilisez la propriété *OleObject* pour accéder au contrôle ActiveX.

Par exemple, le code suivant affecte la propriété *EditMask* d'un contrôle ActiveX en utilisant la valeur du contrôle boîte de saisie (InputMask) de la page de propriétés :

```
procedure TPropertyPage1.UpdateObject;
begin
   {Mettre à jour OleObject à partir de votre contrôle }
   OleObject.EditMask := InputMask.Text;
end;
```

Connexion d'une page de propriétés à un contrôle ActiveX

Pour connecter une page de propriétés à un contrôle ActiveX,

1 Ajoutez *DefinePropertyPage*, avec la constante GUID de la page de propriétés comme paramètre, à l'implémentation de la méthode *DefinePropertyPages* dans l'implémentation des contrôles de l'unité. Par exemple,

```
procedure TButtonX.DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);
begin
   DefinePropertyPage(Class_PropertyPage1);
end;
```

   La constante GUID, Class_PropertyPage1, de la page de propriétés peut être trouvée dans l'unité des pages de propriétés.

   Le GUID est défini dans l'unité d'implémentation de la page de propriétés ; il est généré automatiquement par l'expert Page propriétés.

2 Ajoutez l'unité de la page de propriétés à la clause **uses** de l'unité d'implémentation des contrôles.

Recensement d'un contrôle ActiveX

Une fois le contrôle ActiveX créé, il faut le recenser afin que d'autres applications puissent le trouver et l'utiliser.

Pour recenser un contrôle ActiveX :

- Choisissez Exécuter | Recenser le serveur ActiveX.

Remarque Avant de retirer un contrôle ActiveX de votre système, vous devez annuler son recensement.

Pour annuler le recensement d'un contrôle ActiveX :

- Choisissez Exécuter | Dérecenser le serveur ActiveX.

Comme alternative, vous pouvez utiliser **tregsvr** sur la ligne de commande ou exécuter le regsvr32.exe du système d'exploitation.

Test d'un contrôle ActiveX

Pour tester votre contrôle, ajoutez-le à un paquet et importez-le comme contrôle ActiveX. Cette procédure ajoute le contrôle ActiveX à la palette des composants de Delphi. Vous pouvez alors déposer le contrôle dans une fiche et le tester.

Votre contrôle doit également être testé avec toutes les applications cible utilisant ce contrôle.

Pour déboguer le contrôle ActiveX, sélectionnez Exécuter | Paramètres et entrez le nom de l'application client dans la boîte de saisie Application hôte.

Les paramètres s'appliquent ensuite à l'application hôte. La sélection de Exécuter | Exécuter démarre l'application hôte ou client et permet de définir des points d'arrêts dans le contrôle.

Déploiement d'un contrôle ActiveX sur le Web

Avant de pouvoir utiliser dans des clients Web les contrôles ActiveX que vous avez créés, ils doivent être déployés sur le serveur Web. A chaque fois que vous modifiez le contrôle ActiveX, vous devez le recompiler et le redéployer afin que les applications client voient les modifications.

Avant de déployer un contrôle ActiveX, vous devez disposer d'un serveur Web répondant aux messages du client.

Pour déployer un contrôle ActiveX :

1. Sélectionnez Projet | Options de déploiement Web.
2. Dans la page Projet, définissez dans la zone Répertoire destination l'emplacement de la DLL du contrôle ActiveX (sous la forme d'un chemin sur le serveur Web). Cela peut être un chemin d'accès local ou un chemin d'accès UNC, par exemple, C:\INETPUB\wwwroot.
3. Définissez dans URL destination, l'emplacement sur le serveur Web (sous la forme d'une URL) de la DLL du contrôle ActiveX (sans le nom de fichier), par exemple, http://mymachine.inprise.com/. Voir la documentation de votre serveur Web pour davantage d'informations sur la manière de procéder.
4. Définissez dans Répertoire HTML l'emplacement (sous la forme d'un chemin d'accès) où doit être placé le fichier HTML contenant une référence au

contrôle ActiveX, par exemple, C:\INETPUB\wwwroot. Ce chemin d'accès peut être un chemin d'accès standard ou un chemin d'accès UNC.

5 Définissez les options de déploiement Web souhaitées, comme décrit dans "Paramétrage des options" à la page 38-17.

6 Choisissez OK.

7 Choisissez Projet | Déploiement Web.

Cela crée la base du code de déploiement contenant le contrôle ActiveX dans une bibliothèque ActiveX (ayant l'extension OCX). Selon les options que vous avez spécifiées, cette base du code de déploiement peut aussi contenir un fichier cabinet (ayant l'extension CAB) ou d'information (ayant l'extension INF).

La bibliothèque ActiveX est placée dans le répertoire destination spécifié à l'étape 2. Le fichier HTML porte le même nom que le projet mais avec l'extension .HTM. Il est créé dans le répertoire HTML spécifié à l'étape 4. Le fichier HTML contient une URL faisant référence à la bibliothèque ActiveX à l'emplacement spécifié à l'étape 3.

Remarque Si vous voulez placer ces fichiers sur votre serveur Web, utilisez un utilitaire externe comme ftp.

8 Appelez votre navigateur Web gérant ActiveX et visualisez la page HTML ainsi créée.

Quand cette page HTML est visualisée dans le navigateur Web, la fiche ou le contrôle est affiché et exécuté comme application incorporée dans le navigateur. C'est-à-dire que la bibliothèque s'exécute dans le même processus que l'application navigateur.

Paramétrage des options

Avant de déployer un contrôle ActiveX, vous devez spécifier les options de déploiement Web qui doivent être employées pour créer la bibliothèque ActiveX.

Les options de déploiement Web comprennent des paramètres comme :

- **Déployer les fichiers supplémentaires :** Si votre contrôle ActiveX dépend d'autres paquets ou fichiers, vous pouvez indiquer si ceux-ci doivent être déployés avec le projet. Par défaut, ces fichiers utilisent les mêmes options que celles définies pour le projet même, mais vous pouvez redéfinir ce paramétrage en utilisant les onglets Paquets et Fichiers supplémentaires. Quand vous incluez les paquets et fichiers supplémentaires, Delphi crée un fichier d'extension .INF (pour INFormation). Ce fichier spécifie les divers fichiers nécessaires à télécharger et configurer pour pouvoir exécuter la bibliothèque ActiveX. La syntaxe des fichiers INF accepte l'utilisation d'URL pointant sur les paquets ou fichiers supplémentaires à télécharger.
- **Utiliser la compression de fichier CAB :** Un cabinet est un seul fichier, portant généralement l'extension CAB, qui stocke les fichiers compressés dans une bibliothèque de fichiers. La compression de ces fichiers réduit, en général,

considérablement les temps de téléchargement (jusqu'à 70%). Lors de l'installation, le navigateur décompresse les fichiers stockés dans une archive et les copie sur le système de l'utilisateur. Les fichiers que vous déployez peuvent être des archives .CAB compressées. Chaque fichier déployé peut être un fichier compressé CAB. Vous pouvez spécifier que la bibliothèque ActiveX utilise la compression de fichier CAB dans l'onglet Projet de la boîte de dialogue Options de déploiement Web.

- **Informations de version :** Vous pouvez demander à inclure les informations de version dans le contrôle ActiveX. Ces informations sont définies dans la page Informations de version de la boîte de dialogue Options du projet. Parmi ces informations, il y a le numéro de version qui peut être actualisé automatiquement à chaque fois que vous déployez votre contrôle ActiveX. Si vous incluez d'autres paquets ou fichiers, leurs ressources d'informations de version peuvent également être ajoutées au fichier INF.

Si vous choisissez d'ajouter des fichiers supplémentaires ou si vous utilisez la compression de fichier CAB, la bibliothèque ActiveX résultant peut prendre la forme d'un fichier OCX, d'un fichier CAB contenant un fichier OCX ou d'un fichier INF. Le tableau suivant résume le résultat obtenu avec les différentes combinaisons.

Paquets et/ou fichiers supplémentaires	Compression de fichier CAB	Résultat
Non	Non	Un fichier bibliothèque ActiveX (OCX).
Non	Oui	Un fichier CAB contenant un fichier bibliothèque ActiveX.
Oui	Non	Un fichier INF, un fichier bibliothèque ActiveX et tous les paquets et/ou fichiers supplémentaires.
Oui	Oui	Un fichier INF, un fichier CAB contenant un fichier bibliothèque ActiveX et un fichier CAB pour chacun des fichiers et paquets supplémentaires.

Chapitre

39

Création d'objets MTS ou COM+

Delphi utilise le terme objet transactionnel pour désigner des objets qui utilisent les services de transaction, la sécurité et la gestion des ressources proposés par Microsoft Transaction Server (MTS) (pour les versions de Windows antérieures à Windows 2000) ou par COM+ (pour Windows 2000 ou plus). Ces objets sont conçus pour fonctionner dans un environnement distribué important. Ils ne sont pas disponibles dans les applications multiplate-formes à cause de leur dépendance par rapport à la technologie spécifique à Windows.

Delphi propose un expert qui crée des objets transactionnels afin que vous puissiez tirer profit des avantages des attributs COM+ ou de l'environnement MTS. Ces possibilités rendent particulièrement simple la création et l'implémentation de clients et de serveurs COM, en particulier les serveurs distants.

Remarque Pour les applications de bases de données, Delphi propose également un module de données transactionnel. Pour davantage d'informations, voir chapitre 25, "Création d'applications multiniveaux".

Les objets transactionnels procurent un certain nombre de services de base, tels que :

- Gestion des ressources système, y compris processus, threads et connexions aux bases de données, afin que votre application serveur puisse gérer de nombreux utilisateurs simultanés.
- Initialisation et contrôle des transactions automatiques afin que votre application soit fiable.
- Création, exécution et suppression des composants serveur si nécessaire.
- Utilisation de la sécurité en fonction des rôles afin que seuls les utilisateurs autorisés puissent accéder à votre application.
- Gestion des événements afin que les clients puissent répondre à des conditions se produisant sur le serveur (COM+ uniquement).

En fournissant ces services fondamentaux, MTS ou COM+ vous laisse vous concentrer sur ce qui est spécifique à votre propre application distribuée. La technologie utilisée (MTS ou COM+) dépend du serveur sur lequel vous exécutez votre application. Pour les clients, la différence entre les deux (ou même le fait que l'objet serveur utilise l'un de ces services) est transparente, sauf si le client manipule explicitement des services transactionnels via une interface spéciale.

Principe des objets transactionnels

Habituellement, les objets transactionnels sont de petite taille et servent à des fonctions de gestion particulières. Ils peuvent implémenter les règles de gestion d'une application, en fournissant les vues et les transformations de l'état de l'application. Considérons l'exemple d'une application médicale destinée à un médecin. Les enregistrements médicaux stockés dans plusieurs bases de données donnent l'état permanent de l'application, par exemple l'historique de la santé des patients. Les objets transactionnels mettent à jour cet état afin qu'il reflète les modifications qui lui sont apportées, par exemple les nouveaux patients, les résultats d'analyses sanguines ou les fichiers des radiographies.

Les objets transactionnels diffèrent des autres objets COM en ce qu'ils utilisent un ensemble d'attributs fournis par MTS ou COM+ pour gérer les situations se produisant dans un environnement de calcul distribué. Certains de ces attributs imposent que l'objet transactionnel implémente l'interface *IObjectControl*. *IObjectControl* définit des méthodes qui sont appelées quand l'objet est activé ou désactivé et qui vous permettent de gérer des ressources telles que les connexions aux bases de données. Elle est également nécessaire pour le regroupement d'objets, décrit dans "Regroupement d'objets" à la page 39-9.

Remarque Si vous utilisez MTS, vos objets transactionnels doivent implémenter *IObjectControl*. Avec COM+, *IObjectControl* n'est pas obligatoire, mais reste fortement conseillée. L'expert Objet transactionel fournit un objet qui dérive de *IObjectControl*.

Le client d'un objet transactionnel est appelé un **client de base**. Pour un client de base, un objet transactionnel n'est pas différent d'un autre objet COM.

Avec MTS, l'objet transactionnel doit être généré dans une bibliothèque (DLL) installée dans l'environnement d'exécution MTS (l'exécutable MTS, mtxex.exe). Cela signifie que l'objet serveur est exécuté dans l'espace de processus d'exécution MTS. L'exécutable MTS peut être exécuté dans le même processus que le client de base, dans un processus distinct de la même machine ou comme processus serveur distant sur une machine distincte.

Avec COM+, l'application serveur n'a pas besoin d'être un serveur en processus. Comme les différents services ne sont pas intégrés dans les bibliothèques COM, il n'y a pas besoin d'un processus MTS séparé pour intercepter les appels adressés au serveur. C'est COM (ou plus exactement COM+) qui se charge de la gestion des ressources, des transactions, etc. Néanmoins, l'application serveur doit toujours être installée, mais cette fois dans une application COM+.

La connexion entre le client de base et l'objet transactionnel est gérée par un proxy côté client et par un stub côté serveur, comme pour tous les serveurs hors processus. Les informations sur la connexion sont gérées par le proxy. La connexion entre le client de base et le proxy reste ouverte aussi longtemps que le client demande la connexion au serveur. De ce fait, le client croît qu'il bénéficie d'un accès continu au serveur. En réalité, le proxy peut désactiver et réactiver l'objet, conservant des ressources pour que les autres clients puissent utiliser la connexion. Pour plus d'informations sur l'activation et la désactivation, voir "Activation juste-à-temps" à la page 39-4.

Contraintes d'un objet transactionnel

En sus des exigences de COM, un objet transactionnel doit respecter les contraintes suivantes :

- L'objet doit avoir un fabricant de classe standard. Il est fourni automatiquement par l'expert quand vous créez l'objet.
- Le serveur doit présenter son objet classe en exportant la méthode standard *DllGetClassObject*. Le code effectuant ceci est généré par l'expert.
- Toutes les interfaces et les coclasses du composant doivent être décrites par une bibliothèque de types, fournie par l'expert. Vous pouvez ajouter des méthodes et propriétés à la bibliothèque de types en utilisant l'éditeur de bibliothèques de types. La bibliothèque de types est utilisée par l'explorateur MTS ou le gestionnaire de composants COM+ pour en extraire les informations concernant les composants installés au cours de l'exécution.
- Le serveur peut exporter uniquement les interfaces utilisant le marshalling COM standard automatiquement fourni par l'expert Objet transactionnel. La gestion Delphi des objets transactionnels ne permet pas le marshalling manuel des interfaces personnalisées. Toutes les interfaces doivent être implémentées sous la forme d'interfaces doubles qui utilisent la gestion automatique du marshalling par COM.
- Le serveur doit exporter la fonction *DllRegisterServer* et effectuer l'auto-recensement de ses CLSID, ProgID, interfaces et bibliothèque de types dans cette routine. Tout cela est fourni par l'expert objet transactionnel.

Si vous utilisez MTS et non COM+, les conditions suivantes s'appliquent en plus :

- MTS nécessite que le composant soit une DLL. Les serveurs implémentés en tant qu'exécutables (fichiers .EXE) ne peuvent pas s'exécuter dans l'environnement d'exécution MTS.
- L'objet doit implémenter l'interface *IObjectControl*. La gestion de cette interface est ajoutée automatiquement par l'expert objet transactionnel.
- Un serveur exécuté dans l'espace de processus MTS ne peut pas s'agréger à des objets COM qui ne sont pas exécutés dans MTS.

Gestion des ressources

Il est possible d'administrer les objets transactionnels afin de gérer au mieux les ressources utilisées par votre application. Ces ressources sont aussi bien la mémoire utilisée par les instances d'objets elles-mêmes que toutes les ressources qu'elles utilisent (comme les connexions aux bases de données).

Généralement, vous déterminez la manière dont votre application gère les ressources par la façon dont vous installez et configurez votre objet. Vous pouvez configurer votre objet transactionnel afin qu'il bénéficie des caractéristiques suivantes :

- Activation juste-à-temps
- Regroupement des ressources
- Regroupement d'objets (COM+ uniquement)

Mais, pour que votre objet puisse pleinement bénéficier des avantages de ces services, vous devez utiliser l'interface *IObjectContext* pour indiquer à quel moment les ressources peuvent être libérées en toute sécurité.

Accès au contexte d'un objet

Comme tout objet COM, un objet transactionnel doit être créé avant d'être utilisé. Les clients COM créent un objet en appelant la fonction *CoCreateInstance* de la bibliothèque COM.

Chaque objet transactionnel doit correspondre à un objet de contexte. Cet objet de contexte est implémenté automatiquement par MTS et COM+. Il est utilisé pour gérer l'objet transactionnel. L'interface de l'objet de contexte est *IObjectContext*. Pour accéder à la majorité des méthodes de l'objet de contexte, vous pouvez utiliser la propriété *ObjectContext* de l'objet *TMtsAutoObject*. Par exemple, vous pouvez utiliser la propriété *ObjectContext* comme suit :

```
if ObjectContext.IsCallerInRole ('Manager') ...
```

Il existe une autre façon d'accéder à l'objet de contexte : utiliser les méthodes de l'objet *TMtsAutoObject* :

```
if IsCallerInRole ('Manager') ...
```

Vous pouvez utiliser l'une ou l'autre des méthodes précédentes. Cependant, il existe un léger avantage à utiliser les méthodes *TMtsAutoObject* plutôt qu'à faire référence à la propriété *ObjectContext* lorsque vous testez votre application. Pour la présentation de ces différences, voir "Débogage et test des objets transactionnels" à la page 39-23.

Activation juste-à-temps

La capacité d'un objet à être désactivé et réactivé alors que des clients continuent à lui faire référence est nommée **activation juste-à-temps**. Vu du client, il existe une seule instance de l'objet à partir du moment où le client l'a créée et jusqu'au

moment où il la libère. En réalité, il se peut que l'objet ait été désactivé et réactivé de nombreuses fois. Les objets pouvant être ainsi désactivés, les clients peuvent continuer de faire référence à un objet pendant un temps indéterminé sans affecter les ressources du système. Lorsqu'un objet devient désactivé, toutes les ressources de l'objet sont libérées. Ainsi quand un objet est désactivé, sa connexion à la base de données est libérée et utilisable par d'autres clients.

Un objet transactionnel est créé à l'état désactivé et ne devient actif qu'à la réception d'une requête d'un client. Quand l'objet transactionnel est créé, un objet de contexte est également créé. Cet objet de contexte existe pendant l'intégralité de la durée de vie de l'objet transactionnel correspondant, pendant un ou plusieurs cycles de réactivation. L'objet de contexte, accessible via l'interface *IObjectContext* permet de conserver une trace de l'objet quand il est désactivé ou pendant les transactions de coordination.

Les objets transactionnels sont désactivés dès que cela peut se faire sans risque. C'est la **désactivation dès-que-possible**. Un objet transactionnel est désactivé lorsqu'un des événements suivants se produit :

- **L'objet demande la désactivation avec *SetComplete* ou *SetAbort* :** Un objet appelle la méthode *SetComplete* de *IObjectContext* lorsqu'il a terminé son travail avec succès et qu'il n'a pas besoin de sauver l'état interne de l'objet pour un prochain appel du client. Un objet appelle *SetAbort* pour indiquer qu'il ne peut terminer son travail avec succès et que l'état de l'objet n'a pas besoin d'être sauvé, c'est-à-dire que l'objet reprend l'état qu'il avait avant la transaction. Souvent, les objets sont conçus **sans état**, ce qui signifie que les objets sont désactivés après le retour de toutes les méthodes.
- **Une transaction est validée ou annulée :** Lorsqu'une transaction sur un objet est validée ou annulée, l'objet est désactivé. Parmi ces objets désactivés, les seuls qui continuent à exister sont ceux à qui des clients hors transaction sont en train de faire référence. Les prochains appels à ces objets les réactivent et les font s'exécuter dans la prochaine transaction.
- **Le dernier client libère l'objet :** Bien sûr, lorsqu'un client libère l'objet, il est désactivé et l'objet de contexte également.

Remarque Quand vous installez un objet transactionnel sous COM+ depuis l'EDI, vous pouvez spécifier si l'objet utilise l'activation juste-à-temps en utilisant la page COM+ de l'éditeur de bibliothèques de types. Sélectionnez l'objet (CoClasse) dans l'éditeur de bibliothèques de types, allez sur la page COM+ et cochez ou non la case Activation Just In Time. Sinon, un administrateur système peut spécifier cet attribut en utilisant le gestionnaire de composants COM+ ou l'Explorateur MTS (l'administrateur système peut également redéfinir tous les paramètres que vous avez spécifiés en utilisant l'éditeur de bibliothèques de types).

Regroupement des ressources

Comme les ressources système inactives libérées lors de la désactivation, les ressources libérées sont disponibles pour d'autres objets serveur. Ainsi, une

connexion à une base de données qui n'est plus utilisée par un objet serveur peut être réutilisée par un autre client. C'est ce que l'on appelle le **regroupement des ressources**. Les ressources regroupées sont gérées par un fournisseur de ressources.

Un fournisseur de ressources place les ressources dans un cache, afin que les objets transactionnels installés ensemble puissent les partager. Le fournisseur de ressources gère également les informations d'état partagées non permanentes. Ainsi, les fournisseurs de ressources fonctionnent de la même manière que les gestionnaires de ressources (comme SQL Server) mais sans garantir la permanence.

Pour écrire vos objets transactionnels, vous pouvez utiliser deux types de fournisseurs de ressources déjà définis :

- Les fournisseurs de ressources base de données
- Le gestionnaire de propriétés partagées

Avant que d'autres objets puissent utiliser des ressources regroupées, vous devez explicitement les libérer.

Fournisseurs de ressources base de données

L'ouverture et la fermeture de connexions de bases de données est une opération qui demande du temps. En utilisant un fournisseur de ressources pour regrouper les connexions de bases de données, votre objet peut réutiliser des connexions de bases de données au lieu d'en créer de nouvelles. Si, par exemple, vous utilisez une référence de base de données et un composant de mise à jour de la base dans une application de gestion des clients, vous pouvez installer ces composants ensembles. Ainsi, ils peuvent partager les connexions de bases de données. De cette manière, votre application n'a pas besoin d'avoir un grand nombre de connexions et les nouvelles instances d'objets peuvent accéder aux données plus rapidement en utilisant une connexion déjà ouverte mais inutilisée.

- Si vous utilisez les composants BDE pour vous connecter aux données, le fournisseur de ressources est le moteur de bases de données Borland (BDE). Ce fournisseur de ressources est disponible uniquement quand votre objet transactionnel est installé dans MTS. Pour activer le fournisseur de ressources, utilisez le programme d'administration BDE pour activer l'option MTS POOLING dans la zone de configuration Système/Init.
- Si vous utilisez les composants base de données ADO pour vous connecter à vos données, le fournisseur de ressources est fourni par ADO.

Remarque Il n'y a pas de fournisseur de ressources prédéfini si vous utilisez les composants InterbaseExpress pour accéder à vos données.

Pour les modules de données transactionnels distants, les connexions sont automatiquement intégrées dans des objets transactions et le fournisseur de ressources peut automatiquement récupérer et réutiliser les connexions.

Gestionnaire de propriétés partagées

Le gestionnaire de propriétés partagées est un fournisseur de ressources que vous pouvez utiliser pour partager un état entre plusieurs objets au sein d'un processus serveur. Le gestionnaire de propriétés partagées vous évite l'ajout à votre application d'une grande quantité de code pour gérer les données partagées : le gestionnaire de propriétés partagées conserve l'état des objets en implémentant des verrous et des sémaphores afin de protéger des accès simultanés les propriétés partagées. Le gestionnaire de propriétés partagées évite les conflits de noms en utilisant des **groupes de propriétés partagées** qui établissent des domaines d'appellation uniques pour les propriétés partagées qu'ils contiennent.

Pour utiliser la ressource gestionnaire de propriétés partagées, utilisez d'abord la fonction d'aide *CreateSharedPropertyGroup* pour créer un groupe de propriétés partagées. Ensuite, vous pouvez écrire et lire toutes les propriétés de ce groupe. En utilisant un groupe de propriétés partagées, les informations de l'état sont sauvegardées au travers de toutes les désactivations de l'objet transactionnel. En outre, les informations d'état peuvent être partagées entre tous les objets transactionnels installés dans le même paquet. Vous pouvez installer les objets transactionnels dans un paquet comme décrit dans "Installation d'objets transactionnels" à la page 39-24.

Pour que les objets puissent partager un état, ils doivent s'exécuter dans le même processus. Si vous voulez que des instances de différents composants partagent des propriétés, vous devez installer les composants dans le même paquet MTS ou la même application COM+. Mais comme les administrateurs risquent de déplacer les composants d'un paquet à un autre, le plus sûr est de limiter l'utilisation d'un groupe de propriétés partagées aux instances des composants définies dans la même DLL ou le même EXE.

Les composants partageant des propriétés doivent avoir le même attribut d'activation. Si deux composants du même paquet ont des attributs d'activation différents, ils ne pourront généralement pas partager de propriétés. Par exemple, si un composant est configuré pour s'exécuter dans un processus client et l'autre dans un processus serveur, leurs objets s'exécuteront habituellement dans des processus différents, même s'ils appartiennent au même paquet MTS ou à la même application COM+.

L'exemple suivant illustre la manière d'ajouter du code pour gérer le gestionnaire de propriétés partagées dans un objet transactionnel.

Exemple : partage de propriétés entre les instances d'un objet transactionnel

L'exemple suivant crée un groupe de propriétés appelé MyGroup afin qu'il contienne les propriétés à partager entre les objets et les instances des objets. Cet exemple illustre le partage d'une propriété appelée Counter. Il utilise la fonction utilitaire *CreateSharedPropertyGroup* pour créer le gestionnaire du groupe de propriétés et le groupe de propriétés, puis la méthode *CreateProperty* de l'objet Group pour créer la propriété Counter.

Pour obtenir la valeur d'une propriété, utilisez la méthode *PropertyByName* de l'objet Group, comme ci-dessous. Vous pouvez également utiliser la méthode *PropertyByPosition*.

```
unit Unit1;
interface
uses
  MtsObj, Mtx, ComObj, Project2_TLB;
type
  Tfoobar = class(TMtsAutoObject, Ifoobar)
  private
    Group: ISharedPropertyGroup;
  protected
    procedure OnActivate; override;
    procedure OnDeactivate; override;
    procedure IncCounter;
  end;
implementation
uses ComServ;
{ Tfoobar }
procedure Tfoobar.OnActivate;
var
  Exists: WordBool;
  Counter: ISharedProperty;
begin
  Group := CreateSharedPropertyGroup('MyGroup');
  Counter := Group.CreateProperty('Counter', Exists);
end;
procedure Tfoobar.IncCounter;
var
  Counter: ISharedProperty;
begin
  Counter := Group.PropertyByName['Counter'];
  Counter.Value := Counter.Value + 1;
end;
procedure Tfoobar.OnDeactivate;
begin
  Group := nil;
end;
initialization
  TAutoObjectFactory.Create(ComServer, Tfoobar, Class_foobar, ciMultiInstance,
tmApartment);
end.
```

Libération des ressources

Vous êtes responsable de la libération des ressources d'un objet. Habituellement, vous le faites en appelant les méthodes *SetComplete* et *SetAbort* de *IObjectContext* après avoir servi la demande de chaque client. Ces méthodes libèrent les ressources allouées par le fournisseur de ressources.

Au même moment, vous devez libérer les références à toutes les autres ressources, y compris les références aux autres objets (notamment les objets

transactionnels et les objets de contexte) ainsi que la mémoire occupée par toutes les instances du composant (libération du composant).

La seule fois où vous omettrez ces appels, c'est lorsque vous voudrez maintenir l'état entre les appels client. Pour plus d'informations, voir "Objets avec état et sans état" à la page 39-12.

Regroupement d'objets

Tout comme vous pouvez regrouper des ressources, vous pouvez avec COM+ regrouper des objets. Quand un objet est désactivé, COM+ appelle la méthode *CanBePooled* de l'interface, qui indique que l'objet peut être regroupé afin d'être réutilisé. Si *CanBePooled* renvoie *True*, au lieu d'être détruit quand il est désactivé, l'objet est déplacé dans le groupe d'objets. Il y reste pour une durée limitée durant laquelle il est disponible si un client en fait la demande. C'est seulement quand le groupe d'objets est vide qu'une nouvelle instance de l'objet est créée. Les objets qui renvoient *False* ou qui ne gèrent pas l'interface *IObjectControl* sont détruits quand il sont désactivés.

Le regroupement d'objets n'est pas disponible avec MTS. MTS appelle *CanBePooled* comme indiqué, mais le regroupement n'est pas effectué. Si votre objet est utilisé uniquement avec COM+ et si vous voulez utiliser le regroupement d'objets, définissez la propriété *Pooled True*.

Même si la méthode *CanBePooled* d'un objet renvoie *True*, il peut être configuré afin que COM+ ne le place pas dans le regroupement d'objets. Si vous installez l'objet transactionnel dans COM+ depuis l'EDI, vous pouvez spécifier si COM+ doit essayer de regrouper l'objet en utilisant la page COM+ de l'éditeur de bibliothèques de types. Sélectionnez l'objet (la coclasse) dans l'éditeur de bibliothèques de types, allez sur la page COM+ et cochez ou non la case Pooling d'objets. Par ailleurs, un administrateur système peut spécifier cet attribut en utilisant le gestionnaire de composant COM+.

De la même façon, vous pouvez configurer la durée durant laquelle un objet désactivé reste dans le regroupement d'objets avant d'être libéré. Pour ce faire, utilisez le paramètre Délai maxi de création dans la page COM+ de l'éditeur de bibliothèques de types. Ici encore, l'administrateur système peut spécifier cet attribut en utilisant le gestionnaire de composants COM+.

Support transactionnel MTS et COM+

Le support des transactions qui donne aux objets transactionnels leur nom permet de grouper des actions au sein des transactions. Par exemple, dans une application de dossiers médicaux, si vous avez un composant Transfert pour transférer des dossiers d'un médecin à l'autre, vous pouvez avoir vos méthodes Ajouter et Supprimer dans la même transaction. De cette façon, soit l'intégralité de Transfert fonctionne, soit il peut être ramené à son état précédent. Les transactions simplifient les corrections d'erreurs dans les applications devant accéder à *plusieurs* bases de données.

Les transactions garantissent ce qui suit :

- Toutes les mises à jour d'une même transaction sont soit validées, soit annulées et ramenées à leur état précédent. C'est ce que l'on appelle **atomicité**.
- Une transaction est une transformation correcte de l'état d'un système, en maintenant les invariants de cet état. C'est ce que l'on appelle **consistance**.
- Les transactions simultanées ne voient pas les résultats partiels et non validés les unes des autres afin de ne pas créer d'incohérences dans l'état de l'application. C'est ce que nous appelons **isolation**. Les gestionnaires de ressources utilisent des protocoles de synchronisation basés sur les transactions pour isoler les travaux non validés des transactions actives.
- Les mises à jour validées des ressources gérées (comme les enregistrements d'une base de données) survivent aux pannes, y compris les pannes de communication, les pannes de processus et les pannes système des serveurs. C'est ce que nous appelons **durabilité**. L'ouverture d'une transaction vous permet de récupérer l'état durable après des pannes survenues sur les supports disque.

L'objet de contexte associé à l'objet indique s'il s'exécute dans une transaction et, si c'est le cas, l'identité de la transaction. Quand un objet fait partie d'une transaction, les services proposés par les gestionnaires de ressources et les fournisseurs de ressources s'exécutent également dans la transaction. Les fournisseurs de ressources utilisent l'objet de contexte pour proposer des services à base de transaction. Par exemple, quand un objet exécuté dans une transaction alloue une connexion de base de données en utilisant le fournisseur de ressources ADO ou BDE, la connexion est automatiquement intégrée dans la transaction. Toutes les modifications de bases de données effectuées en utilisant cette connexion font également partie de la transaction et sont donc soit validées ou annulées avec la transaction.

Le travail de plusieurs objets peut être réalisé à l'intérieur d'une seule transaction. Permettre à un objet de résider dans sa propre transaction ou de faire partie d'un groupe d'objets plus large appartenant à une seule transaction est l'un des avantages les plus importants de MTS et COM+. Cela permet d'utiliser un objet de différentes manières ; ainsi le concepteur d'applications peut réutiliser le code dans d'autres applications sans réécrire la logique de l'application. En fait, les développeurs peuvent déterminer comment les objets sont utilisés dans les transactions au moment de l'installation de l'objet transactionnel. Ils peuvent modifier le comportement de transaction simplement en ajoutant un objet à un paquet MTS différent ou à une autre application COM+. Pour davantage d'informations sur l'installation des objets transactionnels, voir "Installation d'objets transactionnels" à la page 39-24.

Attributs transactionnels

Chaque objet transactionnel a un attribut transactionnel stocké dans le catalogue MTS ou recensé dans COM+.

Delphi permet de définir l'attribut transactionnel à la conception en utilisant l'expert objet transactionnel ou l'éditeur de bibliothèques de types.

Chaque attribut transactionnel peut être défini par une des valeurs suivantes :

Requiert une transaction	Les objets doivent s'exécuter *dans la portée d'une transaction*. Lorsqu'un nouvel objet est créé, son contexte hérite de la transaction du contexte du client. Si le client n'a pas de contexte transactionnel, un nouveau contexte transactionnel est automatiquement créé pour l'objet.
Requiert une nouvelle transaction	Les objets doivent s'exécuter *dans leurs propres transactions*. Lorsqu'un nouvel objet est créé, une nouvelle transaction est créée automatiquement pour l'objet, que son client ait ou non une transaction. Un objet ne s'exécute jamais dans la portée de la transaction de son client. Au contraire, le système crée toujours des transactions indépendantes pour les nouveaux objets.
Supporte les transactions	Les objets peuvent s'exécuter *dans la portée des transactions de leur client*. Lorsqu'un nouvel objet est créé, son contexte hérite de la transaction du contexte du client. Cela permet à plusieurs objets d'être rassemblés dans une même transaction. Si le client n'a pas de transaction, le nouveau contexte est également créé sans transaction.
Transactions ignorées	Les objets *ne s'exécutent pas dans la portée des transactions*. Lorsqu'un nouvel objet est créé, son objet de contexte est créé sans transaction, que son client ait ou non une transaction. Cette option est disponible uniquement avec COM+.
Ne supporte pas les transactions	La signification de cette option change selon que l'objet est installé dans MTS ou dans COM+. Avec MTS, cette option a le même effet que Transactions ignorées pour COM+. Dans COM+, non seulement l'objet de contexte est créé dans transaction, mais cette valeur empêche l'activation de l'objet si le client a déjà une transaction.

Initialisation de l'attribut transactionnel

Vous pouvez définir un attribut transactionnel à la création de l'objet transactionnel en utilisant l'expert objet transactionnel.

Vous pouvez également définir (ou modifier) l'attribut transactionnel en utilisant l'éditeur de bibliothèques de types. Pour modifier l'attribut transactionnel en utilisant l'éditeur de bibliothèques de types :

1. Choisissez Voir | Bibliothèque de types pour ouvrir l'éditeur de bibliothèques de types.
2. Sélectionnez la classe correspondant à l'objet transactionnel.
3. Cliquez dans la page COM+ et choisissez l'attribut transactionnel souhaité.

Attention Quand vous spécifiez l'attribut transactionnel, Delphi insère un GUID spécial pour l'attribut spécifié comme donnée personnalisée dans la bibliothèque de types. Cette valeur n'est pas reconnue en dehors de Delphi. Elle n'a d'effet que si vous installez l'objet transactionnel depuis l'EDI. Sinon, un administrateur système doit définir cette valeur en utilisant l'explorateur MTS ou le gestionnaire de composants COM+.

Remarque Si l'objet transactionnel est déjà installé, vous devez tout d'abord le désinstaller puis le réinstaller quand vous modifiez l'attribut transactionnel. Pour ce faire, utilisez Exécuter | Installer les objets MTS ou Exécuter | Installer les objets COM+.

Objets avec état et sans état

Comme tous les objets COM, les objets transactionnels sont capables de maintenir un état interne au cours de leurs multiples interactions avec un client. Par exemple, un client peut définir la valeur d'une propriété lors d'un appel et s'attendre à ce que cette valeur reste inchangée lors de l'appel suivant. Un tel objet est dit **avec état**. Les objets transactionnels peuvent également être **sans état**, ce qui signifie que l'objet ne maintient aucun état intermédiaire pendant l'attente du prochain appel d'un client.

Lorsqu'une transaction est validée ou annulée, tous les objets qui étaient inclus dans la transaction sont désactivés, ce qui fait qu'ils perdent l'état qu'ils avaient acquis au cours de la transaction. C'est un des mécanismes qui garantissent l'isolation des transactions et la cohérence des données ; cela libère également les ressources du serveur pour leur utilisation par d'autres transactions. La fin d'une transaction permet de désactiver un objet et de récupérer ses ressources. Pour davantage d'informations sur la manière dont l'état de l'objet est libéré, voir la section suivante.

Conserver l'état d'un objet impose que cet objet reste activé, conservant ainsi des ressources potentiellement utiles, comme des connexions de bases de données.

Contrôle de l'arrêt des transactions

Un objet transactionnel utilise les méthodes de *IObjectContext* comme le montre le tableau suivant pour déterminer la façon dont se termine une transaction. Ces méthodes, avec l'attribut transactionnel du composant, vous permettent d'engager un ou plusieurs objets dans une transaction.

Tableau 39.1 Méthodes de IObjectContext pour la gestion des transactions

Méthode	Description
SetComplete	Indique que l'objet a terminé son travail avec succès pour la transaction. L'objet est désactivé après le retour de la méthode entrée la première dans le contexte. L'objet est réactivé lors du prochain appel nécessitant son exécution.
SetAbort	Indique que le travail de l'objet ne pourra jamais être validé. L'objet est désactivé au retour de la méthode entrée la première dans le contexte. L'objet est réactivé lors du prochain appel nécessitant son exécution.

Tableau 39.1 Méthodes de IObjectContext pour la gestion des transactions (suite)

Méthode	Description
EnableCommit	Indique que le travail de l'objet n'est pas obligatoirement terminé mais que les mises à jour transactionnelles peuvent être validées dans leur forme actuelle. Utilisez cela pour conserver l'état entre les multiples appels d'un client tout en permettant l'achèvement des transactions. L'objet n'est pas désactivé jusqu'à ce qu'il appelle SetComplete ou SetAbort. EnableCommit est l'état par défaut lorsqu'un objet est activé. C'est pourquoi un objet doit *toujours appeler SetComplete ou SetAbort avant de retourner d'une méthode*, sauf si vous voulez que l'objet conserve son état interne pour le prochain appel d'un client.
DisableCommit	Indique que le travail de l'objet est incohérent et qu'il ne peut se terminer tant que l'objet n'a pas reçu de nouveaux appels de méthodes de la part du client. Appelez cette méthode avant de rendre le contrôle au client afin de conserver l'état entre plusieurs appels client tout en conservant la transaction active. DisableCommit empêche l'objet de se désactiver et de libérer ses ressources au retour de l'appel d'une méthode. Lorsqu'un objet a appelé DisableCommit, si un client tente de valider la transaction avant que l'objet ait appelé EnableCommit ou SetComplete, la transaction est annulée.

Démarrage des transactions

Il est possible de contrôler les transactions de trois manières :

- Elles peuvent être contrôlées par le client.

  Les clients ont un contrôle direct sur les transactions en utilisant un objet contexte de transaction (en employant l'interface *ITransactionContext*).

- Elles peuvent être contrôlées par le serveur.

  Les serveurs peuvent contrôler explicitement les transactions en créant pour elles un objet contexte. Quand le serveur crée ainsi un objet, l'objet créé est automatiquement engagé dans la transaction en cours.

- Les transactions peuvent se produire automatiquement comme résultat des attributs transactionnels de l'objet.

  Il est possible de déclarer les objets transactionnels de telle manière que leurs objets s'exécutent toujours dans une transaction, sans tenir compte de la manière dont les objets sont créés. De cette manière, les objets n'ont pas besoin d'inclure la moindre logique de gestion des transactions. Cette caractéristique réduit également la charge des applications client. Les clients n'ont pas besoin de démarrer une transaction uniquement parce que le composant qu'elles utilisent en a besoin.

Définition d'un objet transaction côté client

Une application basée sur le client de base peut contrôler le contexte transactionnel via l'interface *ITransactionContextEx*. L'exemple de code suivant montre comment une application client peut utiliser *CreateTransactionContextEx*

pour créer le contexte transactionnel. Cette méthode renvoie une interface vers cet objet.

Cet exemple insère l'appel au contexte transactionnel dans un appel de OleCheck, nécessaire car les méthodes de *IObjectContext* sont exposées directement par Windows et ne sont donc pas déclarées **safecall**.

```
procedure TForm1.MoveMoneyClick(Sender: TObject);
begin
  Transfer(CLASS_AccountA, CLASS_AccountB, 100);
end;
procedure TForm1.Transfer(DebitAccountId, CreditAccountId: TGuid; Amount: Currency);
var
  TransactionContextEx: ITransactionContextEx;
  CreditAccountIntf, DebitAccountIntf: IAccount;
begin
  TransactionContextEx := CreateTransactionContextEx;
  try
    OleCheck(TransactionContextEx.CreateInstance(DebitAccountId,
      IAccount, DebitAccountIntf));
    OleCheck(TransactionContextEx.CreateInstance(CreditAccountId,
      IAccount, CreditAccountIntf));
    DebitAccountIntf.Debit(Amount);
    CreditAccountIntf.Credit(Amount);
  except
    TransactionContextEx.Abort;
    raise;
  end;
  TransactionContextEx.Commit;
end;
```

Définition d'un objet transaction côté serveur

Pour contrôler le contexte transactionnel depuis le serveur, créez une instance d'*ObjectContext*. Dans l'exemple suivant, la méthode Transfer est dans l'objet transactionnel. En utilisant *ObjectContext* de cette façon, l'instance de l'objet que nous sommes en train de créer hérite de tous les attributs transactionnels de l'objet qui le crée. L'appel est inséré dans un appel de *OleCheck* car les méthodes de *IObjectContext* sont exposées directement par Windows et ne sont donc pas déclarées safecall.

```
procedure TAccountTransfer.Transfer(DebitAccountId, CreditAccountId: TGuid;
 Amount: Currency);
var
  CreditAccountIntf, DebitAccountIntf: IAccount;
begin
  try
    OleCheck(ObjectContext.CreateInstance(DebitAccountId,
      IAccount, DebitAccountIntf));
    OleCheck(ObjectContext.CreateInstance(CreditAccountId,
      IAccount, CreditAccountIntf));
    DebitAccountIntf.Debit(Amount);
    CreditAccountIntf.Credit(Amount);
  except
    DisableCommit;
```

```
    raise;
  end;
  EnableCommit;
end;
```

Délais des transactions

Le délai des transactions définit la durée (exprimée en secondes) durant laquelle une transaction peut rester active. Les transactions toujours vivantes après ce délai sont automatiquement annulées par le système. Par défaut, cette valeur est de 60 secondes. Vous pouvez désactiver le délai des transactions en spécifiant la valeur 0, cela peut être utile lors du débogage des objets transactionnels.

Pour définir la temporisation sur votre ordinateur :

1. Dans l'explorateur MTS ou le gestionnaire de composants COM+, sélectionnez Computer, My Computer.

   Par défaut, My Computer correspond au système local.

2. Cliquez avec le bouton droit de la souris et choisissez Properties, puis l'onglet Options.

   L'onglet Options est utilisé pour définir la propriété Transaction timout (délai de transaction) pour le système.

3. Changez la valeur de la propriété timeout à 0 pour désactiver les délais de transaction.

4. Choisissez OK pour enregistrer la configuration.

Pour davantage d'informations sur le débogage des applications MTS, voir "Débogage et test des objets transactionnels" à la page 39-23.

Sécurité en fonction des rôles

MTS et COM+ offrent une sécurité en fonction des rôles dans laquelle vous attribuez un rôle à chaque groupe d'utilisateurs. Par exemple, notre application médicale pourrait définir un rôle pour les médecins, un pour les techniciens radiologues et un pour les patients.

Vous définissez des autorisations pour chaque composant et chaque interface de composant en assignant des rôles. Par exemple, dans une application médicale, seuls les médecins seraient autorisés à voir tous les dossiers médicaux, les techniciens radio ne verraient que les radiographies, et les patients leur propre dossier.

Habituellement, les rôles sont définis au cours du développement de l'application et ils sont attribués à chaque paquet MTS ou chaque application COM+. Ils sont ensuite assignés à des utilisateurs spécifiques lorsque l'application est déployée. Les administrateurs peuvent configurer les rôles en utilisant l'explorateur MTS ou le gestionnaire de composants COM+.

Si vous voulez contrôler l'accès à des blocs de code et non à des objets entiers, vous pouvez proposer une sécurité plus affinée en utilisant la méthode *IsCallerInRole* de *IObjectContext*. Cette méthode ne fonctionne que si la sécurité est activée, ce qui peut être testé en appelant la méthode *IsSecurityEnabled* de *IObjectContext*. Ces méthodes sont automatiquement ajoutées comme méthodes de votre objet transactionnel. Par exemple :

```
if IsSecurityEnabled then { Tester si la sécurité est activée }
begin
  if IsCallerInRole('Médecin') then { Tester le rôle de l'appelant }
  begin
    { Exécuter l'appel normalement. }
  end
  else
    { Si ce n'est pas un médecin, effectuer l'opération appropriée en ce cas. }
  end
end
else
  { La sécurité n'est pas activée, effectuer l'opération appropriée }
end;
```

Remarque Pour les applications nécessitant une sécurité renforcée, les objets contexte implémentent l'interface *ISecurityProperty* dont les méthodes permettent d'obtenir l'identificateur de sécurité Windows (SID) de l'appelant direct et créateur de l'objet ainsi que le SID des clients utilisant l'objet.

Présentation de la création des objets transactionnels

La création des objets transactionnels suit le processus suivant :

1. Utilisez l'expert objet transactionnel pour créer l'objet transactionnel.
2. Ajoutez les méthodes et propriétés à l'interface de l'objet en utilisant l'éditeur de bibliothèques de types. Pour des informations sur l'ajout de propriétés et de méthodes en utilisant l'éditeur de bibliothèques de types, voir chapitre 34, "Utilisation des bibliothèques de types".
3. Quand vous implémentez les méthodes de l'objet, vous pouvez utiliser l'interface *IObjectContext* pour gérer les transactions, la persistance de l'état et la sécurité. De plus, si vous transmettez des références d'objet, vous devez bien faire attention à ce qu'elles soient correctement gérées (voir "Transfert de références d'objets" à la page 21).
4. Débogez et testez l'objet transactionnel.
5. Installez l'objet transactionnel dans un paquet MTS ou une application COM+.
6. Administrez vos objets en utilisant l'explorateur MTS ou le gestionnaire de composants COM+.

Utilisation de l'expert objet transactionnel

Utilisez l'expert objet transactionnel pour créer un objet COM exploitant la gestion des ressources, le traitement des transactions et la sécurité en fonction de rôles que propose MTS ou COM+.

Pour afficher l'expert objet transactionnel

1. Choisissez Fichier | Nouveau | Autre.
2. Sélectionnez l'onglet libellé Multi-niveaux.
3. Cliquez deux fois sur l'icône Objet transactionnel.

Dans l'expert, vous devez spécifier :

- Le modèle de thread afin d'indiquer comment les applications client peuvent appeler l'interface de votre objet. Le modèle de thread détermine comment l'objet sera recensé. Vous devez être certain que l'implémentation de votre objet est conforme au modèle sélectionné. Pour plus d'informations sur les modèles de threads, voir "Choix d'un modèle de thread pour un objet transactionnel" à la page 39-18.
- Un modèle de transaction.
- Si votre objet notifie les clients des événements. La gestion des événements est proposée uniquement pour les événements classiques, pas pour les événements COM+.

Lorsque vous suivez cette procédure, une nouvelle unité est ajoutée au projet en cours, elle contient la définition de l'objet transactionnel. De plus, l'expert ajoute une bibliothèque de types au projet et l'ouvre dans l'éditeur de bibliothèques de types. Vous pouvez alors exposer les propriétés et les méthodes de l'interface par le biais de la bibliothèque de types. Vous exposez l'interface comme vous exposeriez n'importe quel objet Automation comme décrit dans "Définition de l'interface d'un objet COM" à la page 36-9.

L'objet transactionnel implémente une **interface double**, qui supporte à la fois la liaison précoce (à la compilation), via la vtable, et la liaison tardive (à l'exécution), via l'interface *IDispatch*.

L'objet transactionnel généré implémente les méthodes de l'interface *IObjectControl*, Activate, Deactivate et CanBePooled.

Il n'est pas absolument nécessaire d'utiliser l'expert objet transactionnel. Vous pouvez convertir tout objet Automation en un objet transactionnel COM+ (et tout objet Automation en processus, en un objet transactionnel MTS) en utilisant la page COM+ de l'éditeur de bibliothèques de types, puis en installant l'objet dans un paquet MTS ou une application COM+. Cependant, l'expert objet transactionnel propose certains avantages :

- Il implémente automatiquement l'interface *IObjectControl*, et ajoute les événements *OnActivate* et *OnDeactivate* aux objets afin que vous puissiez créer des gestionnaires d'événements qui répondent à l'activation ou la désactivation de l'objet.

- Il génère automatiquement une propriété ObjectContext pour simplifier l'accès de l'objet aux méthodes de *IObjectContext* afin de contrôler l'activation et les transactions.

Choix d'un modèle de thread pour un objet transactionnel

Les environnements d'exécution MTS et COM + gèrent les threads à votre place. Les objets transactionnels ne doivent pas créer de threads. Ils ne doivent jamais terminer un thread qui fait un appel dans une DLL.

Lorsque vous spécifiez le modèle de thread en utilisant l'expert objet transactionnel, vous spécifiez comment les objets seront attribués aux threads lors de l'exécution des méthodes.

Tableau 39.2 Modèles de thread pour les objets transactionnels

Modèle de thread	Description	Avantages et inconvénients
Unique	Pas de gestion des threads. Les demandes client sont sérialisées par le mécanisme d'appel. Tous les objets d'un composant à thread unique s'exécutent dans le thread principal. Cela est compatible avec le modèle de thread COM par défaut utilisé pour les composants ne possédant pas d'attribut de registre Modèle de thread et pour les composants COM non réentrants. L'exécution des méthodes est sérialisée entre tous les objets d'un composant et entre tous les composants d'un processus.	Permet aux composants d'utiliser des bibliothèques non réentrantes. Capacités d'évolution extrêmement limitées. Les composants à thread unique et avec état sont prédisposés aux verrous mortels. Vous pouvez éliminer ce problème en utilisant des objets sans état et en appelant SetComplete avant de revenir de chacune des méthodes.
Apartment (ou apartment à thread unique)	Chaque objet est assigné à un thread apartment, dont la durée équivaut à la durée de vie de l'objet ; cependant, plusieurs threads peuvent être utilisés pour plusieurs objets. C'est le modèle concurrentiel COM standard. Chaque apartment est lié à un thread spécifique et possède une pompe à messages Windows.	Apporte une nette amélioration des accès simultanés par rapport au modèle de thread unique. Deux objets peuvent s'exécuter simultanément du moment qu'ils appartiennent à des activités différentes. Semblable à l'apartment COM, mais les objets peuvent être distribués dans plusieurs processus.

Remarque Ces modèles de threads sont semblables à ceux définis par les objets COM. Cependant, comme MTS et COM+ fournissent une meilleure gestion des threads, la signification de chaque modèle de thread diffère. D'autre part, le modèle libre ne s'applique pas aux objets transactionnels à cause de la gestion intégrée des activités.

Activités

Outre le modèle de thread, les objets transactionnels supporte les accès simultanés via les **activités**. Les activités sont enregistrées dans le contexte de l'objet et l'association entre un objet et une activité ne peut être modifiée. Une activité comprend l'objet transactionnel créé par le client de base, ainsi que tous les objets transactionnels créés par cet objet et ses descendants. Ces objets peuvent être distribués dans un ou plusieurs processus s'exécutant sur un ou plusieurs ordinateurs.

Par exemple, une application destinée à des médecins peut posséder un objet transactionnel ajoutant des mises à jour et supprimant des enregistrements dans plusieurs bases de données médicales, chacune représentée par un objet différent. Cet objet d'ajout peut utiliser également d'autres objets, comme un objet de réception qui enregistre la transaction. Il en résulte plusieurs objets transactionnels qui sont, soit directement, soit indirectement, sous le contrôle du client de base. Ces objets appartiennent tous à la même activité.

MTS et COM+ suivent le flux d'exécution dans chaque activité, en empêchant qu'un phénomène de parallélisme n'altère malencontreusement l'état de l'application. Le résultat de cette fonctionnalité est l'unicité du thread logique d'exécution pour toute une série d'objets potentiellement distribuables. Grâce à ce thread logique unique, les applications sont nettement plus faciles à écrire.

Lorsqu'un objet transactionnel est créé à partir d'un contexte existant, en utilisant soit un objet contexte transactionnel ou un objet contexte, le nouvel objet devient membre de la même activité. Autrement dit, le nouveau contexte hérite de l'identificateur d'activité du contexte utilisé pour le créer.

Il ne peut y avoir qu'un seul thread logique d'exécution pour une même activité. Ce comportement est semblable à celui d'un modèle de thread apartment COM, mais les objets peuvent être distribués dans plusieurs processus. Lorsqu'un client de base fait appel à une activité, toutes les autres demandes de travail dans cette activité (par exemple, celles provenant d'un autre thread client) sont bloquées jusqu'à ce que le thread d'exécution initial revienne vers le client.

Avec MTS, chaque objet transactionnel appartient à une activité. Avec COM+, vous pouvez configurer la manière dont l'objet participe aux activités en initialisant la **synchronisation d'appel**. Les options suivantes sont disponibles :

Tableau 39.3 Options d'appel de synchronisation

Option	Signification
Désactivée	COM+ n'affecte pas d'activité à l'objet mais il peut les hériter du contexte de l'appelant. Si l'appelant n'a pas de transaction ou de contexte d'objet, l'objet n'est pas attribué à une activité. Cela donne la même chose que si l'objet n'est pas installé dans une application COM+. Cette option ne doit pas être utilisée si l'un des objets de l'application utilise un gestionnaire de ressources ou si l'objet gère les transactions ou l'activation juste-à-temps.
Non gérée	COM+ n'affecte jamais l'objet à une activité quel que soit le statut de son appelant. Cette option ne doit pas être utilisée si l'un des objets de l'application utilise un gestionnaire de ressources ou si l'objet gère les transactions ou l'activation juste-à-temps.

Tableau 39.3 Options d'appel de synchronisation (suite)

Option	Signification
Gérée	COM+ affecte l'objet à la même activité que l'appelant. Si l'appelant n'appartient pas à une activité, l'objet non plus. Cette option ne doit pas être utilisée si l'un des objets de l'application utilise un gestionnaire de ressources ou si l'objet gère les transactions ou l'activation juste-à-temps.
Obligatoire	COM+ affecte toujours l'objet à une activité, en créant une activité si nécessaire. Cette option doit être utilisée si l'attribut transactionnel est Supporté ou Requiert.
Nouveau Obligatoire	COM+ affecte toujours l'objet à une nouvelle activité distincte de celle de l'appelant.

Génération d'événements dans COM+

Avant COM+, les serveurs Automation utilisaient un ensemble d'interfaces spéciales pour générer des événements. COM+ introduit un nouveau système de gestion des événements. Ce n'est plus l'objet serveur qui gère les événements en mémorisant les clients qui doivent être notifiés et en appelant leurs interfaces quand des événements se produisent. C'est le système sous-jacent, COM+, qui gère ce processus.

Remarque Les objets transactionnels installés dans COM+ peuvent continuer à utiliser l'ancien système de gestion des événements. Néanmoins, laisser COM+ gérer le processus offre une plus grande flexibilité. Ainsi, quand COM+ gère les événements, le client peut être un serveur en processus qui est démarré par COM+ quand l'événement se produit.

Quand un objet COM+ génère des événements, il ne le fait pas directement. Il utilise pour cela un objet événement associé qui a été spécifiquement créé pour générer des événements. L'objet COM+ appelle son objet événement quand il veut déclencher un événement. Quand cela se produit, COM+ appelle tous les clients qui ont manifesté leur intérêt pour cet objet événement particulier.

Utilisation de l'expert objet événement

Vous pouvez créer des objets événements en utilisant l'expert objet événement. L'expert commence par vérifier si le projet en cours contient du code d'implémentation. En effet, les projets contenant des objets événement COM+ ne doivent contenir aucune implémentation. Ils ne peuvent contenir que des définitions d'objets événement (vous pouvez, par contre, placer plusieurs objets événement COM+ dans un même projet).

Pour afficher l'expert objet événement :

1. Choisissez Fichier | Nouveau.
2. Sélectionnez l'onglet libellé ActiveX.
3. Double-cliquez sur l'icône Objet événement COM+.

Dans l'expert événement, spécifiez le nom de l'objet événement, le nom de l'interface qui définit les gestionnaires d'événements et, de manière facultative, une brève description des événements.

Quand vous sortez, l'expert crée un projet contenant une bibliothèque de types qui définit votre objet événement et son interface. Utilisez l'éditeur de bibliothèques de types pour définir les méthodes de cette interface. Ces méthodes sont les gestionnaires d'événements que les clients implémentent pour répondre aux événements.

Le projet objet événement inclut le fichier projet unité _ATL pour importer les modèles de classes ATL et l'unité _TLB pour définir les informations de la bibliothèque de types. Il ne contient aucune unité d'implémentation car les objets événement COM+ n'ont pas d'implémentation. L'implémentation de l'interface est à la charge du client. Quand l'objet serveur appelle un objet événement COM+, COM+ intercepte l'appel et le distribue aux clients recensés. Comme les objets événement COM+ n'ont pas besoin d'objet implémentation, une fois que vous avez défini l'interface de l'objet dans l'éditeur de bibliothèques de types, il vous suffit de compiler le projet et de l'installer dans COM+

COM+ impose certaines restrictions sur les interfaces des objets événement. L'interface définie dans l'éditeur de bibliothèques de types pour l'objet événement doit respecter les règles suivantes :

- L'interface de l'objet événement doit dériver de IDispatch.
- Les noms de méthode doivent être uniques pour toutes les interfaces de l'objet événement.
- Les méthodes de l'interface de l'objet événement doivent toutes renvoyer une valeur HRESULT.
- Le modificateur de toutes les paramètres de méthode doit être blank.

Déclenchement d'événement en utilisant un objet événement COM+

Quand un événement se produit, votre objet COM+ doit appeler l'objet événement et lui demande de déclencher l'événement dans les clients recensés. Pour ce faire, il crée une instance de l'objet événement et appelle la méthode correspondant à l'événement :

Remarque Les objets déclenchant des événements COM+ doivent être, comment les objets événement mêmes, installés dans une application COM+.

Transfert de références d'objets

Remarque Les informations sur le transfert de références d'objets s'appliquent uniquement à MTS, et pas à COM+. Ce mécanisme est requis sous MTS car il est nécessaire de garantir que tous les pointeurs sur des objets s'exécutant sous MTS sont routés

par le biais d'intercepteurs. Puisque les intercepteurs sont construits dans COM+, vous n'avez pas besoin de transmettre des références d'objets.

Dans MTS, vous pouvez passer des références à un objet (par exemple, pour les utiliser en tant que callback) uniquement par un des moyens suivants :

- Avec le retour d'une interface de création d'un objet, comme *CoCreateInstance* (ou son équivalent), *ITransactionContext.CreateInstance* ou *IObjectContext.CreateInstance.*
- Avec un appel à *QueryInterface*.
- Avec une méthode ayant appelé *SafeRef* pour obtenir la référence de l'objet.

Une référence à un objet obtenue par un des moyens précédents est appelée **référence sécurisée**. Les méthodes invoquées en utilisant des références sécurisées s'exécuteront toujours dans le contexte convenable.

Les appels qui utilisent des références sécurisées doivent toujours être transmis à l'environnement d'exécution MTS afin qu'il gère les options de contexte et permette aux objets transactionnels d'avoir des durées de vie indépendantes des références client. Les références sécurisées ne sont pas nécessaires avec COM+.

Utilisation de la méthode SafeRef

Un objet peut utiliser la fonction *SafeRef* pour obtenir une référence sécurisée sur lui-même afin de la transmettre hors de son contexte. La fonction *SafeRef* est définie dans l'unité Mtx.

En entrée, *SafeRef* attend :

- Une référence à l'ID de l'interface (RIID) que l'objet en cours veut transmettre à un autre objet ou à un client.
- Une référence à l'interface IUnknown de l'objet en cours.

SafeRef renvoie un pointeur sur l'interface spécifiée par le paramètre RIID, qui est sécurisé pour passer hors du contexte de l'objet en cours. Elle renvoie **nil** si l'objet demande une référence sécurisée sur un objet autre que lui-même, ou si l'interface requise par le paramètre RIID n'est pas implémentée.

Lorsqu'un objet MTS souhaite transmettre une auto-référence vers un client ou vers un autre objet (par exemple, pour l'utiliser en tant que callback), il doit toujours appeler *SafeRef* d'abord et passer ensuite la référence renvoyée par cet appel. Un objet ne doit jamais passer de pointeur **self**, ni d'auto-référence obtenue par un appel interne à QueryInterface, à un client ou à tout autre objet. Une fois une telle référence transmise hors du contexte de l'objet, ce n'est plus une référence valide.

Appeler *SafeRef* sur une référence déjà sécurisée renvoie la référence sécurisée inchangée, mais le compteur de références sur l'interface est incrémenté.

Lorsqu'un client appelle QueryInterface sur une référence sécurisée, la référence renvoyée au client est également sécurisée.

Un objet qui obtient une référence sécurisée doit libérer la référence sécurisée lorsqu'elle ne lui est plus nécessaire.

Pour plus de détails concernant *SafeRef*, voir la rubrique SafeRef de la documentation MTS.

Callbacks

Les objets peuvent faire des callbacks aux clients et aux autres objets transactionnels. Par exemple, vous pouvez avoir un objet qui crée un autre objet. L'objet créateur peut transmettre à l'objet créé une référence à lui-même ; l'objet créé peut ensuite utiliser cette référence pour appeler l'objet qui l'a créé.

Si vous choisissez d'utiliser les callbacks, tenez compte des restrictions suivantes :

- Un callback au client de base ou à un autre paquet nécessite, sur le client, une sécurité au niveau des accès. De plus, le client doit être un serveur DCOM.
- L'introduction de coupe-feux doit bloquer les callbacks vers le client.
- Le travail effectué par le callback s'exécute dans l'environnement de l'objet qui est appelé. Il peut faire partie de la même transaction, d'une transaction différente ou d'aucune transaction.
- Dans MTS, l'objet créateur doit appeler *SafeRef* et transmettre la référence renvoyée à l'objet créé en vue de se rappeler lui-même.

Débogage et test des objets transactionnels

Vous pouvez déboguer les objets transactionnels locaux et distants. Lors du débogage des objets transactionnels, vous pouvez désactiver la temporisation des transactions.

Le délai des transactions définit la durée (exprimée en secondes) durant laquelle une transaction peut rester active. Les transactions toujours vivantes après ce délai sont automatiquement annulées par le système. Par défaut, cette valeur est de 60 secondes. Vous pouvez désactiver le délai des transactions en spécifiant la valeur 0 ; cela peut être utile lors du débogage.

Pour plus d'informations sur le débogage distant, voir la rubrique Débogage distant dans l'aide en ligne

Lors du test de l'objet transactionnel, vous pouvez commencer par le tester hors de l'environnement MTS, afin que votre environnement de test soit plus simple.

Lorsque vous développez un serveur, vous ne pouvez pas le reconstruire lorsqu'il est encore en mémoire. Il se peut que vous obteniez une erreur du compilateur, du style "Impossible d'écrire dans une DLL alors que l'exécutable est chargé". Pour l'éviter, vous pouvez définir les propriétés du paquet MTS ou de l'application COM+ pour arrêter le serveur lorsqu'il est inactif.

Pour arrêter le serveur lorsqu'il est inactif :

1. Dans l'explorateur MTS ou le gestionnaire de composants COM+, cliquez avec le bouton droit de la souris sur le paquet MTS ou l'application COM+ dans lequel est installé votre objet transactionnel et choisissez Propriétés.

2. Sélectionnez l'onglet Avancé.

   Cet onglet détermine si le processus serveur associé au paquet s'exécute toujours ou s'il s'arrête après un certain laps de temps.

3. Changez le délai en 0, ce qui arrête le serveur dès qu'il n'y a plus de client à servir.

4. Choisissez OK pour enregistrer la configuration.

Remarque Lors des tests hors de l'environnement MTS, vous ne devez pas faire de référence directe à l'*ObjectProperty* de *TMtsObject*. *TMtsObject* implémente des méthodes, comme *SetComplete* et *SetAbort*, dont l'appel est sécurisé lorsque le contexte de l'objet est **nil**.

Installation d'objets transactionnels

Les applications MTS consistent en un groupe d'objets MTS en processus s'exécutant dans une instance unique de l'exécutable MTS (EXE). Un groupe d'objets COM s'exécutant tous dans le même processus est appelé un **paquet**. Sur une machine unique peuvent s'exécuter plusieurs paquets différents, chaque paquet s'exécutant dans un EXE MTS séparé.

Dans COM+, vous manipulez le même groupe, appelé une **application COM+** Dans une application COM+, les objets n'ont pas besoin d'être en processus et il n'y a pas d'environnement d'exécution séparé.

Vous pouvez grouper les composants de votre application dans un seul paquet MTS ou une application COM+ afin qu'ils s'exécutent dans un seul processus. Vous pouvez aussi distribuer vos composants dans des paquets MTS ou des applications COM+ différentes pour partager votre application entre plusieurs processus ou plusieurs machines.

Pour installer des objets transactionnels dans un paquet MTS ou une application COM+ :

1. Si votre système gère COM+, choisissez Exécuter | Installer les objets COM+. Si votre système ne gère pas COM+ mais que MTS est installé, choisissez Exécuter | Installer les objets MTS. Si votre système ne gère ni MTS ni COM+, vous ne verrez pas d'option de menu proposant d'installer les objets transactionnels.

2. Dans la boîte de dialogue Installer l'objet, cochez les objets à installer.

3. Si vous installez des objets MTS, choisissez le bouton Paquet pour obtenir une liste des paquets MTS dans votre système. Si vous installez des objets COM+, choisissez le bouton Application. Indiquez le paquet MTS ou l'application COM+ dans lequel vous installez vos objets. Vous pouvez choisir Dans nouveau paquet ou Dans nouvelle application pour créer un nouveau paquet MTS ou une nouvelle application COM+ dans lequel installer les objets. Choisissez Dans un paquet existant ou Dans une application existante pour installer les objets dans un paquet MTS ou une application COM+ qui existe déjà.

4 Choisissez OK pour rafraîchir le catalogue, ce qui rend les objets accessibles lors de l'exécution.

Les paquets MTS peuvent contenir des composants issus de plusieurs DLL, et des composants issus de la même DLL peuvent être installés dans différents paquets. Cependant, un même composant ne peut être distribué entre plusieurs paquets.

De même, des applications COM+ peuvent contenir des composants issus de plusieurs exécutables et différents composants d'un même exécutable peuvent être installés dans différentes applications COM+.

Remarque Vous pouvez également installer l'objet transactionnel en utilisant le gestionnaire de composants COM+ ou l'explorateur MTS. Faites bien attention dans ce cas à utiliser les mêmes paramètres pour l'objet que ceux apparaissant dans la page COM+ de l'éditeur de bibliothèques de types. Ces paramètres ne sont pas appliqués automatiquement si vous n'installez pas l'objet depuis l'EDI.

Administration d'objets transactionnels

Une fois des objets transactionnels installés, vous pouvez administrer ces objets d'exécution en utilisant l'explorateur MTS (s'ils sont installés dans un paquet MTS) ou le gestionnaire de composants COM+ (s'ils sont installés dans une application COM+). Ces deux outils sont identiques, si ce n'est que l'explorateur MTS agit sur l'environnement d'exécution MTS alors que le gestionnaire de composants COM+ agit sur les objets COM+.

Le gestionnaire de composants COM+ et l'explorateur MTS permettent de gérer et de déployer les objets transactionnels en utilisant une interface graphique. En utilisant ces outils, vous pouvez :

- Configurer les objets transactionnels, les paquets MTS ou les applications COM+ et les rôles.
- Visualiser les propriétés des composants d'un paquet ou d'une application COM+ et visualiser les paquets MTS ou les applications COM+ installés sur un ordinateur.
- Surveiller et gérer les transactions pour les objets placés dans des transactions.
- Déplacer des paquets MTS ou des applications COM+ d'une machine à une autre.
- Permettre à un client local d'utiliser un objet transactionnel distant.

Pour davantage d'informations sur ces outils, reportez-vous au *Guide de l'administrateur* Microsoft approprié.

Partie

V

Création de composants personnalisés

Les chapitres de cette partie présentent les concepts nécessaires à la conception et à l'implémentation de composants personnalisés dans Delphi.

Chapitre

40

Présentation générale de la création d'un composant

Ce chapitreest une présentation générale de la conception des composants et du processus d'écriture des composants pour les applications Delphi. Vous devez toutefois être familier de Delphi et de ses composants standard.

- VCL et CLX
- Composants et classes
- Comment créer un composant ?
- Contenu d'un composant ?
- Création d'un nouveau composant
- Test des composants non installés

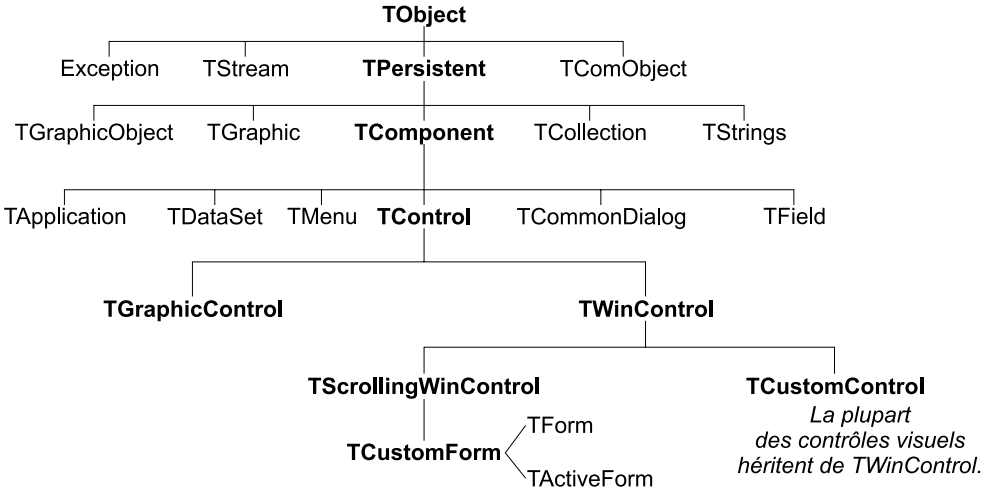
Pour des informations sur l'installation de nouveaux composants, voir "Installation de paquets de composants" à la page 11-6.

VCL et CLX

Les composants de Delphi résident dans deux hiérarchies de classes appelées la bibliothèque des composants visuels (Visual Component Library, VCL) et la bibliothèque des composants multiplates-formes (Component Library for Cross Platform, CLX). La figure suivante présente la relation qui existe entre les classes sélectionnées qui composent la VCL. La hiérarchie CLX est similaire à celle de la VCL, mais les contrôles Windows sont appelés des widgets (par exemple, *TWinControl* est appelé *TWidgetControl*), et il existe d'autres différences. Pour plus de détails sur les hiérarchies de classes et les relations d'héritage entre classes, voir chapitre 41, "Programmation orientée objet et écriture des composants". Pour obtenir un aperçu des différences entre la VCL et la CLX, voir "CLX et VCL" à la page 10-6 et reportez-vous à la référence en ligne de la CLX pour plus de détails sur les composants.

La classe *TComponent* est l'ancêtre partagé de chaque composant de la VCL et la CLX. *TComponent* fournit les propriétés et les événements de base nécessaires au fonctionnement d'un composant dans Delphi. Les différentes branches de la bibliothèque offrent d'autres possibilités plus spécialisées.

Figure 40.1 Hiérarchie des classes de la bibliothèque de composants visuels



Lorsque vous créez un composant, vous l'ajoutez à la VCL ou à la CLX en dérivant une nouvelle classe de l'un des types de classes existant dans la hiérarchie.

Composants et classes

Comme les composants sont des classes, les créateurs de composants manipulent les objets à un niveau différent de celui des développeurs d'applications. La création de nouveaux composants nécessite de dériver de nouvelles classes.

Brièvement, il existe deux différences principales entre la création des composants et leur utilisation dans des applications. Pour la création de composants,

- Vous avez accès à des parties de la classe qui sont inaccessibles aux programmeurs d'applications.
- Vous ajoutez de nouvelles parties (des propriétés, par exemple) aux composants.

A cause de ces différences, il faut prendre en compte un plus grand nombre de conventions, et réfléchir à la manière dont les développeurs d'applications vont utiliser les composants.

Comment créer un composant ?

N'importe quel élément de programme manipulé lors de la phase de conception peut constituer un composant. La création d'un composant consiste à dériver une nouvelle classe à partir d'une classe existante. Vous pouvez dériver un nouveau composant de n'importe quel composant existant, mais les méthodes les plus courantes pour créer des composants sont les suivantes :

- Modification de contrôles existants
- Création de contrôles fenêtrés
- Création de contrôles graphiques
- Sous-classement de contrôles Windows
- Création de composants non visuels

Le tableau suivant présente les différents types de composants et les classes que vous utiliserez comme point de départ pour chacun d'eux.

Tableau 40.1 Points de départ de la création de composants

Pour	Démarrez avec le type suivant
Modifier un composant existant	N'importe quel composant existant tel que *TButton* ou *TListBox*, ou un type de composant abstrait tel que *TCustomListBox*
Créer un contrôle fenêtré (ou widget CLX)	*TWinControl (TWidgetControl* dans CLX*)*
Créer un contrôle graphique	*TGraphicControl*
Sous-classer un contrôle	Tout contrôle Windows (VCL) ou widget (CLX)
Créer un composant non visuel	*TComponent*

Vous pouvez aussi dériver des classes qui ne sont pas des composants et ne peuvent pas être manipulées dans une fiche. Delphi inclut de nombreuses classes de ce type, telles que *TRegIniFile* et *TFont*.

Modification de contrôles existants

Le moyen le plus simple de créer un composant est de modifier un composant existant. Vous pouvez dériver un nouveau composant depuis un composant quelconque de Delphi.

Certains contrôles, tels les boîtes liste et les grilles, possèdent plusieurs variantes d'un thème de base. Dans ce cas, la VCL et la CLX comprennent un type de classe abstraite (son nom contient le mot "custom", comme *TCustomGrid*) à partir de laquelle il est possible de dériver les versions personnalisées.

Vous pouvez, par exemple, créer un type particulier de boîte liste ne possédant pas certaines propriétés de la classe *TListBox*. Comme il n'est pas possible de retirer (masquer) une propriété héritée d'une classe ancêtre, il faut dériver le composant d'un élément situé avant *TListBox* dans la hiérarchie. Au lieu de vous forcer à commencer depuis la classe abstraite *TWinControl* (ou *TWidgetControl* dans la CLX) et à réinventer toutes les fonctions de boîte liste, la VCL ou la CLX fournit

TCustomListBox, qui implémente toutes les propriétés des boîtes liste mais ne les rend pas toutes publiques. En dérivant un composant à partir de l'une des classes abstraites telles que *TCustomListBox*, vous rendez publiques uniquement les propriétés que vous souhaitez mettre à disposition dans votre composant et vous laissez les autres protégées.

Le chapitre 42, "Création de propriétés", explique la publication des propriétés héritées. Le chapitre 48, "Modification d'un composant existant", et le chapitre 50, "Personnalisation d'une grille", montrent des exemples de modification de contrôles existants.

Création de contrôles fenêtrés

Les contrôles fenêtrés de la VCL et de la CLX sont des objets qui apparaissent à l'exécution et avec lesquels l'utilisateur peut interagir. Chaque contrôle fenêtré possède un handle de fenêtre, accessible via sa propriété *Handle*, qui permet au système d'exploitation de l'identifier et d'agir sur lui. Dans le cas d'utilisation de contrôles VCL, le handle permet au contrôle de recevoir la focalisation de saisie et peut être transmis aux fonctions de l'API Windows. Dans la CLX, ces contrôles sont des contrôles basés sur des widgets. Chaque contrôle widget possède un handle, accessible via sa propriété *Handle*, identifiant le widget sous-jacent.

Tous les contrôles fenêtrés descendent de la classe *TWinControl (TWidgetControl dans la CLX)*. Ils incluent la plupart des contrôles fenêtrés standard, tels les boutons poussoirs, les boîtes liste et les boîtes de saisie. Bien que vous puissiez dériver un contrôle original (un qui n'est relié à aucun contrôle existant) directement de *TWinControl (TWidgetControl dans la CLX)*, Delphi fournit pour cela le composant *TCustomControl*. *TCustomControl* est un contrôle fenêtré spécialisé qui permet de réaliser facilement des images visuelles complexes.

Le chapitre 50, "Personnalisation d'une grille", présente un exemple de création d'un contrôle fenêtré.

Création de contrôles graphiques

Si votre contrôle n'a pas besoin de recevoir la focalisation de saisie, vous pouvez en faire un contrôle graphique. Les contrôles graphiques sont semblables aux contrôles fenêtrés, mais ils ne possèdent pas de handle de fenêtre et consomment donc moins de ressources système. Les composants comme *TLabel*, qui ne reçoivent jamais la focalisation de saisie, sont des contrôles graphiques. Bien que ces contrôles ne puissent pas recevoir la focalisation, vous pouvez les créer afin qu'ils réagissent aux messages souris.

Delphi supporte la création de contrôles personnalisés par l'intermédiaire du composant *TGraphicControl*. *TGraphicControl* est une classe abstraite dérivée de *TControl*. Bien qu'il soit possible de dériver des contrôles directement de *TControl*, il est préférable de les dériver de *TGraphicControl*, qui procure un canevas de dessin et sur Windows gère les messages *WM_PAINT* ; il vous suffit de surcharger la méthode *Paint*.

Le chapitre 49, "Création d'un composant graphique", montre un exemple de création d'un contrôle graphique.

Sous-classement de contrôles Windows

En programmation Windows traditionnelle, vous créez des contrôles personnalisés en définissant une nouvelle *classe fenêtre* et en l'enregistrant dans Windows. La classe fenêtre (semblable aux *objets* ou aux *classes* dans la programmation orientée objet). Vous pouvez baser une nouvelle classe fenêtre sur une classe existante : cette opération est appelée *sous-classement*. Vous pouvez ensuite placer votre contrôle dans une bibliothèque dynamiquement liée (DLL), comme les contrôles Windows standard, puis lui fournir une interface.

Avec Delphi vous pouvez créer une "enveloppe" de composant autour de n'importe quelle classe fenêtre existante. Ainsi, si vous possédez déjà une bibliothèque de contrôles personnalisés que vous souhaitez utiliser dans vos applications Delphi, vous pouvez créer des composants Delphi se comportant comme ces contrôles et dériver de nouveaux contrôles à partir d'eux, comme vous le feriez avec n'importe quel composant.

Pour consulter des exemples de sous-classement des contrôles Windows, reportez-vous aux composants de l'unité StdCtls qui représentent les contrôles Windows standard, comme *TEdit*. Pour consulter des exemples CLX, voir QStdCtls.

Création de composants non visuels

Les composants non visuels sont utilisés en tant qu'interfaces pour des éléments comme les bases de données (*TDataSet ou TSQLConnection*) et les horloges système (*TTimer*), et en tant que marques de réservation pour des boîtes de dialogue (*TCommonDialog (VCL)* ou *TDialog* (CLX) et ses descendants). La plupart des composants que vous écrivez sont des contrôles visuels. les composants non visuels peuvent être dérivés directement de *TComponent*, la classe abstraite de base de tous les composants.

Contenu d'un composant ?

Pour que vos composants s'intègrent de manière sûre à l'environnement de Delphi, vous devez suivre certaines conventions. Cette section traite des sujets suivants :

- Suppression des dépendances
- Propriétés, méthodes et événements
- Encapsulation des graphiques
- Recensement

Suppression des dépendances

Une des qualités qui favorisent l'utilisation des composants est le caractère illimité des opérations que l'on peut programmer dans leur code. Par nature, les composants peuvent être incorporés aux applications avec diverses combinaisons, dans des ordres ou des contextes différents. Les composants doivent être conçus pour pouvoir fonctionner dans n'importe quelle situation, sans condition préalable.

La propriété *Handle* des composants *TWinControl* constitue un excellent exemple de suppression des dépendances dans les composants. Si vous avez déjà écrit des applications Windows, vous savez que l'un des points les plus complexes à traiter et les plus susceptibles de générer des erreurs lors de l'exécution d'un programme est l'interdiction d'accéder à une fenêtre ou à un contrôle avant de l'avoir créé par un appel à la fonction API *CreateWindow*. Les contrôles fenêtrés de Delphi évitent cette difficulté en garantissant qu'un handle de fenêtre correct sera toujours disponible dès que nécessaire. En utilisant une propriété pour représenter le handle de fenêtre, le contrôle peut vérifier si la fenêtre a été créée ; si le handle n'est pas correct, le contrôle crée une fenêtre et renvoie son handle. Ainsi, chaque fois que le code d'une application accède à la propriété *Handle*, il est sûr d'obtenir un handle correct.

En les libérant des tâches d'arrière-plan telles que la création des fenêtres, les composants Delphi permettent aux développeurs de se concentrer sur ce qu'ils veulent vraiment réaliser. Pour transmettre un handle de fenêtre à une fonction API, il n'est pas nécessaire de vérifier que le handle existe ni de créer la fenêtre. Le programmeur est certain que les opérations vont se dérouler correctement et n'a pas besoin de le contrôler sans cesse.

Bien que la création de composants sans dépendances soit un peu plus longue, le temps qui y est consacré est généralement très utile. Non seulement cela évite aux développeurs répétitions et travail fastidieux, mais cela réduit la quantité de documentation et de support.

Propriétés, méthodes et événements

En dehors de l'image visible que l'utilisateur du composant manipule dans le concepteur de fiches, les attributs les plus courants d'un composant sont les propriétés, les événements et les méthodes. Un chapitre est consacré à chacun d'eux, mais ce qui suit présente certaines raisons de les utiliser.

Propriétés

Les propriétés donnent au développeur d'applications l'illusion de définir ou de lire la valeur d'une variable, tout en permettant au concepteur de composants de dissimuler la structure de données sous-jacente ou de définir un traitement spécial lorsque la valeur est accédée.

L'utilisation des propriétés présente plusieurs avantages :

- Les propriétés sont disponibles au moment de la conception. Le développeur d'applications peut définir ou modifier les valeurs initiales des propriétés sans écrire de code.
- Les propriétés peuvent contrôler les valeurs ou les formats au moment où le développeur les définit. La validation de la saisie pendant la conception empêche de commettre des erreurs.
- Le composant peut construire les valeurs appropriées à la demande. L'erreur de programmation la plus fréquente est de référencer une variable qui n'a été initialisée. En représentant les données par une propriété, vous êtes sûr qu'une valeur leur est toujours disponible sur demande.
- Les propriétés vous permettent de cacher les données sous une interface simple et cohérente. Vous pouvez modifier la façon dont les informations sont structurées dans une propriété sans que ce changement ne soit perçu par les développeurs d'applications.

Le chapitre 42, "Création de propriétés", explique comment ajouter des propriétés à vos composants.

Evénements

Un événement est une propriété spéciale qui appelle du code, pendant l'exécution, en réponse à une saisie ou à une autre opération. Les événements permettent aux développeurs d'associer des blocs de code spécifiques à des actions spécifiques, telles des manipulations de la souris ou des frappes au clavier. Le code qui s'exécute lorsqu'un événement survient est appelé le *gestionnaire de l'événement*.

Les événements permettent aux développeurs d'applications de spécifier des réponses différentes en fonction des actions possibles sans avoir à créer de nouveaux composants.

Le chapitre 43, "Création d'événements", explique comment implémenter des événements standard et comment en définir de nouveaux.

Méthodes

Les méthodes de classes sont des fonctions et procédures qui opèrent sur une classe plutôt que sur des instances particulières de cette classe. Par exemple, les méthodes constructeur de composants (*Create*) sont toutes des méthodes de classes. Les méthodes de composants sont des procédures et fonctions qui opèrent sur les instances des composants elles-mêmes. Les développeurs d'applications utilisent des méthodes pour que les composants effectuent des actions particulières ou renvoient des valeurs non contenues par des propriétés.

Comme elles nécessitent une exécution de code, les méthodes ne sont disponibles qu'au moment de l'exécution. Elles sont utiles pour plusieurs raisons :

- Les méthodes encapsulent la fonctionnalité d'un composant dans l'objet même où résident les données.

- Les méthodes peuvent cacher des procédures compliquées sous une interface simple et cohérente. Un développeur d'applications peut appeler la méthode *AlignControls* d'un composant sans savoir comment elle fonctionne ni si elle diffère de la méthode *AlignControls* d'un autre composant.
- Les méthodes permettent de mettre à jour plusieurs propriétés avec un seul appel.

Le chapitre 44, "Création de méthodes", explique comment ajouter des méthodes à vos composants.

Encapsulation des graphiques

Delphi simplifie les graphiques Windows en encapsulant les différents outils graphiques dans un canevas. Le canevas représente la surface de dessin d'une fenêtre ou d'un contrôle ; il contient d'autres classes telles qu'un crayon, un pinceau et une police de caractères. Un canevas est semblable à un contexte de périphérique Windows, mais il prend à sa charge toutes les opérations de gestion.

Si vous avez déjà écrit une application Windows graphique, vous connaissez les contraintes imposées par l'interface graphique Windows (GDI), comme les limites sur le nombre de contextes de périphériques disponibles et l'obligation de restaurer l'état initial des objets graphiques avant de les détruire.

Avec Delphi, vous n'avez pas besoin de vous en préoccuper. Pour dessiner sur une fiche ou tout autre composant, il suffit d'accéder à la propriété *Canvas* du composant. Pour personnaliser un crayon ou un pinceau, il faut définir une couleur ou un style. Lorsque vous avez terminé, Delphi dispose des ressources. Delphi conserve les ressources en mémoire cache pour éviter de les recréer, si votre application utilise fréquemment le même type de ressources.

Vous pouvez toujours accéder à l'interface GDI Windows, mais votre code sera beaucoup plus simple et s'exécutera plus rapidement si vous utilisez le canevas intégré aux composants Delphi. Les fonctionnalités graphiques sont décrites en détail au chapitre 45, "Graphiques et composants".

L'encapsulation des graphiques CLX fonctionne d'une manière différente. Un canevas est plutôt un dispositif de dessin. Pour dessiner sur une fiche ou un autre composant, accédez à la propriété *Canvas* du composant. *Canvas* est une propriété et c'est aussi un objet appelé *TCanvas*. *TCanvas* englobe un dispositif de dessin Qt accessible via la propriété *Handle*. Vous pouvez utiliser le handle pour accéder aux fonctions de la bibliothèque graphique Qt de bas niveau.

Si vous voulez personnaliser un crayon ou un pinceau, définissez sa couleur et son style. Lorsque vous avez terminé, Kylix libère les ressources. La CLX met aussi en mémoire cache les ressources.

Vous pouvez utiliser le canevas construit dans les composants CLX par dérivation. La façon dont les images graphiques fonctionnent dans le composant dépend du canevas de l'objet à partir duquel votre composant est dérivé.

Recensement

Avant de pouvoir installer vos composants dans l'EDI de Delphi, vous devez les recenser. Le recensement indique à Delphi où placer le composant sur la palette des composants. Vous pouvez aussi personnaliser la manière dont Delphi stocke les composants dans le fichier fiche. Le recensement est décrit dans le chapitre 47, "Accessibilité des composants au moment de la conception".

Création d'un nouveau composant

Vous pouvez créer un nouveau composant de deux façons :

- Utilisation de l'expert composant
- Création manuelle d'un composant

Vous pouvez utiliser l'une ou l'autre de ces méthodes pour créer un composant aux fonctions minimales, prêt à être installé dans la palette des composants. Après l'installation, vous pouvez placer votre nouveau composant sur une fiche et le tester à la fois en mode conception et en mode exécution. Vous pouvez ensuite ajouter d'autres fonctionnalités au composant, mettre à jour la palette des composants et poursuivre les tests.

Il y a quelques étapes de base à suivre chaque fois que vous créez un nouveau composant. Elles sont décrites ci-après ; pour les autres exemples présentés dans ce document, nous supposerons que vous savez effectuer ces étapes.

1. Création d'une unité pour le nouveau composant.
2. Dérivation du composant à partir d'un type de composant existant.
3. Ajout de propriétés, méthodes et événements.
4. Recensement de votre composant dans Delphi.
5. Création d'un fichier d'aide pour le composant et ses propriétés, méthodes et événements.
6. Création d'un paquet (bibliothèque dynamiquement liée spéciale) pour pouvoir installer le composant dans l'EDI de Delphi.

Lorsque vous avez terminé, le composant complet est constitué des fichiers suivants :

- Un fichier paquet (.BPL) ou un fichier collection de paquets (.DPC)
- Un fichier paquet compilé (.DCP)
- Un fichier unité compilée (.DCU)
- Un fichier bitmap pour la palette (.DCR)
- Un fichier d'aide (.HLP)

La création d'un fichier d'aide à l'attention des utilisateurs de composants est facultative.

Les chapitres du reste de la partie V expliquent tous les aspects de la construction des composants et offrent des exemples complets d'écriture de plusieurs sortes de composants.

Utilisation de l'expert composant

L'expert composant simplifie les premières étapes de création d'un composant. Lorsque vous utilisez l'expert composant, il suffit de spécifier :

- La classe à partir de laquelle le composant est dérivé
- Le nom de classe du nouveau composant
- La page de la palette des composants où vous voulez qu'il apparaisse
- Le nom de l'unité dans laquelle le composant est créé
- Le chemin d'accès à cette unité
- Le nom du paquet dans lequel vous voulez placer le composant

L'expert composant exécute les opérations que vous exécuteriez pour créer manuellement un composant :

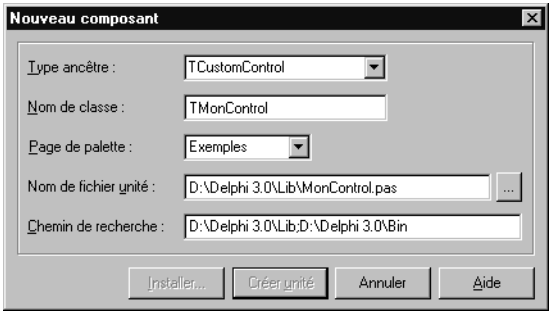
- Création d'une unité
- Dérivation du composant.
- Recensement du composant.

L'expert composant ne peut pas ajouter de composant à une unité existante. Cela ne peut se faire que manuellement.

Pour ouvrir l'expert composant, choisissez l'une de ces deux méthodes :

- Choisissez Composant | Nouveau composant.
- Choisissez Fichier | Nouveau | Autre et double-cliquez sur Composant

Figure 40.2 L'expert composant



Remplissez les champs de l'expert composant :

1 Dans Type ancêtre, spécifiez la classe à partir de laquelle vous dérivez le nouveau composant.

Remarque Dans la liste déroulante, de nombreux composants sont présentés deux fois avec des noms d'unité différents, un pour la VCL et l'autre pour la CLX. Les unités spécifiques à la CLX commencent par Q (telles que QGraphics au lieu de Graphics). Veillez à dériver à partir du bon composant.

2. Dans Nom de classe, spécifiez le nom de classe de votre nouveau composant.
3. Dans Page de palette, spécifiez la page de la palette dans laquelle vous voulez installer le composant.
4. Dans Nom de fichier unité, spécifiez le nom de l'unité dans laquelle vous voulez déclarer la classe du composant.
5. Si l'unité n'est pas dans le chemin de recherche, modifiez ce dernier.

Pour placer un composant dans un paquet nouveau ou non, cliquez sur Composant | Installer et spécifiez le paquet dans la boîte de dialogue qui apparaît.

Attention Si vous dérivez un composant d'une classe de la VCL ou de la CLX dont le nom commence par "custom" (comme *TCustomControl*), ne tentez pas de le placer sur une fiche avant d'avoir surchargé toute méthode abstraite du composant initial. Delphi ne peut pas créer d'instance d'une classe ayant des propriétés ou des méthodes abstraites.

Pour voir le code source de votre unité, cliquez sur Voir l'unité (si l'expert composant est déjà fermé, ouvrez le fichier unité dans l'éditeur de code en sélectionnant Fichier | Ouvrir). Delphi crée une nouvelle unité contenant la déclaration de classe et la procédure *Register*, et ajoute une clause **uses** qui comprend toutes les unités Delphi standard.

L'unité ressemble à cela dans le cas d'une dérivation à partir de *TCustomControl* dans l'unité Controls :

```
unit MyControl;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls;

type
  TMyControl = class(TCustomControl)
  private
  { déclarations privées }
  protected
  { déclarations protégées }
  public
  { déclarations publiques }
  published
  { déclarations publiées }
end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TMyControl]);
end;

end.
```

Dans le cas d'une dérivation à partir de *TCustomControl* dans l'unité QControls, la seule différence est la clause **uses** qui ressemble à ceci :

```
uses
  Windows, Messages, SysUtils, Classes, QControls;
```

Aux endroits où la CLX utilise des unités distinctes, celles-ci sont remplacées par des unités de même nom préfixées par un Q ; Controls est remplacé par QControls.

Création manuelle d'un composant

Le moyen le plus simple de créer un composant est d'utiliser l'expert composant. Cependant, vous pouvez effectuer manuellement les mêmes étapes.

Pour créer un composant manuellement, suivez ces étapes :

1. Création d'un fichier unité
2. Dérivation du composant
3. Recensement du composant

Création d'un fichier unité

Une unité est un module de code Pascal Objet compilé séparèment. Delphi emploie les unités pour plusieurs raisons. Chaque fiche possède sa propre unité et la plupart des composants (ou des groupes logiques de composants) possèdent aussi leurs propres unités

Lorsque vous créez un composant, vous créez une nouvelle unité pour ce composant ou bien vous l'ajoutez à une unité existante.

Pour créer une unité, choisissez Fichier | Nouveau et double-cliquez sur Unité. Delphi crée un nouveau fichier unité et l'ouvre dans l'éditeur de code.

Pour ouvrir une unité existante, choisissez Fichier | Ouvrir et sélectionnez l'unité de code source dans laquelle vous voulez ajouter vos composants.

Remarque Lorsque vous ajoutez un composant à une unité, vérifiez que cette unité ne contient que du code de composant. L'ajout d'un code de composant à une unité qui contient, par exemple, une fiche, provoquera des erreurs dans la palette des composants.

Une fois l'unité, nouvelle ou existante, définie pour le composant, vous pouvez dériver la classe composant.

Dérivation du composant

Chaque composant est une classe dérivée de *TComponent*, de l'un de ses descendants plus spécialisés (tels que *TControl* ou *TGraphicControl*) ou d'une classe composant existante. "Comment créer un composant ?" à la page 40-3 indique les classes à dériver pour obtenir les différentes sortes de composants.

La dérivation des classes est expliquée plus en détail dans la section "Définition de nouvelles classes" à la page 41-1.

Pour dériver un composant, ajoutez une déclaration de type objet à la partie **interface** de l'unité qui contiendra le composant.

Une classe composant simple est un composant non visuel descendant directement de *TComponent*.

Pour créer une classe composant simple, ajoutez la déclaration de classe suivante à la partie **interface** de votre unité composant :

```
type
  TNewComponent = class(TComponent)
  end;
```

Pour l'instant, le nouveau composant ne fait rien de plus que *TComponent*. C'est juste un squelette sur lequel vous allez bâtir votre nouveau composant.

Recensement du composant

Le recensement est une opération simple qui indique à Delphi les composants à ajouter à la bibliothèque des composants et les pages de la palette sur lesquelles ils doivent apparaître. Pour une présentation plus détaillée du processus de recensement, voir chapitre 47, "Accessibilité des composants au moment de la conception".

Pour recenser un composant,

1 Ajoutez une procédure nommée *Register* à la partie **interface** de l'unité du composant. *Register* n'a pas de paramètres, la déclaration est donc très simple :

```
procedure Register;
```

Si vous ajoutez un composant à une unité qui contient déjà des composants, elle doit déjà avoir la procédure *Register* déclarée, afin que vous n'ayiez pas à changer la déclaration.

2 Ecrivez la procédure*Register* dans la partie **implémentation** de l'unité, en appelant *RegisterComponents* pour chaque composant que vous voulez recenser. *RegisterComponents* est une procédure qui prend deux paramètres : le nom d'une page de palette de composants et un ensemble de types de composants. Si vous ajoutez un composant à un recensement existant, vous pouvez soit ajouter le nouveau composant à l'ensemble dans l'instruction existante, soit ajouter une nouvelle instruction qui appelle *RegisterComponents*.

Pour recenser un composant appelé *TMyControl* et le placer sur la page Exemples de la palette, vous devrez ajouter la procédure *Register* suivante à l'unité contenant la déclaration de *TMyControl* :

```
procedure Register;
begin
  RegisterComponents('Samples', [TNewControl]);
end;
```

Cette procédure *Register* place *TMyControl* sur la page Exemples de la palette des composants.

Une fois le composant recensé, vous pouvez le compiler dans un paquet (voir chapitre 47, "Accessibilité des composants au moment de la conception") et l'installer sur la palette des composants.

Test des composants non installés

Vous pouvez tester le comportement d'un composant à l'exécution avant de l'installer sur la palette des composants. Cette technique est particulièrement utile pour le débogage des composants nouvellement créés, mais vous pouvez l'utiliser pour tester n'importe quel composant, que celui-ci apparaisse ou non sur la palette des composants. Pour plus d'informations sur le test des composants déjà installés, voir "Test des composants installés" à la page 40-15.

Vous pouvez tester un composant non installé en émulant les actions exécutées par Delphi quand le composant est sélectionné dans la palette et placé dans une fiche.

Pour tester un composant non installé,

1. Ajoutez le nom de l'unité du composant à la clause **uses** de l'unité fiche.
2. Ajoutez un champ objet à la fiche pour représenter le composant.

   Il s'agit là d'une des différences principales entre votre façon d'ajouter des composants et celle utilisée par Delphi. Vous ajoutez le champ objet à la partie publique à la fin de la déclaration de type de fiche. Delphi l'ajouterait au-dessus, dans la partie de la déclaration de type qu'il gère.

   Il ne faut jamais ajouter de champs à la partie gérée par Delphi de la déclaration de type de fiche. Les éléments de cette partie correspondent à ceux stockés dans le fichier fiche. L'ajout de noms de composants qui n'existent pas sur la fiche peut rendre invalide le fichier de la fiche.
3. Attachez un gestionnaire à l'événement *OnCreate* de la fiche.
4. Construisez le composant dans le gestionnaire *OnCreate* de la fiche.

   Lors de l'appel au constructeur du composant, vous devez transmettre un paramètre spécifiant le propriétaire du composant (le composant chargé de la destruction du composant au moment opportun). Il faut pratiquement toujours transmettre *Self* comme propriétaire. Dans une méthode, *Self* représente une référence sur l'objet contenant la méthode. Dans ce cas, dans le gestionnaire *OnCreate* de la fiche, *Self* représente la fiche.
5. Initialisez la propriété *Parent*.

   La définition de la propriété *Parent* est toujours la première opération à effectuer après la construction d'un contrôle. Le parent est le composant qui contient visuellement le contrôle ; le plus souvent c'est sur la fiche que le contrôle apparaît, mais il peut aussi s'agir d'une boîte groupe ou d'un volet. Normalement, il faut donner à *Parent* la valeur *Self*, c'est-à dire la fiche. Définissez toujours *Parent* avant les autres propriétés du contrôle.

Attention Si votre composant n'est pas un contrôle (c'est-à-dire si *TControl* n'est pas un de ses ancêtres), passez cette étape. Si vous définissez accidentellement la propriété *Parent* de la fiche (à la place de celle du composant) à la valeur *Self*, vous pouvez provoquer un problème du système d'exploitation.

6 Définissez toutes les autres propriétés de composant, comme vous le souhaitez.

Supposons que vous souhaitiez tester un nouveau composant de type *TMyControl* dans une unité appelée *MyControl*. Créez un nouveau projet, puis effectuez les étapes nécessaires pour avoir une unité fiche qui ressemble à ceci :

```
unit Unit1;
interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, MyControl;                        { 1. Ajouter NewTest à la clause uses }

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);       { 3. Attacher un gestionnaire à OnCreate }
  private
    { Private declarations }
  public
    { Public Declarations }
    MyControl1: TMyControl1;                                  { 2. Ajouter un champ objet }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  MyControl1 := TMyControl.Create(Self);                    { 4. Construire le composant }
  MyControl1.Parent := Self;          { 5. Définir la propriété Parent si le composant est
                                              // un contrôle }
  MyControl1.Left := 12;                                  { 6. Définir d'autres propriétés... }
  ⋮                                                                ...continuer si nécessaire}
end;
end.
```

Test des composants installés

Vous pouvez tester le comportement d'un composant à la conception après son installation sur la palette des composants. Cette technique est particulièrement utile pour le débogage des composants nouvellement créés, mais vous pouvez l'utiliser pour tester n'importe quel composant, que celui-ci apparaisse ou non sur la palette des composants.

Pour plus d'informations sur le test des composants qui n'ont pas encore été installés, voir "Test des composants non installés" à la page 40-14.

Le test de vos composants après l'installation vous permet de déboguer le composant qui génère seulement des exceptions à la conception lorqu'il est déposé sur une fiche.

Testez un composant installé en exécutant une seconde instance de Delphi :

1. A partir de l'environnement de développement intégré de Delphi, sélectionnez Projet | Options | et sur la page Répertoires/Conditions, affectez à la zone Sources de débogage le fichier source du composant.
2. Sélectionnez ensuite Outils | Options du débogueur. Sur la page Exceptions du langage, activez les exceptions à suivre.
3. Ouvrez le fichier source du composant et définissez des points d'arrêt.
4. Sélectionnez Exécuter | Paramètres et affectez au champ Application hôte le nom et l'emplacement du fichier exécutable de Delphi.
5. Dans la boîte de dialogue Paramètres d'exécution, cliquez sur le bouton Charger pour démarrer une seconde instance de Delphi.
6. Déposez ensuite les composants à tester sur la fiche, ce qui devrait provoquer l'arrêt sur les points d'arrêt définis dans le source.

Chapitre

41

Programmation orientée objet et écriture des composants

Si vous avez écrit des applications avec Delphi, vous savez déjà qu'une classe contient à la fois du texte et du code, et que les classes sont manipulables aussi bien au moment de la conception qu'à l'exécution. C'est ainsi que vous êtes devenu utilisateur de composants.

Lorsque vous créez de nouveaux composants, votre approche des classes n'est pas celle du développeur d'applications standard. Vous essayez de cacher les travaux internes du composant aux développeurs qui vont les utiliser. En choisissant les ancêtres appropriés à vos composants, en concevant des interfaces qui exposent seulement les propriétés et les méthodes dont les développeurs ont besoin, en suivant les autres conseils de ce chapitre, vous pourrez créer des composants réutilisables parfaitement maniables.

Avant de commencer à créer des composants, vous devez comprendre les sujets suivants qui se rapportent à la programmation orientée objet (OOP) :

- Définition de nouvelles classes
- Ancêtres, descendants et hiérarchies des classes
- Contrôle des accès
- Répartition des méthodes
- Membres abstraits d'une classe
- Classes et pointeurs

Définition de nouvelles classes

La différence entre un concepteur de composants et un développeur d'applications est la suivante : le concepteur de composants crée de nouvelles classes et le développeur d'applications manipule les instances de ces classes.

Une classe est d'abord un type. Comme programmeur, vous travaillez sans arrêt avec les types et les instances, même si vous ne parlez pas en ces termes. Par exemple, vous créez des variables d'un certain type, par exemple *Integer*. Les classes sont habituellement plus complexes que de simples types de données, mais elles fonctionnent de la même façon : en affectant différentes valeurs aux instances d'un même type, vous effectuez différentes tâches.

Par exemple, il est courant de créer une fiche contenant deux boutons appelés OK et Annuler. Chacun correspond à une instance de la classe *TButton*, mais, en attribuant une valeur différente à leurs propriétés *Caption* et différents gestionnaires à leurs événements *OnClick*, vous faites se comporter différemment les deux instances.

Dérivation de nouvelles classes

Deux raisons peuvent vous amener à dériver une nouvelle classe :

- Modifier les valeurs par défaut d'une classe pour éviter les répétitions
- Ajout de nouvelles capacités à une classe

L'objectif est le même dans ces deux cas : créer des objets réutilisables. Si vous concevez vos objets en ayant en tête leur réutilisation, vous gagnerez un temps considérable. Attribuez à vos classes des valeurs par défaut exploitables mais rendez-les personnalisables.

Modifier les valeurs par défaut d'une classe pour éviter les répétitions

Dans tout programme, les répétitions superflues sont à proscrire. Si vous vous surprenez à répéter les mêmes lignes de code, vous serez sans doute amené à les placer à part dans une sous-routine ou fonction, ou encore à construire une bibliothèque de routines utilisables par un autre programme. Le même raisonnement s'applique aux composants. Si vous modifiez fréquemment les mêmes propriétés ou si vous appelez les mêmes méthodes, vous créerez sans doute un nouveau composant qui effectue ces tâches par défaut.

Par exemple, supposons qu'à chaque création d'une nouvelle application, vous ajoutez une boîte de dialogue accomplissant une fonction déterminée. Bien qu'il soit simple de recréer à chaque fois cette boîte de dialogue, c'est superflu. Vous pouvez concevoir la boîte de dialogue une fois pour toute, définir ses propriétés puis installer le composant enveloppe associé dans la palette des composants. En faisant du dialogue un composant réutilisable, non seulement vous éliminez une tâche répétitive mais renforcez la standardisation et minimisez les erreurs qui peuvent être occasionnées par chaque nouvelle création de la boîte de dialogue.

Le chapitre 48, "Modification d'un composant existant", montre un exemple qui modifie les propriétés par défaut d'un composant.

Remarque Si vous voulez ne modifier que les propriétés publiées d'un composant existant ou enregistrer des gestionnaires d'événement spécifiques à un composant ou à un groupe de composants, vous pourrez accomplir ceci plus facilement en créant un *modèle de composant*.

Ajout de nouvelles capacités à une classe

Une raison classique de créer de nouveaux composants est l'ajout de fonctionnalités qui ne se trouvent pas dans les composants existants. Pour cela, vous dérivez le nouveau composant à partir d'un composant existant ou à partir d'une classe de base abstraite, comme *TComponent* ou *TControl*.

Dérivez votre nouveau composant à partir de la classe contenant le sous-ensemble le plus proche des caractéristiques recherchées. Vous pouvez ajouter des fonctionnalités à une classe, mais vous ne pouvez pas en soustraire. Par conséquent, si une classe de composant contient des propriétés que vous ne souhaitez *pas* inclure dans la vôtre, effectuez la dérivation à partir de l'ancêtre de ce composant.

Par exemple, pour ajouter des fonctionnalités à une boîte liste, vous devez dériver un nouveau composant à partir de *TListBox*. Mais, si vous souhaitez ajouter de nouvelles fonctionnalités et exclure certaines de celles des boîtes liste standard, il vous faut dériver votre boîte liste de l'ancêtre de *TListBox*, *TCustomListBox*. Recréez (ou rendez visibles) les fonctionnalités de la boîte liste que vous voulez, puis ajoutez vos propres fonctionnalités.

Le chapitre 50, "Personnalisation d'une grille", montre un exemple qui personnalise une classe abstraite de composant.

Déclaration d'une nouvelle classe de composant

Outre les composants standard, Delphi fournit de nombreuses classes abstraites qui serviront de base pour dériver de nouveaux composants. Le tableau 40.1 à la page 40-3 montre les classes que vous pouvez utiliser pour créer vos propres composants.

Pour déclarer une nouvelle classe de composant, ajoutez une déclaration de classe au fichier unité du composant.

Voici la déclaration d'un composant graphique simple :

```
type
  TSampleShape = class(TGraphicControl)
  end;
```

Une déclaration de composant achevée comprend généralement la déclaration des propriétés, des événements et des méthodes avant le **end**. Mais une déclaration comme celle ci-dessus est aussi admise et représente un point de départ pour l'ajout de fonctionnalités à votre composant.

Ancêtres, descendants et hiérarchies des classes

Les développeurs d'applications peuvent se servir du fait que chaque contrôle a des propriétés *Top* et *Left* qui déterminent son emplacement sur la fiche. Pour eux, il importe peu que tous les contrôles aient hérité ces propriétés d'un ancêtre commun, *TControl*. Mais, lorsque vous écrivez un composant, vous devez savoir

à partir de quelle classe vous le dérivez de façon à ce qu'il reçoive en héritage les éléments que vous souhaitez. Vous devez également connaître toutes les fonctionnalités dont hérite votre composant de manière à les exploiter sans avoir à les recréer vous-même.

La classe à partir de laquelle vous effectuez la dérivation est l'*ancêtre immédiat*. Chaque composant hérite de son ancêtre immédiat, et lui-même de son ancêtre immédiat. Toutes les classes dont hérite un composant sont les ancêtres de ce composant ; le composant est un *descendant* de ces ancêtres.

L'ensemble des relations ancêtre-descendant de l'application constitue les hiérarchies des classes. Dans cette hiérarchie, chaque génération contient plus que ses ancêtres puisqu'une classe hérite de tout ce que contiennent ses ancêtres et ajoute de nouvelles propriétés et de nouvelles méthodes, ou redéfinit celles qui existent.

Si vous ne spécifiez aucun ancêtre immédiat, Delphi dérive votre composant à partir de l'ancêtre par défaut, *TObject*. *TObject* est le dernier ancêtre de toutes les classes de la hirarchie d'objets.

La règle générale du choix de l'objet de départ de la dérivation est simple : prenez l'objet qui contient le plus possible ce que vous voulez inclure dans votre nouvel objet, mais qui ne comprend rien de ce que vous ne voulez pas dans le nouvel objet. Vous pouvez toujours ajouter des choses à vos objets, mais vous ne pouvez pas en retirer.

Contrôle des accès

Il existe cinq niveaux de *contrôle d'accès,* également appelé *visibilité,* aux propriétés, méthodes et champs. La visibilité détermine quel code peut accéder à quelles parties de la classe. En spécifiant la visibilité, vous définissez l'*interface* de vos composants.

montre les niveaux de visibilité, en allant du plus restrictif au plus accessible:

Tableau 41.1 Niveaux de visibilité d'un objet

Visibilité	Signification	Utilisé pour
private	Accessible uniquement au code de l'unité où est définie la classe.	Masquer les détails d'implémentation.
protected	Accessible au code de ou des unités où sont définis la classe et ses descendants.	Définition de l'interface avec le concepteur des composants.
public	Accessible à tout le code.	Définition de l'interface d'exécution.
automated	Accessible à tout le code. Les informations d'automatisation sont générées.	Automatisation OLE seulement.
published	Accessible à tout le code et depuis l'inspecteur d'objets.	Définition de l'interface de conception.

Déclarez les membres en **private** si vous voulez qu'ils ne soient disponibles que dans la classe où ils ont été définis. Déclarez-les en **protected** si vous voulez qu'ils ne soient disponibles que dans cette classe et ses descendants. Souvenez-vous que si un membre est disponible n'importe où dans un fichier unité, il est disponible *partout* dans ce fichier. Ainsi, si vous définissez deux classes dans la même unité, elles pourront accéder à l'une ou l'autre des méthodes privées. Et si vous dérivez une classe dans une unité différente de son ancêtre, toutes les classes de la nouvelle unité pourront accéder aux méthodes protégées de l'ancêtre.

Masquer les détails d'implémentation

Déclarer **private** une partie d'une classe rend cette partie invisible au code situé hors du fichier unité de la classe. Dans l'unité qui contient la déclaration, le code peut accéder à cette partie comme si elle était publique.

Voici un exemple qui montre comment le fait de déclarer une donnée membre **private** empêche les utilisateurs d'accéder aux informations. La liste montre les deux unités de fiche VCL. Chaque fiche a un gestionnaire pour son événement *OnCreate* qui affecte une valeur à la donnée membre private. Le compilateur autorise l'affectation à la donnée membre uniquement dans la fiche où elle a été déclarée.

```
unit HideInfo;
interface

uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs;

type
  TSecretForm = class(TForm)                              { déclare une nouvelle fiche }
    procedure FormCreate(Sender: TObject);
  private                                                   { déclare la partie privée }
    FSecretCode: Integer;                             { déclare une donnée membre private }
  end;

var
  SecretForm: TSecretForm;

implementation
procedure TSecretForm.FormCreate(Sender: TObject);
begin
  FSecretCode := 42;                                      { ceci se compile correctement }
end;
end.                                                                        { fin de l'unité }

unit TestHide;                                       { il s'agit du fichier fiche principal }

interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs,
  HideInfo;                                            { utilise l'unité avec TSecretForm }

type
  TTestForm = class(TForm)
    procedure FormCreate(Sender: TObject);
  end;
var
```

```
  TestForm: TTestForm;

implementation
procedure TTestForm.FormCreate(Sender: TObject);
begin
  SecretForm.FSecretCode := 13;                          { le compilateur s'arrête avec
                                                    // "Identificateur de champ attendu" }
end;
end.                                  { fin de l'unité}
```

Bien qu'un programme utilisant l'unité *HideInfo* puisse utiliser des objets de type *TSecretForm*, il ne peut pas accéder à la donnée membre *FSecretCode* dans aucun de ces objets.

Définition de l'interface avec le concepteur des composants

Déclarer **protected** une partie d'une classe rend cette partie uniquement visible par cette classe et par ses descendants (et par les autres classes qui partagent leurs fichiers unité).

Vous pouvez utiliser les déclarations **protected** pour définir *l'interface de conception des composants* d'une classe. Les unités de l'application ne peuvent pas accéder aux parties **protected**, mais les classes dérivées le peuvent. Cela signifie que les concepteurs des composants peuvent modifier la façon dont fonctionne une classe sans rendre apparents ces détails aux développeurs d'applications.

Remarque Une erreur commune consiste à essayer d'accéder aux méthodes protégées d'un gestionnaire d'événement. Les gestionnaires d'événements sont généralement des méthodes de la fiche, et non du composant qui reçoit l'événement. Par conséquent, ils n'accèdent pas aux méthodes protégées du composant (à moins que le composant ne soit déclaré dans la même unité que la fiche).

Définition de l'interface d'exécution

Déclarer **public** une partie d'une classe rend cette partie visible par tout code qui dispose d'un accès global à cette classe.

Les parties publiques sont disponibles à l'ensemble du code lors de l'exécution, et constituent par là-même l'*interface d'exécution*. L'interface d'exécution sert aux éléments qui sont sans signification ou inappropriés au moment de la conception ; comme les propriétés contenant des informations uniquement disponibles à l'exécution ou accessibles uniquement en lecture. Les méthodes destinées à être appelées par les développeurs d'applications doivent également être publiques.

Voici un exemple montrant deux propriétés accessibles uniquement en lecture et déclarées comme faisant partie de l'interface d'exécution d'un composant :

```
type
  TSampleComponent = class(TComponent)
  private
    FTempCelsius: Integer;                { les détails de l'implémentation sont privés }
```

```
    function GetTempFahrenheit: Integer;
  public
    property TempCelsius: Integer read FTempCelsius;      { les propriétés sont publiques }
    property TempFahrenheit: Integer read GetTempFahrenheit;
  end;
⋮
function TSampleComponent.GetTempFahrenheit: Integer;
begin
  Result := FTempCelsius * 9 div 5 + 32;
end;
```

Définition de l'interface de conception

Déclarer **published** une partie d'une classe rend publique cette partie et génère également les informations de types à l'exécution. Entre autres rôles, les informations de types à l'exécution permettent à l'inspecteur d'objets d'accéder aux propriétés et aux événements.

Parce qu'ils apparaissent dans l'inspecteur d'objets, les parties **published** d'une classe définissent *l'interface de conception* de cette classe. L'interface de conception doit inclure toutes les caractéristiques d'une classe qu'un développeur d'applications peut vouloir personnaliser au moment de la conception, tout en excluant toutes les propriétés qui dépendent d'une information spécifique issue de l'environnement d'exécution.

Les propriétés accessibles en lecture uniquement ne peuvent pas faire partie de l'interface de conception car le développeur d'applications ne peut pas leur affecter des valeurs directement. Les propriétés accessibles en lecture uniquement doivent donc être déclarées **public** plutôt que **published**.

Voici l'exemple d'une propriété published nommée *Temperature*. De ce fait, elle apparaît dans l'inspecteur d'objets au moment de la conception.

```
type
  TSampleComponent = class(TComponent)
  private
    FTemperature: Integer;                    { les détails d'implémentation sont privés }
  published
    property Temperature: Integer read FTemperature write FTemperature; { peut s'écrire ! }
  end;
```

Répartition des méthodes

Le terme de *Dispatch* fait référence à la façon dont un programme détermine où il doit rechercher une méthode de classe lorsqu'il rencontre un appel à cette méthode. Le code qui appelle une méthode de classe ressemble à tout autre appel de fonction. Mais les classes ont des façons différentes de répartir les méthodes.

Les trois types de répartition de méthodes sont

- Statique
- Virtuel
- Dynamique

Méthodes statiques

Toutes les méthodes sont statiques à moins que vous ne les déclariez spécifiquement autrement. Les méthodes statiques fonctionnent comme des procédures ou des fonctions normales. Le compilateur détermine l'adresse exacte de la méthode et la lie au moment de la compilation.

Le premier avantage des méthodes statiques est qu'elles sont réparties très rapidement. Comme le compilateur peut déterminer l'adresse exacte de la méthode, il la lie directement. Les méthodes virtuelles et dynamiques, au contraire, utilisent des moyens indirects pour donner l'adresse de leurs méthodes lors de l'exécution, ce qui prend davantage de temps.

Une méthode statique ne change pas lorsqu'elle est héritée d'une classe descendante. Si vous déclarez une classe comprenant une méthode statique, puis dérivez une nouvelle classe à partir de celle-ci, la classe dérivée partage exactement la même méthode à la même adresse. Cela signifie que vous ne pouvez pas redéfinir des méthodes statiques. Une méthode statique réalise toujours la même chose quelle que soit la classe qui y est appelée. Si vous déclarez une méthode dans une classe dérivée ayant le même nom qu'une méthode statique dans la classe ancêtre, la nouvelle méthode remplace simplement celle héritée dans la classe dérivée.

Exemple de méthodes statiques

Dans le code suivant, le premier composant déclare deux méthodes statiques. Le deuxième déclare deux méthodes statiques de même nom qui remplacent les méthodes héritées du premier composant.

```
type
  TFirstComponent = class(TComponent)
    procedure Move;
    procedure Flash;
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;         { différent de la méthode héritée, malgré la même déclaration }
    function Flash(HowOften: Integer): Integer;                  { c'est aussi différent }
  end;
```

Méthodes virtuelles

Les méthodes virtuelles emploient un mécanisme de répartition plus compliqué et plus souple que les méthodes statiques. Une méthode virtuelle peut être redéfinie dans des classes descendantes, mais est toujours appelée dans la classe ancêtre. L'adresse d'une méthode virtuelle n'est pas déterminée lors de la

compilation ; à la place, l'objet où la méthode est définie donne l'adresse lors de l'exécution.

Pour qu'une méthode soit virtuelle, ajoutez la directive **virtual** après la déclaration de la méthode. La directive **virtual** crée une entrée dans le *tableau de méthode virtuelle*, de l'objet, ou VMT, qui contient les adresses de toutes les méthodes virtuelles d'un type objet.

Lorsque vous dérivez une nouvelle classe d'une classe existante, la nouvelle classe a son propre VMT, qui comprend toutes les entrées provenant du VMT de l'ancêtre plus toutes les méthodes virtuelles supplémentaires déclarées dans la nouvelle classe.

Surcharge des méthodes

Surcharger une méthode signifie l'étendre ou la redéfinir plutôt que la remplacer. Une classe descendante peut surcharger toutes ses méthodes virtuelles héritées.

Pour surcharger une méthode dans une classe descendante, ajoutez la directive **override** à la fin de la déclaration de méthode.

La surcharge d'une méthode provoque une erreur de compilation si

- La méthode n'existe pas dans la classe ancêtre.
- La méthode de l'ancêtre du même nom est statique.
- Les déclarations ne sont pas identiques (le numéro et le type des paramètres arguments diffèrent).

Le code suivant montre la déclaration de deux composants simples. Le premier déclare trois méthodes, chacune avec un type de répartition différent. L'autre, dérivé du premier, remplace la méthode statique et surcharge les méthodes virtuelles.

```
type
  TFirstComponent = class(TCustomControl)
    procedure Move;                          { méthode statique }
    procedure Flash; virtual;               { méthode virtuelle }
    procedure Beep; dynamic;        { méthode virtuelle dynamique }
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;               { déclare une nouvelle méthode }
    procedure Flash; override;     { surcharge la méthode héritée }
    procedure Beep; override;      { surcharge la méthode héritée }
  end;
```

Méthodes dynamiques

Les méthodes dynamiques sont des méthodes virtuelles avec un mécanisme de répartition légèrement différent. Comme les méthodes dynamiques n'ont pas d'entrées dans le tableau de méthode virtuelle de l'objet, elles peuvent réduire la taille de la mémoire consommée par les objets. Cependant les méthodes de répartition dynamiques sont quelque peu plus lentes que les méthodes de répartition virtuelles normales. Si une méthode est fréquemment appelée, ou si

son exécution nécessite un temps court, vous devrez probablement la déclarer virtuelle plutôt que dynamique.

Les objets doivent stocker les adresses de leurs méthodes dynamiques. Mais plutôt que de recevoir les entrées dans le tableau de méthode virtuelle, les méthodes dynamiques sont indiquées séparément. La liste des méthodes dynamiques contient des entrées uniquement pour les méthodes introduites ou surchargées par une classe particulière (le tableau de méthode virtuelle, à l'inverse, comprend toutes les méthodes virtuelles de l'objet, à la fois héritées et introduites). Les méthodes dynamiques héritées sont réparties en cherchant chaque liste de méthode dynamique de l'ancêtre, en allant en arrière dans l'arborescence de l'héritage.

Pour rendre une méthode dynamique, ajoutez la directive **dynamic** après la déclaration de méthode.

Membres abstraits d'une classe

Lorsqu'une méthode est déclarée **abstract** dans une classe ancêtre, vous devez la surfacer (c'est-à-dire la redéclarer et l'implémenter) dans tout composant descendant avant d'utiliser le nouveau composant dans les applications. Delphi ne peut créer d'instance d'une classe contenant des membres abstraits. Pour plus d'informations sur le surfaçage des constituants hérités des classes, voir chapitre 42, "Création de propriétés", et chapitre 44, "Création de méthodes".

Classes et pointeurs

Chaque classe (et par conséquent chaque composant) est en fait un pointeur. Le compilateur déréférence automatiquement les pointeurs de classe à votre place, aussi n'avez-vous généralement pas besoin de vous poser ces questions. Le statut des classes en tant que pointeurs devient important lorsque vous passez une classe comme paramètre. En général, vous transmettrez les classes par valeur plutôt que par référence. Car les classes sont déjà des pointeurs, c'est-à-dire des références ; transmettre une classe par référence serait transmettre une référence à une référence.

Chapitre

42

Création de propriétés

Les propriétés sont la partie la plus visible des composants. Le développeur d'applications a la possibilité de les voir et de les manipuler au moment de la conception et dispose d'un retour immédiat au fur et à mesure que réagissent les composants dans le concepteur de fiches. Les propriétés conçues correctement facilitent l'utilisation de vos composants par d'autres personnes et leur maintenance par vous-même.

Pour utiliser de façon optimale les propriétés de vos composants, vous devez connaître les points suivants :

- Pourquoi créer des propriétés ?
- Types de propriétés
- Publication des propriétés héritées
- Définition des propriétés
- Création de propriétés tableau
- Stockage et chargement des propriétés

Pourquoi créer des propriétés ?

Du point de vue du développeur d'applications, les propriétés ressemblent à des variables. Les développeurs peuvent définir ou lire les valeurs des propriétés comme s'il s'agissait de champs. La seule opération interdite avec une propriété et autorisée avec une variable consiste à la transmettre comme paramètre **var**.

Les propriétés ont une puissance bien supérieure à celle de simples champs car

- Les développeurs d'applications peuvent définir des propriétés au moment de la conception. Contrairement aux méthodes, qui ne sont accessibles qu'à l'exécution, les propriétés permettent au développeur de personnaliser les composants avant l'exécution de l'application. Les propriétés apparaissent dans l'inspecteur d'objets, ce qui simplifie le travail du programmeur ; au lieu de traiter plusieurs paramètres pour construire un objet, il laisse Delphi lire

des valeurs dans l'inspecteur d'objets. L'inspecteur d'objets valide les affectations des valeurs aux propriétés dès qu'elles sont effectuées.

- Les propriétés peuvent masquer les détails de l'implémentation. Par exemple, des données stockées de façon interne sous une forme cryptée peuvent apparaître non cryptées en tant que la valeur d'une propriété ; bien que la valeur puisse être un simple nombre, le composant peut rechercher cette valeur dans une base de données ou effectuer des calculs complexes afin de la récupérer. Les propriétés permettent d'associer des opérations complexes à une simple affectation ; ce qui apparaît comme l'affectation d'un champ correspond en fait à un appel de méthode et cette dernière peut accomplir à peu près n'importe quelle tâche.
- Les propriétés peuvent être virtuelles. Ce qui paraît être une seule propriété pour le développeur d'applications peut être implémenté de manière différente dans des composants différents.

Un exemple simple est la propriété *Top* que tous les contrôles possèdent. L'attribution d'une nouvelle valeur à *Top* n'a pas pour seul effet de modifier une valeur mémorisée ; elle provoque aussi le déplacement et le réaffichage du contrôle. Les effets de la définition d'une propriété ne se limitent pas nécessairement à un composant unique ; par exemple, donner la valeur *True* à la propriété *Down d'un turbobouton* a pour effet d'attribuer la valeur *False* à tous les autres turboboutons du même groupe.

Types de propriétés

Une propriété peut avoir un type quelconque. Les divers types sont affichés de manière différente dans l'inspecteur d'objets, ce qui valide l'affectation des propriétés effectuées au moment de la conception.

Tableau 42.1 Affichage des propriétés dans l'inspecteur d'objets

Type de propriété	Traitement de l'inspecteur d'objets
Simple	Les propriétés de type numérique, caractère et chaîne apparaissent dans l'inspecteur d'objets comme des nombres, caractères et chaînes. Le développeur d'applications peut modifier directement la valeur de ces propriétés.
Enuméré	Les propriétés de type énuméré (y compris le type *Boolean*) apparaissent comme des chaînes éditables. Le développeur peut également passer en revue toutes les valeurs possibles en double-cliquant sur la colonne contenant la valeur et il existe une liste déroulante montrant toutes les valeurs possibles.
Ensemble	Les propriétés de type ensemble apparaissent dans l'inspecteur d'objets comme des ensembles. En double-cliquant sur la propriété, le développeur peut développer l'ensemble et traiter chacun des éléments comme une valeur booléenne (*true* si cet élément appartient à l'ensemble).

Tableau 42.1 Affichage des propriétés dans l'inspecteur d'objets (suite)

Type de propriété	Traitement de l'inspecteur d'objets
Objet	Les propriétés qui sont elles-mêmes des classes ont souvent leur propre éditeur de propriétés, qui est spécifié dans la procédure de recensement du composant. Si la classe d'une propriété a ses propres propriétés publiées (**published**), l'inspecteur d'objets permet au développeur d'étendre la liste (en double-cliquant) afin d'inclure ces propriétés et de les modifier individuellement. Les propriétés doivent descendre de *TPersistent*.
Tableau	Les propriétés tableau doivent disposer d'un éditeur de propriétés spécifique. L'inspecteur d'objets ne dispose d'aucune fonction intégrée permettant de modifier les propriétés de ce type. Vous pouvez spécifier un éditeur de propriétés lorsque vous recensez vos composants.

Publication des propriétés héritées

Tous les composants héritent des propriétés de leurs classes ancêtre. Lorsque vous dérivez un composant à partir d'un composant existant, le nouveau composant hérite de toutes les propriétés de l'ancêtre immédiat. Si vous effectuez la dérivation à partir d'une des classes abstraites, aucune des propriétés héritées n'est **published**, la plupart sont **protected** ou **public**.

Pour rendre disponible dans l'inspecteur d'objets une propriété **protected** ou **private** au moment de la conception, vous devez redéclarer **published** la propriété. Redéclarer une propriété signifie ajouter la déclaration d'une propriété héritée à la déclaration de la classe descendante.

Si vous dérivez un composant VCL de *TWinControl*, par exemple, il hérite de la propriété protégée *DockSite*. En redéclarant *DockSite* dans votre nouveau composant, vous pouvez changer le niveau de protection en public ou publié.

Le code suivant montre une redéclaration de *DockSite* en publié, le rendant disponible lors de la conception.

```
type
  TSampleComponent = class(TWinControl)
  published
    property DockSite;
  end;
```

Lorsque vous redéclarez une propriété, vous spécifiez uniquement le nom de la propriété, non le type ni les autres informations décrites ci-dessous dans "Définition des propriétés". Vous pouvez aussi déclarer de nouvelles valeurs par défaut et spécifier si la propriété est ou non stockée.

Les redéclarations peuvent augmenter la visibilité d'une propriété, mais pas la réduire. Vous pouvez ainsi rendre une propriété protégée publique, mais vous ne pouvez pas masquer une propriété publique en la redéclarant protégée.

Définition des propriétés

Cette section montre comment déclarer de nouvelles propriétés et explique certaines conventions respectées par les composants standard. Ces rubriques comprennent :

- Déclaration des propriétés
- Stockage interne des données
- Accès direct
- Méthodes d'accès
- Valeurs par défaut d'une propriété

Déclaration des propriétés

Une propriété est déclarée dans la déclaration de sa classe composant. Pour déclarer une propriété, vous devez spécifier les trois éléments suivants :

- Le nom de la propriété.
- Le type de la propriété.
- Les méthodes utilisées pour lire et écrire la valeur de la propriété. Si aucune méthode d'écriture n'est déclarée, la propriété est accessible uniquement en lecture.

Les propriétés déclarées dans une section **published** de la déclaration de classe du composant sont modifiables dans l'inspecteur d'objets lors de la conception. La valeur d'une propriété published est enregistrée avec le composant dans le fichier fiche. Les propriétés déclarées dans une **section public** sont accessibles à l'exécution et peuvent être lues ou définies par le code du programme.

Voici une déclaration typique pour une propriété appelée *Count*.

```
type
  TYourComponent = class(TComponent)
  private
    FCount: Integer;                                  { utilisé pour le stockage interne }
    procedure SetCount (Value: Integer);                          { méthode d'écriture }
  public
    property Count: Integer read FCount write SetCount;
  end;
```

Stockage interne des données

Il n'existe aucune restriction quant au stockage des données d'une propriété. Toutefois, les composants Delphi respectent généralement les conventions suivantes :

- Les données des propriétés sont stockées dans des champs.
- Les champs utilisés pour stocker les données d'une propriété sont déclarés **private** et ne peuvent être accédées qu'à partir du composant lui-même. Les composants dérivés doivent utiliser la propriété héritée ; ils ne nécessitent pas un accès direct au stockage interne des données de la propriété.

- Les identificateurs de ces champs sont composés de la lettre *F* suivie du nom de la propriété. Par exemple, la donnée brute de la propriété *Width* définie pour *TControl* est stockée dans un champ appelé *FWidth*.

Le principe qui sous-tend ces conventions est le suivant : seules les méthodes d'implémentation d'une propriété doivent pouvoir accéder aux données associées à cette propriété. Si une méthode ou une autre propriété a besoin de changer ces données, elle doit le faire via la propriété et non directement par un accès aux données stockées. Cela garantit que l'implémentation d'une propriété héritée puisse être modifiée sans invalider les composants dérivés.

Accès direct

L'*accès direct* est le moyen le plus simple d'accéder aux données d'une propriété. Autrement dit, les parties **read** et **write** de la déclaration d'une propriété spécifient que l'affectation ou la lecture de la valeur de la propriété s'effectue directement dans le champ de stockage interne sans appel à une méthode d'accès. L'accès direct est utile lorsque vous voulez rendre une propriété accessible dans l'inspecteur d'objets, mais que vous ne voulez pas que le changement de sa valeur déclenche un processus immédiatement.

En général, vous définirez un accès direct pour la partie **read** d'une déclaration de propriété et utiliserez une méthode d'accès pour la partie **write**. Cela permet de mettre à jour l'état du composant lorsque la valeur de la propriété change.

La déclaration de type composant suivante montre une propriété qui utilise l'accès direct pour les parties **read** et **write**.

```
type
  TSampleComponent = class(TComponent)
  private                                          { le stockage interne est privé }
    FMyProperty: Boolean;              { déclare la donnée membre pour contenir la valeur }
  published                                 { rend la propriété disponible à la conception }
    property MyProperty: Boolean read FMyProperty write FMyProperty;
  end;
```

Méthodes d'accès

Vous pouvez spécifier une méthode d'accès plutôt qu'un champ dans les parties **read** et **write** d'une déclaration de propriété. Les méthodes d'accès doivent être **protected**, et sont habituellement déclarées comme **virtual** ; cela autorise les composants descendants à surcharger l'implémentation de la propriété.

Evitez de rendre publiques les méthodes d'accès. Les conserver **protected** vous prémunit contre toute modification accidentelle d'une propriété par un développeur d'applications qui appellerait ces méthodes.

Voici une classe qui déclare trois propriétés en utilisant le spécificateur d'index, qui autorise aux trois propriétés d'avoir les mêmes méthodes d'accès en lecture et en écriture :

```
type
  TSampleCalendar = class(TCustomGrid)
```

```
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  private
    function GetDateElement(Index: Integer): Integer; { remarquez le paramètre Index }
    procedure SetDateElement(Index: Integer; Value: Integer);
  ⋮
```

Comme chaque élément de la date (day, month et year) est un int et comme la définition de chacun requiert le codage de la date lorsqu'elle est définie, le code évite la duplication en partageant les méthodes de lecture et d'écriture pour les trois propriétés. Vous n'avez besoin que d'une seule méthode pour lire un élément date et une autre pour écrire l'élément date.

Voici la méthode read qui obtient l'élément date :

```
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);          { décompose la date codée en éléments }
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;
```

Voici la méthode write qui définit l'élément date approprié :

```
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
  AYear, AMonth, ADay: Word;
begin
  if Value > 0 then                              { tous les éléments doivent être positifs }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);            { prend les éléments date actuels }
    case Index of                                    { définit le nouvel élément selon Index }
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);                   { code la date modifiée }
    Refresh;                                          { actualise le calendrier visible }
  end;
end;
```

Méthode read

La méthode read d'une propriété est une fonction qui n'accepte aucun paramètre (sauf pour ce qui est mentionné ci-après) et renvoie une valeur du même type que la propriété. Par convention, le nom de la fonction est *Get* suivi du nom de la propriété. Par exemple, la méthode **read** pour une propriété intitulée *Count*

serait *GetCount*. La méthode read manipule les données internes afin de générer une valeur de la propriété respectant le type demandé.

Les seules exceptions à la règle "aucun paramètre" sont les propriétés tableau et les propriétés qui utilisent un spécificateur d'index (voir "Création de propriétés tableau" à la page 42-8), pour lesquelles cet index est transmis comme paramètre. Utilisez des spécificateurs d'index pour créer une méthode read unique partagée par plusieurs propriétés. Pour plus d'informations sur les spécificateurs d'index, consultez le *Guide du langage Pascal Objet*.

Si vous ne déclarez aucune méthode **read**, la propriété fonctionne uniquement en écriture. Les propriétés fonctionnant en écriture uniquement sont très rares.

Méthode write

La méthode write d'une propriété est une procédure acceptant un seul paramètre (sauf pour ce qui est mentionné ci-après) du même type que la propriété. Le paramètre peut être transmis par référence ou par valeur et peut porter le nom de votre choix. Par convention, le nom de la méthode write est *Set* suivi du nom de la propriété. Par exemple, la méthode **write** d'une propriété intitulée *Count* serait *SetCount*. La valeur transmise en paramètre devient la nouvelle valeur de la propriété ; la méthode **write** doit accomplir les manipulations nécessaires pour placer les données concernées à l'emplacement de stockage interne de la propriété.

Les seules exceptions à la règle "paramètre unique" sont les propriétés tableau et les propriétés qui utilisent un spécificateur d'index, pour lesquelles cet index est transmis comme second paramètre. Utilisez des spécificateurs d'index pour créer une méthode read unique partagée par plusieurs propriétés. Pour plus d'informations sur les spécificateurs d'index, consultez le *Guide du langage Pascal Objet*.

Si vous ne déclarez aucune méthode **write**, la propriété fonctionne uniquement en lecture. Les méthodes write testent normalement si une nouvelle valeur diffère de la valeur actuelle avant de modifier la propriété. Par exemple, voici une méthode **write** simple d'une propriété de type entier appelée *Count* stockant sa valeur courante dans un champ appelé *FCount*.

```
procedure TMyComponent.SetCount(Value: Integer);
begin
  if Value <> FCount then
  begin
    FCount := Value;
    Update;
  end;
end;
```

Valeurs par défaut d'une propriété

Lorsque vous déclarez une propriété, vous pouvez déclarer une *valeur par défaut*. Delphi utilise cette valeur par défaut pour déterminer si une propriété doit être stockée dans un fichier fiche. Si vous ne donnez pas de valeur par défaut à une propriété, Delphi stocke *toujours* cette propriété.

Pour spécifier une valeur par défaut pour une propriété, ajoutez la directive **default** à la déclaration (ou à la redéclaration) de la propriété, suivie par la valeur par défaut. Par exemple,

```
property Cool Boolean read GetCool write SetCool default True;
```

Remarque Déclarer une valeur par défaut pour une propriété n'a pas pour effet de définir cette propriété par cette valeur. La méthode constructeur du composant doit initialiser la valeur des propriétés lorsque c'est nécessaire. Toutefois, comme les objets initialisent toujours leurs champs à 0, il n'est pas nécessaire que le constructeur initialise les propriétés entières à 0, les propriétés chaînes à null ni les propriétés booléennes à *False*.

Spécification d'aucune valeur par défaut

Lorsque vous redéclarez une propriété, vous pouvez indiquer que la propriété ne possède pas de valeur par défaut, même si une telle valeur est définie pour la propriété reçue en héritage.

Pour indiquer qu'une propriété n'a pas de valeur par défaut, ajoutez la directive **nodefault** à la déclaration de la propriété. Par exemple,

```
property FavoriteFlavor string nodefault;
```

Lorsque vous déclarez une propriété pour la première fois, vous n'êtes pas obligé de spécifier **nodefault** car, dans ce cas, l'absence de valeur par défaut a la même signification.

Voici la déclaration d'un composant qui comprend une propriété booléenne unique appelée *IsTrue* dont la valeur par défaut est *True*. La déclaration ci-dessous (dans la section **implémentation** de l'unité) représente le constructeur qui initialise la propriété.

```
type
  TSampleComponent = class(TComponent)
  private
    FIsTrue: Boolean;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property IsTrue: Boolean read FIsTrue write FIsTrue default True;
  end;
⋮
constructor TSampleComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                          { appelle le constructeur hérité }
  FIsTrue := True;                                     { définit la valeur par défaut }
end;
```

Création de propriétés tableau

Certaines propriétés se prêtent à l'indexation. Par exemple, la propriété *Lines* de *TMemo* est la liste indexée des chaînes qui constituent le texte du mémo et vous pouvez la traiter comme un tableau de chaînes. *Lines* fournit un accès à un

élément particulier (une chaîne) dans un ensemble plus large de données (le texte du mémo).

Les propriétés tableau sont déclarées comme les autres propriétés. Les seules différences sont les suivantes :

- La déclaration de la propriété doit comprendre un ou plusieurs index ayant chacun un type défini. Les index peuvent avoir n'importe quel type.
- Les parties **read** et **write** de la déclaration de la propriété, lorsqu'elles sont spécifiées, doivent être des méthodes. Il ne peut s'agir de champs.

Les méthodes read et write d'une propriété tableau acceptent des paramètres supplémentaires correspondant aux index. Les paramètres doivent respecter l'ordre et le type des index spécifiés dans la déclaration.

Bien qu'ils se ressemblent, il existe quelques différences importantes entre les tableaux et les propriétés tableau. Contrairement aux indices d'un tableau, l'index d'une propriété tableau n'est pas obligatoirement de type entier. Par exemple, vous pouvez indexer une propriété en utilisant une chaîne. En outre, vous ne pouvez référencer qu'un seul élément d'une propriété et non une plage d'éléments.

L'exemple suivant est la déclaration d'une propriété renvoyant une chaîne en fonction de la valeur d'un index de type entier.

```
type
  TDemoComponent = class(TComponent)
  private
    function GetNumberName(Index: Integer): string;
  public
    property NumberName[Index: Integer]: string read GetNumberName;
  end;
⋮
function TDemoComponent.GetNumberName(Index: Integer): string;
begin
  Result := 'Inconnu';
  case Index of
    -MaxInt..-1: Result := 'Négatif';
    0: Result := 'Zéro';
    1..100: Result := 'Petit';
    101..MaxInt: Result := 'Grand';
  end;
end;
```

Création de propriétés pour les sous-composants

Par défaut, quand la valeur d'une propriété est un autre composant, vous affectez une valeur à cette propriété en ajoutant une instance de l'autre composant dans la fiche ou le module de données, puis en affectant ce composant comme valeur de la propriété. Néanmoins, il est également possible pour votre composant de créer sa propre instance de l'objet qui implémente la valeur de la propriété. Un tel composant dédié est appelé un sous-composant.

Les sous-composants peuvent représenter tout objet persistant (tout descendant de *TPersistent*). A l'inverse des composants distincts qui peuvent être affectés comme valeur d'une propriété, les propriétés publiées des sous-composants sont enregistrées avec le composant qui les crée. Pour que cela fonctionne, les conditions suivantes doivent être satisfaites :

- Le propriétaire (*Owner)* du sous-composant doit être le composant qui le crée et l'utilise comme valeur d'une propriété publiée. Pour les sous-composants qui sont des descendants de *TComponent*, vous pouvez réaliser cela en définissant la propriété *Owner* du sous-composant. Pour les autres sous-composants, vous devez redéfinir la méthode *GetOwner* de l'objet persistant afin qu'elle renvoie le composant de création.
- Si le sous-composant est un descendant de *TComponent*, il doit indiquer qu'il est un sous-composant en appelant la méthode *SetSubComponent*. Généralement, cet appel est effectué par le propriétaire quand il crée le sous-composant ou par le constructeur du sous-composant.

Généralement, les propriétés dont les valeurs sont des sous-composants sont en lecture seule. Si vous permettez la modification d'une propriété dont la valeur est un sous-composant, la méthode de définition de propriété doit libérer le sous-composant quand un autre composant est affecté à la propriété. En outre, le composant ré-instancie souvent son sous-composant lorsque la propriété est définie à nil. Sinon, dès qu'un autre composant est affecté à la propriété, le sous-composant ne peut plus être restauré à la conception. L'exemple suivant illustre une telle méthode de définition d'une propriété dont la valeur est un composant *TTimer* :

```
procedure TDemoComponent.SetTimerProp(Value: TTimer);
begin
  if Value <> FTimer then
  begin
    if Value <> nil then
    begin
      if (FTimer <> nil and FTimer.Owner = self then
        FTimer.Free;
      FTimer := Value;
      FTimer,FreeNotification(self);
    end
    else { valeur nil }
    begin
      if FTimer.Owner <> self then
      {
        FTimer := TTimer.Create(self);
        FTimer.SetSubComponent(True);
        FTimer.FreeNotification(self);
      }
    end;
  end;
end;
```

Notez que la méthode de définition de propriété ci-dessus a appelé la méthode *FreeNotification* du composant affecté à la propriété. Cet appel garantit que le composant représentant la valeur de la propriété envoie une notification s'il est

sur le point d'être détruit. Il envoie cette notification en appelant la méthode *Notification*. Vous gérez cet appel en redéfinissant la méthode *Notification*, comme suit :

```
procedure TDemoComponent.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and (AComponent = FTimer) then
    FTimer := nil;
end;
```

Création de propriétés pour les interfaces

Vous pouvez utiliser une interface en tant que valeur d'une propriété publiée. Néanmoins, le mécanisme par lequel votre composant reçoit les notifications de l'implémentation de cette interface diffère. Dans la rubrique précédente, la méthode de définition de propriété a appelé la méthode *FreeNotification* du composant qui a été affecté en tant que valeur de la propriété. Cela a permis au composant d'effectuer la mise à jour lui-même quand le composant représentant la valeur de la propriété a été libéré. Néanmoins, quand la valeur d'une propriété est une interface, vous n'avez pas accès au composant qui implémente cette interface. En conséquence, vous ne pouvez pas appeler sa méthode *FreeNotification*.

Pour gérer cette situation, vous pouvez appeler la méthode *ReferenceInterface* de votre composant :

```
procedure TDemoComponent.SetMyIntfProp(const Value: IMyInterface);
begin
  ReferenceInterface(FIntfField, opRemove);
  FIntfField := Value;
  ReferenceInterface(FIntfField, opInsert);
end;
```

L'appel de la méthode *ReferenceInterface* avec une interface spécifiée fait la même chose que l'appel de la méthode *FreeNotification* d'un autre composant. Ainsi, après l'appel de la méthode *ReferenceInterface* à partir de la méthode de définition de propriété, vous pouvez redéfinir la méthode *Notification* pour gérer les notifications depuis l'implémenteur de l'interface:

```
procedure TDemoComponent.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Assigned(MyIntfProp)) and (AComponent.IsImplementorOf(MyInftProp)) then
    MyIntfProp := nil;
end;
```

Notez que le code *Notification* affecte **nil** à la propriété *MyIntfProp*, et pas le champ privé (*FIntfField*). Cela garantit que *Notification* appelle la méthode de définition de propriété, qui appelle *ReferenceInterface* pour retirer la demande de notification qui a été établie quand la valeur de la propriété a été préalablement définie. Toutes les affectations de la propriété d'interface doivent être effectuées par le biais de la méthode de définition de propriété.

Stockage et chargement des propriétés

Delphi stocke les fiches et leurs composants dans des fichiers fiche (.dfm dans la VCL et .xfm dans la CLX). Un fichier fiche stocke les propriétés d'une fiche et de ses composants. Lorsque les développeurs Delphi ajoutent à leurs fiches les composants que vous avez écrits, vos composants doivent être capables d'enregistrer leurs propriétés dans le fichier fiche lors de sa sauvegarde. De même, lorsqu'ils sont chargés dans Delphi ou exécutés comme éléments d'une application, vos composants doivent être capables de se restituer eux-mêmes à partir du fichier fiche.

La plupart du temps, vous n'aurez rien à faire pour que vos composants fonctionnent avec un fichier fiche car la fonction de stockage et de chargement d'une représentation fait partie du comportement reçu en héritage par tous les composants. Toutefois, dans certaines circonstances, vous pouvez souhaiter modifier le stockage d'un composant ou son initialisation au chargement. C'est pourquoi il est conseillé de comprendre les mécanismes sous-jacents.

Les aspects du stockage de propriétés qu'il est nécessaire d'expliquer sont les suivants :

- Utilisation du mécanisme de stockage et de chargement
- Spécification des valeurs par défaut
- Détermination du stockage
- Initialisation après chargement
- Stockage et chargement des propriétés non publiées

Utilisation du mécanisme de stockage et de chargement

La description d'une fiche est la liste des propriétés de la fiche accompagnée d'une liste semblable pour chacun de ses composants. Chaque composant, y compris la fiche elle-même, est responsable du stockage et du chargement de sa propre description.

Lorsqu'il se stocke lui-même, un composant écrit implicitement les valeurs de toutes ses propriétés publiques ou publiées si celles-ci sont différentes de leurs valeurs par défaut, en respectant l'ordre dans lequel ont été déclarées ces valeurs. Au chargement, le composant commence par se construire lui-même, toutes les propriétés récupérant leurs valeurs par défaut, puis il lit les valeurs stockées des propriétés dont les valeurs ne correspondent pas aux valeurs par défaut.

Ce mécanisme implicite répond à la plupart des besoins des composants et ne nécessite aucune intervention particulière de la part de l'auteur du composant. Néanmoins, il existe plusieurs moyens de personnaliser le processus de stockage et de chargement pour répondre aux besoins particuliers d'un composant.

Spécification des valeurs par défaut

Les composants Delphi ne stockent la valeur des propriétés que si elles diffèrent des valeurs par défaut. Sauf indication contraire, Delphi suppose qu'une propriété n'a pas de valeur par défaut, ce qui a pour conséquence que le composant stocke toujours la propriété, quelle que soit sa valeur.

Pour spécifier une valeur par défaut pour une propriété, ajoutez la directive **default** et la nouvelle valeur par défaut à la fin de la déclaration de la propriété.

Vous pouvez également spécifier une valeur par défaut en redéclarant une propriété. En fait, l'attribution d'une autre valeur par défaut est l'une des raisons qui peut vous amener à redéclarer une propriété.

Remarque La spécification d'une valeur par défaut n'a pas pour effet d'attribuer cette valeur à la propriété lorsque l'objet est créé. Le constructeur du composant doit s'en charger. Une propriété dont la valeur n'est pas définie par le constructeur du composant, a la valeur zéro, ou la valeur affichée par la propriété quand son stockage en mémoire est 0. Par défaut, les nombres valent donc 0, les booléens *False*, les pointeurs **nil**, etc. En cas de doute, affectez une valeur dans la méthode du constructeur.

Le code suivant montre la déclaration d'un composant qui attribue une valeur par défaut à la propriété *Align,* ainsi que l'implémentation du constructeur du composant qui affecte cette valeur. Dans notre exemple, le nouveau composant est un cas particulier du composant volet standard utilisé en tant que barre d'état dans une fenêtre. Il doit donc implicitement s'aligner sur la bordure inférieure de son propriétaire.

```
type
  TStatusBar = class(TPanel)
  public
    constructor Create(AOwner: TComponent); override;          { surcharge pour définir la
                                                              // nouvelle valeur par défaut }
  published
    property Align default alBottom;       { redéclare avec la nouvelle valeur par défaut }
  end;
⋮
constructor TStatusBar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                          { effectue une initialisation héritée }
  Align := alBottom;                    { affecte une nouvelle valeur par défaut à Align }
end;
```

Détermination du stockage

Vous pouvez choisir si Delphi stocke ou non chacune des propriétés de vos composants. Par défaut, sont stockées toutes les propriétés de la partie **published** de la déclaration de classe. Vous pouvez choisir de ne pas stocker une propriété ou de désigner une fonction qui détermine dynamiquement le stockage de la propriété.

Pour contrôler le stockage par Delphi d'une propriété, ajoutez la directive **stored** à la déclaration de propriété, suivie par *true*, *false* ou le nom d'une fonction booléenne.

Le code suivant montre un composant avec la déclaration de trois nouvelles propriétés. La première est toujours stockée, la deuxième ne l'est jamais et la troisième est stockée selon la valeur d'une fonction booléenne :

```
type
  TSampleComponent = class(TComponent)
  protected
    function StoreIt: Boolean;
  public
  ⋮
  published
    property Important: Integer stored True;         { toujours stockée }
    property Unimportant: Integer stored False;      { jamais stockée }
    property Sometimes: Integer stored StoreIt;      { dépend de la valeur de la fonction }
  end;
```

Initialisation après chargement

Après qu'un composant a lu les valeurs de toutes ses propriétés dans sa description stockée, il appelle une méthode virtuelle appelée *Loaded*, qui effectue toutes les initialisations nécessaires. L'appel de *Loaded* s'exécute avant que ne s'affichent la fiche et ses contrôles, ainsi vous n'avez pas à vous soucier du scintillement de l'écran provoqué par ces initialisations.

Pour initialiser un composant après le chargement des valeurs des propriétés, vous devez surcharger la méthode *Loaded*.

Remarque La première opération à accomplir dans une méthode *Loaded* consiste à appeler la méthode *Loaded* reçue en héritage. Ceci afin d'être sûr que toutes les propriétés reçues en héritage sont correctement initialisées avant d'effectuer l'initialisation de votre propre composant.

Le code suivant provient du composant *TDatabase*. Après chargement, la base de données essaie de rétablir toutes les connexions ouvertes au moment du stockage, et spécifie comment gérer toutes les exceptions qui se produisent pendant la connexion.

```
procedure TDatabase.Loaded;
begin
  inherited Loaded;                                  { appelle d'abord la méthode héritée }
  try
    if FStreamedConnected then Open                    { rétablit les connexions }
    else CheckSessionName(False);
  except
    if csDesigning in ComponentState then                  { lors de la conception... }
      Application.HandleException(Self)          { permet à Delphi de gérer l'exception }
    else raise;                                                { sinon, redéclenche }
  end;
end;
```

Stockage et chargement des propriétés non publiées

Par défaut, seules les propriétés publiées sont chargées et enregistrées avec un composant. Cependant, il est possible de charger et d'enregistrer des propriétés non publiées. Ceci permet d'obtenir des propriétés persistantes n'apparaissant pas dans l'inspecteur d'objets. Cela permet aussi le stockage et le chargement des valeurs de propriétés par les composants, valeurs de propriétés que Delphi ne sait pas comment lire ni écrire car elles sont trop complexes. Par exemple, l'objet *TStrings* ne peut pas compter sur le comportement automatique de Delphi pour stocker et charger les chaînes qu'il représente et doit utiliser le mécanisme suivant.

Vous pouvez enregistrer des propriétés non publiées en ajoutant du code indiquant à Delphi comment charger et enregistrer la valeur de propriété.

Pour écrire votre propre code afin de charger et d'enregistrer des propriétés, utilisez les étapes suivantes :

1. Création de méthodes pour le stockage et le chargement de valeurs de propriétés.
2. Redéfinition de la méthode *DefineProperties*, en transmettant ces méthodes à un objet filer.

Création de méthodes pour le stockage et le chargement de valeurs de propriétés

Pour stocker et charger des propriétés non publiées, vous devez d'abord créer une méthode afin de stocker la valeur de propriété et une autre méthode pour la charger. Deux possibilités s'offrent à vous :

- Créer une méthode de type *TWriterProc* afin de stocker la valeur de propriété et une méthode de type *TReaderProc* pour la charger. Cette approche vous permet de profiter des capacités intégrées de Delphi concernant l'enregistrement et le chargement de types simples. Si la valeur de votre propriété est construite à partir de types que Delphi sait enregistrer et charger, utilisez cette approche.
- Créer deux méthodes de type *TStreamProc*, une pour stocker et une pour charger la valeur de propriété. *TStreamProc* accepte un flux comme argument et vous pouvez utiliser les méthodes du flux afin d'écrire et lire les valeurs de propriétés.

Prenons pour exemple une propriété représentant un composant créé à l'exécution. Delphi sait comment écrire cette valeur, mais ne le fait pas automatiquement car le composant n'est pas créé dans le concepteur de fiches. Puisque le système de flux peut dès à présent charger et enregistrer des composants, vous pouvez utiliser la première approche.

Les méthodes suivantes chargent et stockent le composant créé dynamiquement qui représente la valeur d'une propriété appelée *MyCompProperty* :

```
procedure TSampleComponent.LoadCompProperty(Reader: TReader);
begin
```

```
  if Reader.ReadBoolean then
    MyCompProperty := Reader.ReadComponent(nil);
end;
procedure TSampleComponent.StoreCompProperty(Writer: TWriter);
begin
  Writer.WriteBoolean(MyCompProperty <> nil);
  if MyCompProperty <> nil then
    Writer.WriteComponent(MyCompProperty);
end;
```

Redéfinition de la méthode DefineProperties

Après avoir créé des méthodes de stockage et de chargement de la valeur de propriété, vous pouvez redéfinir la méthode *DefineProperties* du composant. Delphi appelle cette méthode lors du chargement ou du stockage du composant. Dans la méthode *DefineProperties*, vous devez appeler la méthode *DefineProperty* ou *DefineBinaryProperty* du filer en cours, en lui transmettant la méthode à utiliser pour le chargement ou l'enregistrement de la valeur de propriété. Si vos méthodes de chargement et de stockage sont des types *TWriterProc* et *TReaderProc*, vous appelez alors la méthode *DefineProperty* du filer. Si vous avez créé des méthodes de type *TStreamProc*, appelez plutôt *DefineBinaryProperty*.

Peu importe la méthode utilisée pour définir la propriété, vous lui transmettez les méthodes enregistrant et chargeant votre valeur de propriété ainsi qu'une valeur booléenne indiquant si la valeur de propriété doit être écrite ou non. S'il peut s'agir d'une valeur héritée ou si elle a une valeur par défaut, il n'est pas nécessaire de l'écrire.

Soit par exemple la méthode *LoadCompProperty* du type *TReaderProc* et la méthode *StoreCompProperty* du type *TWriterProc*, vous redéfiniriez *DefineProperties* de la manière suivante :

```
procedure TSampleComponent.DefineProperties(Filer: TFiler);
  function DoWrite: Boolean;
  begin
    if Filer.Ancestor <> nil then { recherche l'ancêtre pour une valeur héritée }
    begin
      if TSampleComponent(Filer.Ancestor).MyCompProperty = nil then
        Result := MyCompProperty <> nil
      else if MyCompProperty = nil or
         TSampleComponent(Filer.Ancestor).MyCompProperty.Name <> MyCompProperty.Name then
        Result := True
      else Result := False;
    end
    else { aucune valeur héritée-- cherche la valeur par défaut (nil) }
      Result := MyCompProperty <> nil;
  end;
begin
  inherited; { autorise la définition des propriétés par les classes de base}
  Filer.DefineProperty('MyCompProperty', LoadCompProperty, StoreCompProperty, DoWrite);
end;
```

Chapitre

43

Création d'événements

Un événement est un lien entre une occurrence du système (une action de l'utilisateur ou un changement de focalisation, par exemple) et le fragment de code qui assure effectivement la réponse. Le code de réponse est le *gestionnaire de l'événement*, dont l'écriture est presque toujours du ressort du développeur de l'application. Grâce aux événements, les développeurs peuvent personnaliser le comportement des composants sans avoir besoin de modifier les classes elles-mêmes. Cela s'appelle la *délégation*.

Les événements associés aux actions utilisateur les plus usuelles (par exemple, les actions de la souris) sont intégrés dans les composants standard, mais vous pouvez aussi définir de nouveaux événements. Pour être capable de créer des événements dans un composant, vous devez avoir compris ce qui suit :

- Qu'est-ce qu'un événement ?
- Implémentation des événements standard
- Définition de vos propres événements

Les événements étant implémentés en tant que propriétés, il vaut mieux avoir bien compris le chapitre 42, "Création de propriétés", avant de créer ou de modifier les événements d'un composant.

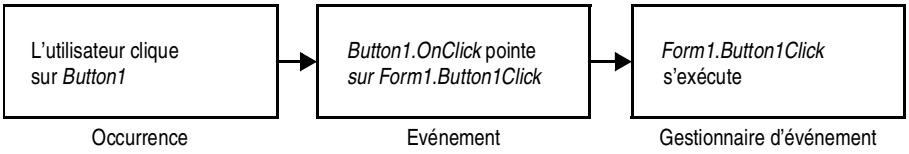
Qu'est-ce qu'un événement ?

Un événement est un mécanisme qui établit un lien entre une occurrence et une partie de code. Plus précisément, un événement est un pointeur de méthodequi pointe sur une méthode dans une instance de classe spécifique.

Du point de vue du développeur d'applications, l'événement est simplement un nom relié à une occurrence du système, comme *OnClick*, auquel du code peut être attaché. Par exemple, un bouton-poussoir appelé *Button1* dispose d'une méthode *OnClick*. Par défaut, Delphi génère l'événement appelé *Button1Click* dans la fiche contenant le bouton et l'associe à l'événement *OnClick*. Lorsqu'un

événement clic de souris se produit sur le bouton, ce dernier appelle la méthode associée à *OnClick qui est pour notre exemple Button1Click*.

Pour écrire un événement, vous devez comprendre ceci :



- Les événements sont des pointeurs de méthodes.
- Les événements sont des propriétés.
- Les types d'événements sont des types de pointeurs de méthodes.
- Les types gestionnaire d'événement sont des procédures.
- Les gestionnaires d'événements sont facultatifs.

Les événements sont des pointeurs de méthodes

Delphi utilise les pointeurs de méthodes pour implémenter les événements. Un pointeur de méthode est un type particulier de pointeur qui pointe sur une méthode spécifique située dans une instance d'objet. En tant qu'auteur de composant, vous pouvez voir les pointeurs de méthodes comme des marques de réservation. Après la détection d'un événement par votre code, vous appelez la méthode (si elle existe) définie par l'utilisateur pour gérer cet événement.

Les pointeurs de méthodes fonctionnent comme les autres types procéduraux, mais ils maintiennent un pointeur caché sur un objet. Quand le développeur d'applications associe un gestionnaire à un événement du composant, l'association ne s'effectue pas seulement avec une méthode ayant un nom particulier, mais avec une méthode d'une instance d'objet particulière. Cet objet est généralement la fiche contenant le composant, mais ce n'est pas toujours le cas.

Tous les contrôles, par exemple, héritent d'une méthode dynamique appelée *Click* pour la gestion des événements de clic avec la souris :

```
procedure Click; dynamic;
```

L'implémentation de Click appelle le gestionnaire utilisateur des événements liés aux clics de la souris, s'il existe. Si l'utilisateur a affecté un gestionnaire à l'événement OnClick d'un contrôle, le fait de cliquer sur ce dernier génère l'appel de la méthode. Si aucun gestionnaire n'est affecté, rien ne se produit.

Les événements sont des propriétés

Ces composants utilisent les propriétés pour implémenter les événements. Mais, contrairement à la plupart des propriétés, les propriétés événement ne font pas appel à des méthodes pour implémenter leurs parties **read** et **write**. Elles utilisent plutôt un champ de classe de même type que la propriété.

Par convention, le nom de la donnée membre est le même que celui de la propriété précédé de la lettre *F*. Par exemple, le pointeur de la méthode *OnClick* est stocké dans une donnée membre intitulée *FOnClick* de type *TNotifyEvent* ; la déclaration de la propriété de l'événement *OnClick* ressemble à ceci :

```
type
  TControl = class(TComponent)
  private
    FOnClick: TNotifyEvent;                    { déclare une donnée membre pour contenir
                                                 // le pointeur de méthode }
    ⋮
  protected
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
  end;
```

Pour en savoir davantage sur *TNotifyEvent* et sur les autres types d'événements, voir la prochaine section, "Les types d'événements sont des types de pointeurs de méthodes".

Comme pour toute autre propriété, vous pouvez définir ou modifier la valeur d'un événement au moment de l'exécution. L'avantage principal des événements implémentés sous la forme de propriétés est que l'utilisateur de composant peut lui associer un gestionnaire au moment de la conception à l'aide de l'inspecteur d'objets.

Les types d'événements sont des types de pointeurs de méthodes

Comme un événement est un pointeur sur un gestionnaire d'événement, le type d'une propriété événement correspond nécessairement à un pointeur de méthode. De même, tout code utilisé comme gestionnaire d'événement doit être de type méthode d'objet.

Toutes les méthodes gestionnaire d'événement sont des procédures. Pour être compatible avec un événement d'un type particulier, une méthode gestionnaire d'événement doit avoir le même nombre de paramètres, les paramètres étant de même type et transmis dans le même ordre.

Delphi définit des types de méthode pour tous les événements standard. Lorsque vous créez vos propres événements, vous pouvez utiliser un type existant s'il est approprié ou définir votre propre type.

Les types gestionnaire d'événement sont des procédures

Bien que le compilateur vous permet de déclarer des types pointeur de méthode qui sont des fonctions, vous ne devrez jamais le faire pour la gestion d'événements. Comme une fonction vide renvoie un résultat non défini, un gestionnaire d'événement vide qui était une fonction ne pourra pas toujours être correct. Pour cette raison, tous vos événements et leurs gestionnaires d'événements associés doivent être des procédures.

Bien qu'un gestionnaire d'événement ne puisse pas être une fonction, vous pouvez toujours obtenir les informations à partir du code du développeur de l'application en utilisant les paramètres **var**. Lorsque vous effectuerez ceci,

vérifiez que vous affectez une valeur correcte au paramètre avant d'appeler le gestionnaire afin de ne pas rendre obligatoire la modification de la valeur par le code de l'utilisateur.

Un exemple de transmission des paramètres **var** à un gestionnaire d'événement est fourni par l'événement *OnKeyPress*, de type *TKeyPressEvent*. *TKeyPressEvent* définit deux paramètres, l'un indiquant l'objet ayant généré l'événement et l'autre la touche enfoncée :

```
type
  TKeyPressEvent = procedure(Sender: TObject; var Key: Char) of object;
```

Normalement, le paramètre *Key* contient le caractère tapé par l'utilisateur. Toutefois dans certaines circonstances, l'utilisateur de composant peut souhaiter changer ce caractère. Par exemple, pour forcer tous les caractères en majuscules dans un éditeur. Dans un cas comme celui-là, l'utilisateur doit définir le gestionnaire suivant pour gérer les frappes de touches :

```
procedure TForm1.Edit1KeyPressed(Sender: TObject; var Key: Char);
begin
  Key := UpCase(Key);
end;
```

Vous pouvez également utiliser les paramètres **var** pour permettre à l'utilisateur de surcharger la gestion par défaut.

Les gestionnaires d'événements sont facultatifs

Lorsque vous créez des composants, n'oubliez pas que les développeurs qui les utilisent ne vont pas forcément leur associer des gestionnaires. Cela signifie que l'exécution de vos composants ne doit pas échouer ni générer d'erreur parce que l'utilisateur n'a pas associé un gestionnaire à un événement. Le mécanisme pour appeler un gestionnaire et gérer les événements, alors que l'utilisateur n'a pas associé de gestionnaire, est décrit dans "Appel de l'événement" à la page 43-9.)

Des événements se produisent en permanence dans une application GUI. Le simple fait de déplacer le pointeur de la souris sur un composant visuel envoie de nombreux messages de déplacement de la souris que le composant traduit en événements *OnMouseMove*. Dans la plupart des cas, les développeurs ne veulent pas gérer les événements déplacement de souris et il ne faut pas que cela pose un problème. Aussi, les composants que vous créez doivent posséder des gestionnaires pour ces événements.

Mais surtout, les développeurs d'applications peuvent écrire le code qu'ils veulent dans un gestionnaire d'événement. Les gestionnaires d'événements des composants de la VCL et de la CLX sont écrits de façon à minimiser le risque de génération d'erreurs. Evidemment, vous ne pouvez pas empêcher les erreurs de logique dans le code de l'application, mais vous pouvez vous assurer que les structures de données sont initialisées avant que les événements ne soient appelés, de sorte que les développeurs ne tentent pas d'accéder à des données incorrectes.

Implémentation des événements standard

Les contrôles fournis avec Delphi héritent des événements correspondant aux occurrences les plus courantes. Ces événements sont appelés *événements standard*. Bien que tous ces événements soient intégrés aux contrôles standard, ils sont souvent déclarés **protected**, pour que les développeurs ne puissent pas leur associer de gestionnaire. Lorsque vous créez un contrôle, vous pouvez choisir de rendre visibles certains événements aux utilisateurs de votre contrôle.

Les trois points suivants sont à prendre en compte pour incorporer des événements standard dans vos contrôles :

- Identification des événements standard
- Rendre visibles des événements
- Changement de la gestion des événements standard

Identification des événements standard

Il existe deux catégories d'événements standard : ceux définis pour tous les contrôles et ceux définis uniquement pour les contrôles fenêtrés standard.

Evénements standard pour tous les contrôles

Les événements de base sont définis dans la classe *TControl*. Tous les contrôles, qu'ils soient fenêtrés, graphiques ou personnalisés, héritent de ces événements. Les événements suivants sont disponibles pour l'ensemble des contrôles :

OnClick	*OnDragDrop*	*OnEndDrag*	*OnMouseMove*
OnDblClick	*OnDragOver*	*OnMouseDown*	*OnMouseUp*

Les événements standard disposent de méthodes virtuelles protégées, déclarées dans *TControl*, dont les noms correspondent aux noms des événements. Par exemple, les événements *OnClick* appellent une méthode nommée *Click*, et les événements *OnEndDrag* appellent une méthode nommée *DoEndDrag*.

Evénements standard pour les contrôles standard

Outre les événements communs à tous les contrôles, les contrôles fenêtrés standard (ceux descendant de *TWinControl* dans la VCL et de *TWidgetControl* dans la CLX) disposent des événements suivants :

OnEnter	*OnKeyDown*	*OnKeyPress*
OnKeyUp	*OnExit*	

Comme les événements standard de *TControl*, les événements des contrôles fenêtrés disposent de méthodes correspondantes. Les événements de touches standard présentés ci-dessus répondent à toutes les frappes de touches normales.

Remarque VCL Pour répondre aux frappes de touches spéciales (comme la touche Alt), vous devez répondre au message WM_GETDLGCODE ou CM_WANTSPECIALKEYS de Windows. Voir chapitre 46, "Gestion des messages", pour plus d'informations sur l'écriture de gestionnaires de messages.

Rendre visibles des événements

Les événements standard de *TControl* et *TWinControl* (*TWidgetControl* dans la CLX) sont déclarés **protected**, de même que les méthodes correspondantes. Si vous héritez de l'une de ces classes abstraites et voulez rendre leurs événements accessibles à l'exécution ou pendant la conception, vous devez redéclarer les événements soit public, soit published.

La redéclaration d'une propriété sans spécifier d'implémentation conserve les mêmes méthodes d'implémentation en ne modifiant que leur niveau de protection. Vous pouvez donc prendre en compte un événement défini dans *TControl* mais non visible, et le rendre visible en le déclarant **public** ou **published**.

Par exemple, pour créer un composant qui rende visible en mode conception l'événement *OnClick*, ajoutez les lignes suivantes à la déclaration de classe du composant.

```
type
  TMyControl = class(TCustomControl)
  ⋮
  published
    property OnClick;
  end;
```

Changement de la gestion des événements standard

Pour modifier la façon dont votre composant répond à un certain type d'événement, vous pouvez être tenté d'écrire un fragment de code pour l'associer à l'événement en question. C'est précisément ce que ferait le développeur d'applications. Mais lorsque vous créez un composant, vous devez faire en sorte que l'événement reste disponible pour les développeurs qui vont utiliser le composant.

C'est ce qui justifie les méthodes protégées de l'implémentation associées à chaque événement standard. En surchargeant la méthode d'implémentation, vous pouvez modifier la gestion interne de l'événement et en appelant la méthode reçue en héritage, vous préservez la gestion standard, y compris la gestion de l'événement par le code du développeur d'applications.

L'ordre dans lequel vous appelez les méthodes est significatif. En règle générale, vous appelez d'abord la méthode héritée pour que le gestionnaire d'événement du développeur d'applications s'exécute avant vos modifications (et parfois, empêche l'exécution de vos modifications). Toutefois, dans certaines situations, vous voulez exécuter votre code avant d'appeler la méthode héritée. Par exemple, si le code reçu en héritage dépend d'une façon ou d'une autre de l'état de composant et si votre

code agit sur cet état, vous devez d'abord effectuer ces changements d'état avant d'autoriser le code de l'utilisateur à y répondre.

Supposons que vous écrivez un composant et que vous souhaitez modifier la façon dont il répond aux clics de souris. Au lieu d'associer un gestionnaire à l'événement *OnClick*, comme le ferait le développeur d'applications, surchargez la méthode protégée *Click* :

```
procedure click override       { déclaration forward }
⋮

procedure TMyControl.Click;
begin
  inherited Click;        { exécute la gestion standard, y compris l'appel au gestionnaire}
...                       { vos modifications s'insèrent ici}
end;
```

Définition de vos propres événements

Il est relativement rare de définir des événements entièrement nouveaux. Toutefois, il peut arriver qu'un comportement complètement différent soit introduit par un composant et il faut alors lui définir un événement.

Voici les étapes qui interviennent dans la définition d'un événement :

- Déclenchement de l'événement
- Définition du type de gestionnaire
- Déclaration de l'événement
- Appel de l'événement

Déclenchement de l'événement

Vous avez besoin de savoir ce qui a déclenché l'événement. Pour certains événements, la réponse est évidente. Par exemple, un événement associé à l'enfoncement du bouton de souris se produit lorsque l'utilisateur clique avec le bouton gauche de la souris provoquant l'envoi par Windows d'un message *WM_LBUTTONDOWN* à l'application. La réception de ce message provoque l'appel de la méthode *MouseDown* d'un composant qui à son tour appelle le code que l'utilisateur a associé à l'événement *OnMouseDown*.

Néanmoins, certains événements sont liés de façon moins évidente à des occurrences externes moins spécifiques. Par exemple, une barre de défilement dispose d'un événement *OnChange* qui peut être déclenché par plusieurs occurrences, telles des frappes de touche, des clics de souris, ou des modifications dans d'autres contrôles. Lorsque vous définissez vos événements, assurez-vous que les occurrences appellent tous les événements appropriés.

Deux sortes d'événements

Les deux sortes d'occurrences pour lesquelles vous pouvez être amené à définir des événements sont les interactions utilisateur et les modifications d'état. Les

événements de type interaction utilisateur sont pratiquement toujours déclenchés par un message issu de Windows indiquant que l'utilisateur a agi sur votre composant d'une façon qui peut nécessiter une réponse de votre part. Les événements de modification d'état peuvent aussi être le fait de messages issus de Windows (par exemple, des changements de focalisation ou d'activation). Cependant, ils peuvent également survenir à la suite d'une modification de propriété ou de l'exécution d'une autre partie de code.

Vous disposez d'un contrôle total sur le déclenchement des événements que vous avez vous-même définis. Définissez les événements avec soin de sorte que les développeurs soient capables de les comprendre et de les utiliser.

Définition du type de gestionnaire

Après avoir détecté que l'un de vos événements s'est produit, vous devez définir la façon de le gérer. Cela implique que vous devez déterminer le type du gestionnaire d'événement. Dans la plupart des cas, les gestionnaires d'événements que vous définissez vous-même seront des notifications simples ou spécifiques à des événements particuliers. Il est également possible de récupérer de l'information en provenance du gestionnaire.

Notifications simples

Un événement de type notification ne fait qu'indiquer qu'un événement particulier s'est produit sans fournir aucune information sur le moment et l'endroit où il s'est produit. Les notifications utilisent le type *TNotifyEvent*, qui véhiculent un paramètre unique correspondant à l'émetteur de l'événement. Les seuls éléments "connus" du gestionnaire associé à une notification sont donc le type d'événement et le composant impliqué. Par exemple, les événements clic de souris sont des notifications. Lorsque vous écrivez un gestionnaire pour un événement de ce type, vous ne récupérez que deux informations : le fait qu'un clic s'est produit et le composant impliqué.

Une notification est un processus à sens unique. Il n'existe aucun mécanisme pour renvoyer une information en retour ou pour inhiber la gestion d'une notification.

Gestionnaires d'événements spécifiques

Dans certains cas, savoir qu'un événement s'est produit et connaître le composant impliqué n'est pas suffisant. Par exemple, si l'événement correspond à l'enfoncement d'une touche, le gestionnaire voudra savoir quelle est cette touche. Dans un cas comme celui-là, vous devez disposer d'un gestionnaire qui accepte des paramètres pour ces informations supplémentaires.

Si votre événement a été généré en réponse à un message, les paramètres transmis au gestionnaire d'événement seront vraisemblablement issus des paramètres du message.

Renvoi d'informations à partir du gestionnaire

Comme tous les gestionnaires d'événements sont des procédures, la seule façon de renvoyer des informations à partir d'un gestionnaire consiste à faire appel à un paramètre **var**. Vos composants peuvent utiliser les informations ainsi récupérées pour déterminer le traitement éventuel d'un événement après l'exécution du gestionnaire de l'utilisateur.

Par exemple, tous les événements liés aux touches (*OnKeyDown*, *OnKeyUp* et *OnKeyPress*) transmettent par référence la valeur de la touche enfoncée dans un paramètre intitulé *Key*. Le gestionnaire d'événement peut changer *Key* de façon à donner l'impression à l'application qu'une touche différente est impliquée dans l'événement. Cela permet par exemple de forcer en majuscules les caractères tapés.

Déclaration de l'événement

Une fois déterminé le type de votre gestionnaire d'événement, vous pouvez déclarer le pointeur de méthode et la propriété pour l'événement. N'oubliez pas d'attribuer un nom à l'événement qui soit à la fois significatif et descriptif pour que l'utilisateur puisse comprendre son rôle. Dans la mesure du possible, choisissez des noms de propriétés qui ressemblent à ceux de composants déjà définis.

Les noms d'événement débutent par "On"

Dans Delphi, les noms de la plupart des événements commencent par "On". Il s'agit d'une simple convention ; le compilateur n'impose pas cette restriction. L'inspecteur d'objets détermine qu'une propriété est un événement en examinant le type de la propriété : toutes les propriétés de type pointeur de méthode sont interprétées comme des événements et apparaissent donc dans la page Evénements.

Les développeurs s'attendent à trouver les événements dans la liste alphabétique à l'endroit des noms commençant par "On." Vous risquez d'introduire une certaine confusion en utilisant une autre convention.

Remarque Exception principale à cette règle : les nombreux événements qui se produisent avant ou après certaines occurrences commencent par "Before" ou "After".

Appel de l'événement

Il est préférable de centraliser tous les appels à un événement. Autrement dit, créez une méthode virtuelle dans votre composant qui appelle le gestionnaire d'événement de l'application (s'il a été défini) et qui fournit une gestion par défaut.

Le fait de rassembler tous les appels à un événement en un seul endroit vous permet d'être sûr qu'un programmeur, qui dérive un nouveau composant à partir du vôtre, pourra personnaliser la gestion de l'événement en surchargeant cette méthode sans avoir à parcourir votre code pour repérer les endroits où l'événement est appelé.

Deux autres considérations sont à prendre en compte concernant l'appel de l'événement :

- Les gestionnaires vides doivent être valides.
- Les utilisateurs peuvent surcharger la gestion par défaut.

Les gestionnaires vides doivent être valides

Vous ne devez jamais créer une situation dans laquelle un gestionnaire d'événement vide provoque une erreur, ou dans laquelle le bon fonctionnement d'un composant dépend d'une réponse spécifique provenant du code de gestion d'un événement dans l'application.

Un gestionnaire vide doit produire le même effet qu'un gestionnaire absent. Aussi, le code pour appeler le gestionnaire d'événement dans une application doit ressembler à ceci :

```
if Assigned(OnClick) then OnClick(Self);
... { exécute la gestion par défaut }
```

Il ne doit *en aucun cas* ressembler à ceci :

```
if Assigned(OnClick) then OnClick(Self)
else { exécute la gestion par défaut};
```

Les utilisateurs peuvent surcharger la gestion par défaut

Pour certains types d'événements, les développeurs peuvent vouloir remplacer la gestion par défaut ou même supprimer l'ensemble des réponses. Pour permettre cela, vous devez transmettre au gestionnaire un argument par référence et vérifier si le gestionnaire renvoie une certaine valeur.

Cela reste dans la lignée de l'affirmation qu'un gestionnaire vide doit produire le même effet qu'un gestionnaire absent : puisqu'un gestionnaire vide ne modifie en rien la valeur des arguments passés par référence, la gestion par défaut se déroule toujours après l'appel du gestionnaire vide.

Par exemple, lors de la gestion des événements frappe de touches, le développeur d'applications peut omettre la gestion par défaut de la frappe de touches du composant en attribuant le caractère null (#0) au paramètre **var** *Key*. La logique de programmation sous-jacente est la suivante :

```
if Assigned(OnKeyPress) then OnKeyPress(Self, Key);
if Key <> #0 then ... { exécute la gestion par défaut}
```

Le code réel est légèrement différent car il prend également en compte les messages Windows mais la logique reste la même. Par défaut, le composant appelle le gestionnaire défini par l'utilisateur avant d'exécuter la gestion standard. Si le gestionnaire défini par l'utilisateur attribue le caractère null à *Key*, le composant omet l'exécution de la gestion par défaut.

Chapitre

44

Création de méthodes

Les méthodes des composants sont des procédures et des fonctions intégrées dans la structure d'une classe. Il n'existe pratiquement aucune restriction sur ce que peuvent réaliser les méthodes d'un composant, mais Delphi n'en respecte pas moins un certain nombre de standards qu'il est préférable de suivre. Ce sont :

- Eviter les interdépendances
- Noms des méthodes
- Protection des méthodes
- Rendre virtuelles des méthodes
- Déclaration des méthodes

En général, les composants ne doivent pas contenir beaucoup de méthodes et vous devez chercher à minimiser le nombre des méthodes appelées par une application. Il est préférable d'encapsuler sous la forme de propriétés des caractéristiques qu'il serait tentant d'implémenter sous forme de méthodes. Les propriétés fournissent une interface qui s'inscrit parfaitement dans l'environnement Delphi et sont accessibles au moment de la conception.

Eviter les interdépendances

Lorsque vous écrivez un composant, vous devez réduire au minimum les conditions préalables imposées aux développeurs. Dans toute la mesure du possible, les développeurs doivent pouvoir faire ce qu'ils veulent de votre composant, et à tout moment. Il existe des situations où vous ne pourrez répondre à cette exigence mais le but n'en demeure pas moins de s'en approcher au plus près.

La liste suivante donne quelques indications sur ce qu'il faut éviter :

- Les méthodes qu'un utilisateur doit *obligatoirement* appeler pour utiliser un composant.

- Les méthodes qui doivent s'exécuter selon un ordre défini.
- Les méthodes qui placent le composant dans un état ou un mode pour lequel certains événements ou certaines méthodes deviennent incorrectes.

La meilleure façon de gérer les situations de ce type est de fournir le moyen d'en sortir. Par exemple, si l'appel d'une méthode a pour effet de placer votre composant dans un état où l'appel d'une autre méthode s'avère incorrect, vous devez modifier cette seconde méthode de telle manière que si elle est appelée alors que le composant se trouve dans un état impropre, elle corrige cet état avant d'exécuter son propre code principal. Faute de mieux, vous devrez déclencher une exception si l'utilisateur appelle une méthode non valide.

En d'autres termes, si vous générez une situation dans laquelle il existe des interdépendances entre certaines parties de votre code, il est de *votre* responsabilité de vous assurer qu'une utilisation incorrecte du code n'engendre pas de problème. Un message d'avertissement, par exemple, est préférable à une fin d'exécution anormale si l'utilisateur n'a pas respecté ces interdépendances.

Noms des méthodes

Delphi n'impose aucune restriction quant à la façon de nommer les méthodes et leurs paramètres. Toutefois, certaines conventions facilitent l'exploitation des méthodes par les développeurs d'applications. Souvenez-vous que l'architecture même d'un composant a son influence sur les différents types de personnes qui pourront utiliser ce composant.

Si vous avez l'habitude d'écrire du code qui ne s'adresse qu'à un nombre restreint de programmeurs, vous ne vous êtes sans doute jamais interrogé sur le choix du nom des entités que vous manipulez. Il est souhaitable de choisir des noms compréhensibles car vos composants s'adressent à tous, y compris à ceux qui ne connaissent pas bien votre code (ou qui maîtrisent imparfaitement la programmation).

Voici quelques suggestions pour définir des noms de méthode compréhensibles :

- Choisissez des noms descriptifs. Utilisez des verbes d'action.

  Un nom tel que *CollerPressepapiers* est plus explicite que *Coller* ou *CP*.

- Les noms de fonctions doivent refléter la nature de ce qu'elles renvoient.

  Bien qu'il puisse paraître évident, à vous programmeur, que le rôle d'une fonction intitulée *X* soit de renvoyer la coordonnée horizontale d'un élément, un nom tel que *ObtenirPositionHorizontale* sera compris par tout le monde.

Comme dernière considération, assurez-vous que votre méthode ait réellement besoin d'être créée comme telle. Que le nom de votre méthode puisse être un verbe est un bon repère. Si ce n'est pas le cas, demandez-vous s'il ne serait pas préférable de transformer votre méthode en propriété.

Protection des méthodes

Toutes les parties des classes, y compris les champs, les méthodes et les propriétés, ont différents niveaux de protection ou de "visibilité", comme l'explique "Contrôle des accès" à la page 41-4. Il est facile de choisir le niveau qui convient.

La plupart des méthodes écrites dans vos composants sont **publiques** ou **protégées**. Il n'y a généralement pas lieu de déclarer une méthode **private**, à moins qu'elle soit réellement spécifique à ce type de composant, au point que même les composants dérivés ne peuvent pas y accéder.

Méthodes qui doivent être publiques

Toutes les méthodes qui peuvent être appelées par les développeurs d'applications doivent être déclarées **public**. N'oubliez pas que la plupart des appels aux méthodes ont lieu dans les gestionnaires d'événements, aussi les méthodes doivent éviter de gaspiller les ressources système ou de placer le système d'exploitation dans un état où il n'est plus en mesure de répondre à l'utilisateur.

Remarque Les constructeurs et destructeurs sont toujours déclarés **public**.

Méthodes qui doivent être protégées

Toute méthode d'implémentation d'un composant doit être déclarée **protected** afin d'empêcher les applications de les appeler à un moment inopportun. Si vous avez défini des méthodes qui doivent demeurer inaccessibles au code, tout en restant accessibles aux classes dérivées, vous devez les déclarer **protected**.

Par exemple, supposons une méthode dont l'exécution dépend de l'initialisation préalable d'une donnée. Si cette méthode est déclarée publique, il peut arriver que les applications tentent de l'appeler avant l'initialisation de la donnée. Mais, en la déclarant **protected**, les applications ne peuvent le faire directement. Vous pouvez alors définir d'autres méthodes publiques qui se chargent d'initialiser la donnée avant d'appeler la méthode **protected**.

Les méthodes d'implémentation des propriétés doivent être déclarées comme virtuelles et **protected**. Les méthodes ainsi déclarées permettent aux développeurs d'applications de surcharger l'implémentation des propriétés, augmentant leurs fonctionalités ou les remplaçant complètement. De telles propriétés sont complètement polymorphes. Instaurer un accès **protected** à ces méthodes garantit que les développeurs ne pourront pas les appeler par accident ni modifier la propriété par inadvertance.

Méthodes abstraites

Une méthode est parfois déclarée **abstract** dans un composant Delphi. Dans la VCL et la CLX, les méthodes abstraites se produisent habituellement dans les classes dont les noms commencent par "custom", comme dans*TCustomGrid*. De telles classes sont elles-mêmes abstraites, au sens où elles ne servent qu'à la dérivation de classes descendantes.

Bien que vous puissiez créer un objet instance d'une classe contenant un membre abstrait, ce n'est pas recommandé. L'appel du membre abstrait entraîne une exception EAbstractError.

La directive **abstract** est utilisée pour indiquer des parties de classes qui doivent être surfacées et définies dans des composants descendants ; cela force les écrivains de composants à redéclarer le membre abstrait dans des classes descendantes avant que des instances actuelles de la classe puissent être créées.

Rendre virtuelles des méthodes

Vous rendrez virtuelles les méthodes lorsque vous souhaitez que des types différents puissent exécuter des codes différents en réponse au même appel de méthode.

Si vous créez des composants pour qu'ils soient exploitables par les développeurs d'applications directement, vous voudrez probablement rendre non virtuelles vos méthodes. D'autre part, si vous créez des composants abstraits desquels d'autres composants vont dériver, vous devez envisager de rendre virtuelles les méthodes ajoutées. De cette façon, les composants dérivés pourront surcharger les méthodes virtuelles reçues en héritage.

Déclaration des méthodes

La déclaration d'une méthode dans un composant ne diffère en rien de celle d'une méthode d'une autre classe.

Pour déclarer une nouvelle méthode dans un composant, vous devez :

- Ajouter la déclaration à la déclaration de type du composant dans le fichier en-tête de ce dernier.
- Implémenter la méthode dans la partie **implementation** de l'unité du composant.

Le code suivant montre un composant qui définit deux nouvelles méthodes, l'une est déclarée protected static et l'autre public et virtual.

```
type
  TSampleComponent = class(TControl)
  protected
    procedure MakeBigger;                          { déclare la méthode protected static }
```

```
  public
    function CalculateArea: Integer; virtual;         { déclare la méthode public virtual }
  end;
⋮
implementation
⋮
procedure TSampleComponent.MakeBigger;                    { implémente la première méthode}
begin
  Height := Height + 5;
  Width := Width + 5;
end;

function TSampleComponent.CalculateArea: Integer;         { implémente la deuxième méthode}
begin
  Result := Width * Height;
end;
```

Chapitre

45

Graphiques et composants

Windows fournit une puissante interface GDI (Graphics Device Interface) servant à dessiner des graphiques indépendamment des périphériques. Malheureusement, GDI impose au programmeur des contraintes supplémentaires telles que la gestion des ressources graphiques. Delphi prend en charge toutes ces tâches GDI ingrates, vous laisse vous concentrer sur le travail productif, vous épargnant les recherches de handles perdus ou de ressources non restituées.

De même que toute partie de l'API Windows, vous pouvez appeler les fonctions GDI directement depuis votre application Delphi. Toutefois, vous vous rendrez vite compte que l'utilisation de l'encapsulation Delphi des fonctions graphiques est un moyen plus efficace et plus rapide de créer des graphiques.

Les rubriques de cette section comprennent :

- Présentation des graphiques
- Utilisation du canevas
- Travail sur les images
- Bitmaps hors écran
- Réponse aux changements

Présentation des graphiques

Delphi encapsule à différents niveaux les fonctions GDI de Windows. Le programmeur qui écrit des composants doit comprendre comment ceux-ci affichent leurs images à l'écran. Lorsque vous appelez directement les fonctions GDI, vous devez disposer d'un handle sur un contexte de périphérique dans lequel vous avez sélectionné des outils de dessin comme les crayons, les pinceaux et les fontes. Après avoir tracé vos images, vous devez remettre le contexte de périphérique dans son état initial avant de le restituer.

Remarque CLX Les fonctions GDI sont spécifiques à Windows et ne s'appliquent pas aux applications multiplates-formes CLX.

Pour vous épargner la gestion de vos graphiques à un niveau détaillé, Delphi fournit une interface à la fois simple et complète : il s'agit de la propriété *Canvas des composants*. Le canevas garantit qu'un contexte de périphérique valide est disponible et restitue le contexte lorsqu'il est inutilisé. Il dispose également de propriétés spécifiques pour représenter la fonte, le crayon et le pinceau en cours.

Le canevas gère toutes ces ressources à votre place et vous n'avez donc pas le souci de créer, de sélectionner ou de restituer les éléments comme le handle d'un crayon. Vous n'avez qu'à indiquer au canevas le crayon à utiliser et il se charge du reste.

L'un des avantages de laisser Delphi gérer les ressources graphiques à votre place est que Delphi peut mettre en mémoire cache les ressources en vue d'un usage ultérieur, ce qui accélère considérablement les opérations répétitives. Par exemple, supposons qu'un programme crée, utilise puis restitue un outil crayon d'un certain type plusieurs fois de suite, vous devez répéter ces étapes chaque fois que vous l'utilisez. Comme Delphi stocke en mémoire cache les ressources graphiques, il y a une forte probabilité qu'un outil que vous utilisez de façon répétitive se trouve dans la mémoire cache. Aussi, au lieu de recréer l'outil en question, Delphi se contentera de réutiliser celui qui existe déjà.

Par exemple, vous pouvez imaginer une application avec des dizaines de fiches ouvertes et des centaines de contrôles. Chacun de ces contrôles peut présenter une ou plusieurs propriétés *TFont*. Comme cela peut générer des centaines ou des milliers d'instances d'objets *TFont*, la plupart des applications n'utiliseront que deux ou trois handles de fontes grâce à un cache de fontes.

Voici deux exemples montrant à quel point le code Delphi manipulant des graphiques peut être simple. Le premier utilise les fonctions GDI standard pour dessiner une ellipse jaune bordée de bleu comme le ferait d'autres outils de développement. Le second utilise un canevas pour dessiner la même ellipse dans une application écrite avec Delphi.

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var
  PenHandle, OldPenHandle: HPEN;
  BrushHandle, OldBrushHandle: HBRUSH;
begin
  PenHandle := CreatePen(PS_SOLID, 1, RGB(0, 0, 255));              { crée un crayon bleu }
  OldPenHandle := SelectObject(PaintDC, PenHandle);  { dit à DC d'utiliser le crayon bleu }
  BrushHandle := CreateSolidBrush(RGB(255, 255, 0));              { crée un pinceau jaune }
  OldBrushHandle := SelectObject(PaintDC, BrushHandle);    { dit à DC d'utiliser le pinceau
                                                                                  // jaune }
  Ellipse(HDC, 10, 10, 50, 50);                                        { dessine l'ellipse }
  SelectObject(OldBrushHandle);                                { restaure le pinceau original }
  DeleteObject(BrushHandle);                                     { supprime le pinceau jaune }
  SelectObject(OldPenHandle);                                   { restaure le crayon original }
  DeleteObject(PenHandle);                                          { détruit le crayon bleu }
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
  with Canvas do
  begin
```

```
    Pen.Color := clBlue;                    { crée le crayon bleu }
    Brush.Color := clYellow;                { crée le pinceau jaune }
    Ellipse(10, 10, 50, 50);                { trace l'ellipse }
  end;
end;
```

Utilisation du canevas

La classe canevas encapsule les contrôles graphiques à plusieurs niveaux, allant des fonctions de haut niveau (pour dessiner des lignes, des formes et du texte) aux propriétés de niveau intermédiaire pour manipuler les fonctionnalités de dessin du canevas ; et dans la VCL, offre un accès de bas niveau au GDI Windows.

Le tableau suivant résume les possibilités du canevas.

Tableau 45.1 Résumé des possibilités du canevas

Niveau	Opération	Outils
Haut	Dessin de lignes et de formes	Méthodes comme *MoveTo*, *LineTo*, *Rectangle* et *Ellipse*
	Affichage et mesure de texte	Méthodes *TextOut*, *TextHeight*, *TextWidth* et *TextRect*
	Remplissage de zones	Méthodes *FillRect* et *FloodFill*
Intermédiaire	Personnalisation de texte et des graphiques	Propriétés *Pen*, *Brush* et *Font*
	Manipulation de pixels	Propriété *Pixels*.
	Copie et fusion d'images	Méthodes *Draw*, *StretchDraw*, *BrushCopy* et *CopyRect* ; propriété *CopyMode*
Bas	Appel des fonctions GDI de Windows	Propriété *Handle*

Pour plus d'informations sur les classes canevas, leurs méthodes et leurs propriétés, reportez-vous à l'aide en ligne.

Travail sur les images

Dans Delphi, la part la plus importante de votre travail sur les graphiques se limitera au dessin direct sur le canevas des composants et des fiches. Mais Delphi fournit également les moyens de gérer les images graphiques indépendantes, comme les bitmaps, les métafichiers et les icônes, en assurant dans le même temps la gestion automatique des palettes.

Les trois sujets suivants sont nécessaires à la compréhension du travail sur les images dans Delphi :

- Utilisation d'une image, d'un graphique ou d'un canevas
- Chargement et stockage des graphiques
- Gestion des palettes

Utilisation d'une image, d'un graphique ou d'un canevas

Il existe trois sortes de classes dans Delphi intervenant sur les graphiques :

- Un *canevas* représente une surface de dessin point par point dans une fiche, un contrôle graphique, une imprimante ou un bitmap. Un canevas est toujours une propriété de quelque chose d'autre, jamais une classe autonome.
- Un *graphique* représente une image graphique telle qu'elle se trouve dans un fichier ou une ressource, comme un bitmap, une icône ou un métafichier. Delphi définit les classes *TBitmap*, *TIcon* et *TMetafile*, toutes descendants de l'objet générique *TGraphic*. Vous pouvez aussi définir vos propres classes graphiques. En définissant une interface standard minimale pour tous les graphiques, *TGraphic* fournit un mécanisme simple destiné aux applications pour qu'elles puissent exploiter facilement les différentes sortes de graphiques disponibles.
- Une *image* est le conteneur d'un graphique, elle peut donc contenir n'importe quelle classe graphique. Autrement dit, un élément de type *TPicture* peut contenir un bitmap, une icône, un métafichier, un type de graphique défini par l'utilisateur, et l'application y accède d'une seule façon par l'intermédiaire de la classe image. Par exemple, un contrôle image dispose d'une propriété *Picture*, de type *TPicture*, qui le rend capable d'afficher les images de plusieurs sortes de graphiques.

Souvenez-vous qu'une classe image a toujours un graphique et qu'un graphique peut avoir un canevas. Le seul graphique standard ayant un canevas est *TBitmap*. Normalement, lorsque vous avez affaire à une image, vous travaillez uniquement avec les constituants de la classe graphique exposés via *TPicture*. Si vous voulez accéder aux constituants spécifiques de la classe graphique elle-même, vous pouvez faire référence à la propriété *Graphic* de l'image.

Chargement et stockage des graphiques

Toutes les images et tous les graphiques de Delphi peuvent charger leurs images à partir d'un fichier et les stocker en retour dans ce même fichier (ou dans un autre). Ceci peut s'effectuer à tout moment.

Remarque CLX Vous pouvez aussi charger des images à partir d'un fichier et les enregistrer dans une source MIME Qt ou un objet flux en cas de création de composants CLX.

Pour charger une image dans un objet image à partir d'un fichier, appelez la méthode *LoadFromFile* de l'objet image. Pour enregistrer une image dans un fichier à partir d'un objet image, appelez la méthode *SaveToFile* de l'objet image.

LoadFromFile et *SaveToFile* acceptent le nom d'un fichier comme seul paramètre. *LoadFromFile* utilise l'extension du fichier pour déterminer le type d'objet graphique créé ou chargé. *SaveToFile* utilise le type de fichier approprié au type d'objet graphique enregistré.

Pour charger un bitmap dans l'objet image d'un contrôle image, par exemple, vous devez transmettre le nom du fichier bitmap à la méthode *LoadFromFile* de l'objet image :

```
procedure TForm1.LoadBitmapClick(Sender: TObject);
begin
  Image1.Picture.LoadFromFile('RANDOM.BMP');
end;
```

L'objet image reconnaît .bmp comme extension par défaut des fichiers bitmap, elle crée donc le graphique en tant que *TBitmap*, avant d'appeler la méthode *LoadFromFile* du graphique. Puisque le graphique est un bitmap, l'image est chargée depuis le fichier en tant que bitmap.

Gestion des palettes

Pour les composants VCL, lorsque l'exécution fait intervenir un périphérique supportant les palettes (typiquement un mode vidéo 256 couleurs), Delphi assure automatiquement la réalisation des palettes. Si un contrôle dispose d'une palette, vous pouvez utiliser deux méthodes héritées de *TControl* pour contrôler la façon dont Windows prend en compte la palette.

Les points suivants sont à prendre en compte lorsque vous travaillez avec des palettes :

- Spécification d'une palette pour un contrôle
- Réponse aux changements de palette

La plupart des contrôles n'ont pas besoin de palette. Toutefois, dans le cas des contrôles contenant des images graphiques riches en couleurs (tels que le contrôle image) une interaction entre Windows et le pilote de périphérique écran peut être nécessaire pour afficher correctement le contrôle. Dans Windows, ce processus est appelé *réalisation* de palettes.

La réalisation de palettes est le processus mis en œuvre pour que la fenêtre en avant-plan utilise la totalité de la palette qui lui est associée. Quant aux fenêtres en arrière-plan, elles exploitent autant que possible les couleurs de leurs palettes propres, l'approximation des couleurs manquantes se faisant en prenant la couleur disponible la plus proche dans la palette "réelle". Windows effectue un travail permanent de réalisation des palettes au fur et à mesure que les fenêtres sont déplacées d'avant en arrière-plan.

Remarque Delphi n'assure ni la création, ni la maintenance des palettes autres que celles des bitmaps. Toutefois, si vous disposez d'un handle de palette, les contrôles Delphi peuvent se charger de sa gestion.

Spécification d'une palette pour un contrôle

Pour spécifier la palette d'un contrôle VCL, vous devez surcharger la méthode *GetPalette* du contrôle pour qu'elle renvoie le handle de la palette.

Le fait de spécifier la palette d'un contrôle a deux conséquences pour votre application :

- Cela informe l'application que la palette de votre contrôle doit être réalisée.
- Cela désigne la palette à utiliser pour cette réalisation.

Réponse aux changements de palette

Si votre contrôle VCL spécifie une palette en surchargeant *GetPalette*, Delphi se chargera de répondre automatiquement aux messages de palette en provenance de Windows. La méthode qui gère les messages de palette est *PaletteChanged*.

Le rôle primaire de *PaletteChanged* est de déterminer s'il est nécessaire de réaliser les palettes des contrôles en avant et arrière-plan. Windows gère la réalisation des palettes en faisant en sorte que la fenêtre la plus en avant dispose de la palette d'avant-plan, la résolution des couleurs des autres fenêtres se faisant à l'aide des palettes d'arrière-plan. Delphi va même plus loin car il réalise les palettes des contrôles d'une fenêtre en respectant l'ordre de tabulation. Seul cas où vous voudrez peut-être redéfinir ce comportement : lorsqu'un contrôle, autre que le premier dans l'ordre de tabulation, doit récupérer la palette d'avant-plan.

Bitmaps hors écran

Lorsque vous dessinez des images graphiques complexes, une technique de programmation graphique habituelle consiste à créer d'abord un bitmap hors écran, puis à dessiner l'image dans ce bitmap et à copier ensuite la totalité de l'image du bitmap vers sa destination finale à l'écran. L'utilisation d'une image hors-écran réduit le scintillement causé par un tracé répétitif à l'écran.

Dans Delphi, la classe bitmap qui représente les images point par point stockées dans les ressources et les fichiers, peut également fonctionner en tant qu'image hors écran.

Les points suivants sont à prendre en compte lorsque vous travaillez avec des bitmaps hors écran :

- Création et gestion des bitmaps hors écran.
- Copie des images bitmap.

Création et gestion des bitmaps hors écran

Lorsque vous créez des images graphiques complexes, vous devez généralement éviter de les dessiner directement sur le canevas qui apparaît à l'écran. Au lieu de les dessiner sur le canevas d'une fiche ou d'un contrôle, vous devez plutôt construire un objet bitmap puis dessiner sur son canevas avant de copier la totalité de l'image sur le canevas de l'écran.

La méthode *Paint* d'un contrôle graphique est un exemple d'utilisation typique d'un bitmap hors écran. Comme avec tout objet temporaire, l'objet bitmap doit être protégé par un bloc **try**..**finally** :

```
type
  TFancyControl = class(TGraphicControl)
  protected
    procedure Paint; override;                          { surcharge la méthode Paint }
  end;

procedure TFancyControl.Paint;
var
  Bitmap: TBitmap;                      { variable temporaire pour le bitmap hors écran }
begin
  Bitmap := TBitmap.Create;                                { construit l'objet bitmap }
  try
    { draw on the bitmap }
    { copy the result into the control's canvas }
  finally
    Bitmap.Free;                                            { détruit l'objet bitmap }
  end;
end;
```

Copie des images bitmap

Delphi fournit quatre moyens pour copier des images d'un canevas à un autre. Selon l'effet recherché, vous pouvez appeler différentes méthodes.

Le tableau suivant résume les méthodes de copie d'image des objets canevas.

Tableau 45.2 Méthodes de copie des images

Pour créer cet effet	Appelez cette méthode
Copie d'un graphique entier.	Draw
Copie et redimensionnement d'un graphique.	StretchDraw
Copie d'une partie d'un canevas.	CopyRect
Copie d'un bitmap avec effets.	BrushCopy

Réponse aux changements

Tous les objets graphiques, y compris les canevas et les objets dont ils sont propriétaires (crayons, pinceaux et fontes), disposent d'événements intégrés pour répondre aux changements. Grâce à ces événements, vous pouvez faire en sorte que vos composants (ou les applications qui les utilisent) répondent aux changements en redessinant leurs images.

S'agissant d'objets graphiques, la réponse aux changements est particulièrement importante si ces objets sont publiés comme éléments accessibles de l'interface de conception de vos composants. La seule façon d'être certain que l'aspect d'un

composant au moment de la conception corresponde aux propriétés définies dans l'inspecteur d'objets consiste à répondre aux changements apportés aux objets.

Pour répondre aux modifications d'un objet graphique, vous devez associer une méthode à l'événement *OnChange* de sa classe.

Le composant forme publie les propriétés représentant le crayon et le pinceau qu'il utilise pour tracer sa forme. Le constructeur du composant associe une méthode à l'événement *OnChange* de chacun, ce qui a pour effet de provoquer le rafraîchissement de l'image du composant si le crayon ou le pinceau est modifié :

```
type
  TShape = class(TGraphicControl)
  public
    procedure StyleChanged(Sender: TObject);
  end;
⋮
implementation
⋮
constructor TShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                  { appelle toujours le constructeur hérité ! }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                          { construit le crayon }
  FPen.OnChange := StyleChanged;             { affecte la méthode à l'événement OnChange }
  FBrush := TBrush.Create;                                     { construit le pinceau }
  FBrush.OnChange := StyleChanged;           { affecte la méthode à l'événement OnChange }
end;

procedure TShape.StyleChanged(Sender: TObject);
begin
  Invalidate();                                       { détruit et redessine le composant }
end;
```

Chapitre

46

Gestion des messages

La gestion des *messages* envoyés par Windows aux applications est l'une des clés de la programmation Windows traditionnelle. Delphi gère la plupart des messages standard à votre place. Mais, vous aurez peut-être à gérer des messages que Delphi ne prend pas en compte, ou encore à créer puis à gérer vos propres messages. Les composants CLX ne gèrent pas les messages Windows mais vous pouvez créer des gestionnaires de messages pour vos propres messages.

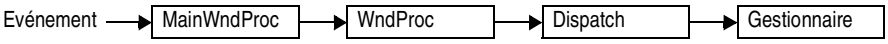
Il y a trois aspects à prendre en considération lorsque vous travaillez avec des messages :

- Compréhension du système de gestion des messages
- Modification de la gestion des messages
- Création de nouveaux gestionnaires de messages

Compréhension du système de gestion des messages

Toutes les classes Delphi disposent d'un mécanisme intégré pour gérer les messages : ce sont les *méthodes de gestion des messages* ou *gestionnaires de messages*. L'idée sous-jacente aux gestionnaires de messages est la suivante : un objet reçoit des messages qu'il répartit selon le message en appelant une méthode choisie dans un ensemble de méthodes spécifiques. Un gestionnaire par défaut est appelé si aucune méthode n'est définie pour le message.

Le diagramme suivant illustre le fonctionnement du système de répartition de message :



La bibliothèque des composants visuels définit un système de répartition des messages qui convertit tous les messages Windows (y compris ceux définis par l'utilisateur) destinés à une classe spécifique en appels à des méthodes. Pour la CLX, le système de répartition ne comprend pas MainWndProc et WndProc.

Vous n'aurez sans doute jamais besoin de modifier le mécanisme de répartition des messages. En revanche, vous aurez à écrire des méthodes de gestion des messages. Voir "Déclaration d'une nouvelle méthode de gestion d'un message" à la page 46-7, pour plus de détails sur ce sujet.

Que contient un message Windows ?

Remarque Cette information s'applique uniquement à l'écriture de composants VCL.

Un message Windows est un enregistrement de données contenant plusieurs données membres exploitables. Le plus important est celui qui contient une valeur de la taille d'un entier identifiant le message. Windows définit de nombreux messages et l'unité *Messages* déclare tous leurs identificateurs.

Un paramètre contient 16 bits, l'autre 32 bits. Vous voyez souvent du code Windows qui fait référence à ces valeurs avec *wParam* et *lParam*, comme "paramètre de type word" et "paramètre de type long." Souvent, chaque paramètre contient une information, qui fait référence à un mot de poids fort dans le paramètre de type long.

A l'origine, un programmeur Windows devait mémoriser le contenu de chaque paramètre ou consulter l'API Windows. Microsoft a récemment donné un nom aux paramètres. Ces "décomposeurs de message" ainsi appelés simplifient la compréhension des informations accompagnant chaque message. Par exemple, les paramètres pour le message *WM_KEYDOWN* maintenant appelés *nVirtKey* et *lKeyData*, donnent plus d'informations spécifiques que *wParam* et *lParam*.

Pour chaque type de message, Delphi définit un type d'enregistrement qui donne un nom mnémonique à chaque paramètre. Les messages souris transmettent par exemple les coordonnées x et y de l'événement souris dans le paramètre de type long, une dans le mot de poids fort, et l'autre dans le mot de poids faible. Avec l'utilisation de la structure souris-message, vous n'avez pas à vous soucier du mot dont il s'agit, car vous faites référence aux paramètres par les noms *XPos* et *YPos* au lieu de *lParamLo* et *lParamHi*.

Répartition des messages

Remarque Cette information s'applique uniquement à l'écriture de composants VCL.

Lorsqu'une application crée une fenêtre, elle recense une *procédure fenêtre* avec le modèle Windows. La procédure fenêtre représente la routine qui gère les messages pour la fenêtre. Habituellement, la procédure fenêtre contient une instruction longue **case** avec des entrées pour chaque message devant être géré par la fenêtre. N'oubliez pas que "fenêtre" dans ce sens signifie seulement quelque chose sur l'écran : chaque fenêtre, chaque contrôle, etc. A chaque fois que vous créez un nouveau type de fenêtre, vous devez créer une procédure fenêtre complète.

Delphi simplifie la répartition des messages de plusieurs manières :

- Chaque composant hérite d'un système complet de répartition de message.

- Le système de répartition de message dispose d'une gestion par défaut. Vous ne définissez de gestionnaire que pour les messages auxquels vous souhaitez spécifiquement répondre.
- Vous pouvez modifier des parties de la gestion de message en vous appuyant sur les méthodes reçues en héritage pour la majeure partie du traitement.

Le bénéfice le plus évident de cette répartition de message est le suivant : à tout moment, vous pouvez envoyer n'importe quel message à n'importe quel composant. Si le composant n'a pas de gestionnaire défini pour ce message, le système de gestion par défaut s'en charge, généralement en ignorant le message.

Suivi du flux des messages

La méthode *MainWndProc* est recensée par Delphi comme procédure de fenêtre pour tous les types de composants d'une application. *MainWndProc* contient un bloc de gestion des exceptions qui transmet la structure du message en provenance de Windows à la méthode virtuelle *WndProc*, gérant les exceptions éventuelles à l'aide de la méthode *HandleException* de la classe application.

MainWndProc est une méthode non virtuelle qui n'effectue aucune gestion particulière des messages. Cette gestion a lieu dans *WndProc*, chaque type de composant ayant la possibilité de surcharger cette méthode pour répondre à ses besoins spécifiques.

Les méthodes *WndProc* vérifient les conditions spéciales qui peuvent affecter le traitement et "interceptent", s'il le faut, les messages non souhaités. Par exemple, lorsque vous faites glisser un composant, celui-ci ignore les événements du clavier, et la méthode *WndProc* de *TWinControl* ne transmet ces événements que si l'utilisateur ne fait pas glisser le composant. Enfin, *WndProc* appelle *Dispatch*, une méthode non virtuelle héritée de *TObject*, qui détermine la méthode à appeler pour gérer le message.

Dispatch utilise la donnée membre *Msg* de l'enregistrement du message pour déterminer comment répartir le message particulier. Si le composant définit un gestionnaire pour ce message, *Dispatch* appelle cette méthode. Si aucun gestionnaire n'est défini, *Dispatch* appelle *DefaultHandler*.

Modification de la gestion des messages

Remarque Cette information s'applique uniquement à l'écriture de composants VCL.

Avant de modifier la gestion des messages de vos composants, vous devez être certain de ce que vous voulez effectivement faire. Delphi convertit la plupart des messages en événements que l'auteur ou l'utilisateur du composant peut gérer. Plutôt que de modifier le comportement définissant la gestion du message, vous voudrez généralement modifier le comportement définissant la gestion de l'événement.

Pour modifier la gestion d'un message dans les composants VCL, vous devez surcharger la méthode qui gère ce message. En outre, dans certaines

circonstances, vous pouvez empêcher un composant de gérer un message en interceptant ce message.

Surcharge de la méthode du gestionnaire

Pour modifier la façon dont un composant gère un message en particulier, vous devez surcharger la méthode qui le gère. Si le composant ne gère pas le message en question, vous devez déclarer une nouvelle méthode de gestion du message.

Pour surcharger la méthode de gestion d'un message, déclarez une nouvelle méthode dans votre composant avec le même index de message que la méthode surchargée. N'utilisez *pas* la directive **override** ; vous devez utiliser la directive **message** et un index de message correspondant.

Remarquez qu'il n'est pas nécessaire que le nom de la méthode et le type du paramètre **var** simple correspondent à la méthode surchargée. Seul l'index de message est significatif. Pour plus de clarté, cependant, il est préférable de suivre la convention d'appel des méthodes de gestion de message après les messages qu'elles gèrent.

Par exemple, pour surcharger la gestion du message *WM_PAINT d'un composant*, redéclarez la méthode *WMPaint* :

```
type
  TMyComponent = class(...)
  :
  procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
end;
```

Utilisation des paramètres d'un message

Une fois l'intérieur d'une méthode de gestion de message, votre composant peut accéder à tous les paramètres de la structure du message. Puisque le paramètre passé au gestionnaire message est un paramètre **var**, le gestionnaire peut modifier la valeur du paramètre si c'est nécessaire. Le seul paramètre qui change fréquemment est *le champ Result* du message : il s'agit de la valeur renvoyée par l'appel de *SendMessage* qui a émis le message.

Remarque Cette information s'applique uniquement à l'écriture de composants VCL.

Comme le type du paramètre *Message* transmis à la méthode de gestion dépend du message géré, vous devez vous reporter à la documentation des messages Windows pour connaître le nom et la signification de chaque paramètre. Si pour une raison ou pour une autre, vous avez à vous référer aux paramètres d'un message en utilisant l'ancienne convention d'appellation (*WParam*, *LParam*, etc.), vous devez transtyper *Message* vers le type générique *TMessage*, qui utilise ces noms de paramètres.

Interception des messages

Dans certaines circonstances, vous pouvez souhaiter que certains messages soient ignorés par vos composants. Autrement dit, vous voulez empêcher le composant de répartir un message à son gestionnaire. Pour intercepter un message de cette façon, vous devez surcharger la méthode virtuelle *WndProc*.

Pour les composants VCL, la méthode *WndProc* sélectionne les messages avant de les transmettre à la méthode *Dispatch* qui, à son tour, détermine la méthode qui gère le message. En surchargeant *WndProc*, votre composant a la possibilité de filtrer certains messages avant qu'ils ne soient transmis. La surcharge de *WndProc* pour un contrôle dérivé de *TWinControl* ressemble à ceci :

```
procedure TMyControl.WndProc(var Message: TMessage);
begin
  { test pour déterminer la poursuite du traitement }
  inherited WndProc(Message);
end;
```

Le composant *TControl* définit des plages entières de messages liés à la souris qu'il filtre lorsqu'un utilisateur fait glisser puis lâche un contrôle. Une méthode *WndProc* surchargée peut agir par deux moyens :

- Elle peut filtrer des plages entières de messages au lieu de spécifier un gestionnaire pour chacun d'eux.
- Elle peut inhiber totalement la répartition des messages de façon à ce que les gestionnaires ne soient jamais appelés.

Pour la CLX, un contrôle peut être dérivé de *TWidgetControl* et vous pourriez redéfinir *EventFilter* au lieu de *WndProc*.

Voici une partie de la méthode *WndProc* pour *TControl*, par exemple :

```
procedure TControl.WndProc(var Message: TMessage);
begin
  ⋮
  if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
    if Dragging then                          { gestion spécifique d'une opération glisser }
      DragMouseMsg(TWMMouse(Message))
    else
      ⋮                                            { gestion normale des autres opérations }
    end;
⋮                                                              { autrement gestion normale }
end;
```

Création de nouveaux gestionnaires de messages

Puisque Delphi fournit des gestionnaires pour la plupart des messages standard, vous avez à définir de nouveaux gestionnaires de message uniquement lorsque vous définissez vous-mêmes vos propres messages. Pour travailler avec des messages définis par l'utilisateur, nous allons étudier les deux points suivants :

- Définition de vos propres messages
- Déclaration d'une nouvelle méthode de gestion d'un message

Les composants CLX ne gèrent pas les messages Windows mais vous pouvez créer des gestionnaires de messages pour vos propres messages. Notez que vous ne pouvez pas créer des gestionnaires de messages pour les événements Qt car ce sont des objets et non des identificateurs de messages.

Définition de vos propres messages

De nombreux composants standard définissent des messages pour leur usage interne. Définir des messages peut servir à émettre des informations qui ne sont pas prises en compte par les messages standard ou à notifier un changement d'état. Vous pouvez définir vos propres messages dans la VCL et la CLX.

La définition d'un message est un processus à deux étapes :

1 Déclaration d'un identificateur de message
2 Déclaration d'un type enregistrement de message

Déclaration d'un identificateur de message

Un identificateur de message est une constante de la taille d'un entier. Windows se réserve pour son propre usage les messages dont le numéro est inférieur à 1 024. Lorsque vous déclarez vos propres messages, vous devez donc toujours débuter par un numéro supérieur.

La constante *WM_APP* représente le numéro de départ pour les messages définis par l'utilisateur. Lorsque vous définissez un identificateur de message, utilisez *WM_APP*.

Notez que certains contrôles Windows standard utilisent des messages compris dans la plage des messages utilisateur. Entre autres contrôles, il y a les boîtes liste, les boîtes à options, les boîtes de saisie et les boutons de commande. Si vous dérivez un composant à partir de l'un d'eux et si vous voulez lui associer un nouveau message, vérifiez le contenu du l'unité Messages pour voir quels messages Windows sont déjà définis pour ce contrôle.

Le code suivant définit deux messages utilisateur :

```
const
  WM_MYFIRSTMESSAGE = WM_APP + 400;
  WM_MYSECONDMESSAGE = WM_APP + 401;
```

Déclaration d'un type enregistrement de message

Si vous voulez attribuer un nom explicite aux paramètres de votre message, vous devez déclarer un type enregistrement pour le message. L'enregistrement de message correspond au type du paramètre transmis à la méthode de gestion du message. Si vous n'utilisez pas les paramètres du message ou si vous souhaitez utiliser l'ancienne notation (*wParam*, *lParam*, etc.), utilisez l'enregistrement de message implicite, *TMessage*.

Pour déclarer un type enregistrement de message, respectez les conventions suivantes :

1 Nommez le type enregistrement d'après le message en rajoutant le préfixe *T* à son nom.

2 Donnez au premier champ de l'enregistrement *le nom Msg* et le type *TMsgParam*.

3 Définissez les deux octets suivants pour qu'ils correspondent au paramètre *Word*, et les deux suivants inutilisés.

ou

Définissez les quatre octets suivants pour qu'ils correspondent au paramètre *Longint*.

4 Ajoutez un dernier champintitulé *Result* de type *Longint*.

Par exemple, voici un enregistrement de message pour tous les messages de souris, *TWMMouse*, qui utilisent un enregistrement variant pour définir deux ensembles de noms pour les mêmes paramètres.

```
type
  TWMMouse = record
    Msg: TMsgParam;          ( en premier, l'identificateur du message )
    Keys: Word;              ( voici wParam )
    case Integer of          ( deux manières de voir lParam )
      0: (
         XPos: Integer;      ( soit les coordonnées x et y... )
         YPos: Integer);
      1: (
         Pos: TPoint;        ( ... soit un simple point )
         Result: Longint);   ( et enfin, la donnée membre résultat )
end;
```

Déclaration d'une nouvelle méthode de gestion d'un message

Deux circonstances vous amènent à déclarer de nouvelles méthodes de gestion des messages :

- Votre composant a besoin de gérer un message Windows qui n'est pas pris en compte par les composants standard.
- Vous avez défini votre propre message et vous souhaitez l'utiliser avec vos composants.

Pour déclarer la méthode de gestion d'un message, suivez les étapes ci-après :

1 Déclarez la méthode dans une partie **protected** de la déclaration de la classe du composant.

2 Faites de la méthode une procédure.

3 Nommez la méthode suivant le message qu'elle gère en supprimant les caractères de soulignement de son nom.

4 Transmettez un seul paramètre **var** appelé *Message*, du type défini par l'enregistrement du message.

5 A l'intérieur de l'implémentation de la méthode message, écrivez le code de gestion spécifique au composant.

6 Appelez le gestionnaire de message transmis en héritage.

Voici la déclaration, par exemple, d'un gestionnaire de message pour un message utilisateur intitulé *CM_CHANGECOLOR*.

```
const
  CM_CHANGECOLOR = WM_APP + 400;

type
  TMyComponent = class(TControl)
  ⋮
protected
  procedure CMChangeColor(var Message: TMessage); message CM_CHANGECOLOR;
end;

procedure TMyComponent.CMChangeColor(var Message: TMessage);
begin
  Color := Message.lParam;
  inherited;
end;
```

Chapitre

47

Accessibilité des composants au moment de la conception

Ce chapitre décrit les étapes permettant de rendre les composants que vous créez accessibles dans l'EDI. Rendre vos composants accessibles à la conception est un processus qui nécessite plusieurs étapes :

- Recensement des composants
- Ajout de bitmaps à la palette
- Fournir l'aide pour vos composants
- Ajout d'éditeurs de propriétés
- Ajout d'éditeurs de composants
- Compilation des composants en paquets

Toutes les étapes ne s'appliquent pas à tous les composants. Par exemple, si vous ne définissez ni propriété ni événement nouveau, il n'est pas nécessaire de fournir de l'aide. Le recensement et la compilation sont les seules étapes obligatoires.

Une fois que vos composants ont été recensés et compilés en paquets, ils peuvent être distribués à d'autres développeurs et installés dans l'EDI. Pour plus d'informations sur l'installation des paquets dans l'EDI, voir "Installation de paquets de composants" à la page 11-6.

Recensement des composants

Le recensement fonctionne en ayant l'unité de compilation comme base. Si vous créez plusieurs composants dans la même unité de compilation, ils seront tous recensés en même temps.

Pour recenser un composant, ajoutez une procédure *Register* àl'unité. Dans la procédure *Register*, vous recensez les composants et déterminez leur emplacement sur la palette des composants.

Remarque Si vous créez votre composant en choisissant Composant | Nouveau Composant dans l'EDI, le code requis pour recenser votre composant est automatiquement ajouté.

Les étapes d'un recensement manuel de composant sont :

- Déclaration de la procédure Register
- Ecriture de la procédure Register

Déclaration de la procédure Register

Le recensement implique l'écriture d'une procédure unique dans l'unité, qui doit être nommée *Register*. La procédure *Register* doit apparaître dans la partie interface de l'unité et, contrairement au reste du Pascal Objet, son nom tient compte des différences majuscules/minuscules.

Le code suivant montre la présentation d'une seule unité qui crée et recense des nouveaux composants :

```
unit MyBtns;
interface
type
  ...                                                { déclarez vos composants ici }
procedure Register;                    { ceci doit apparaître dans la section interface }
implementation
  ...                                        { l'implémentation de composant vient ici }
procedure Register;
begin
  ...                                                { recense les composants }
end;
end.
```

Dans la procédure *Register*, appelez *RegisterComponents* pour chaque composant que vous souhaitez ajouter à la palette des composants. Si l'unité contient plusieurs composants, vous pouvez les recenser en une seule fois.

Ecriture de la procédure Register

Dans la procédure *Register* d'une unité contenant des composants, vous devez recenser chaque composant que vous voulez ajouter à la palette des composants. Si l'unité contient plusieurs composants, vous pouvez les recenser en une seule fois.

Pour recenser un composant, appelez la procédure *RegisterComponents* pour chacune des pages de la palette auxquelles vous souhaitez ajouter des composants. *RegisterComponents* nécessite trois informations importantes :

1 Spécification des composants
2 Spécification de la page de palette
3 Utilisation de la fonction RegisterComponents

Spécification des composants

Dans la procédure Register, transmettez les noms de composant dans un tableau ouvert, que vous pouvez construire dans l'appel à RegisterComponents.

```
RegisterComponents('Miscellaneous', [TMyComponent]);
```

Vous pourriez aussi recenser plusieurs composants en même temps sur la même page ou recenser des composants sur des pages différentes, comme dans le code suivant :

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TFirst, TSecond]);        { deux sur cette page... }
  RegisterComponents('Assorted', [TThird]);                      { ...un sur une autre... }
  RegisterComponents(LoadStr(srStandard), [TFourth]);     { ...et un sur la page Standard }
end;
```

Spécification de la page de palette

Le nom de la page de palette est une chaîne. Si le nom que vous donnez pour la page de palette n'existe pas, Delphi crée une nouvelle page avec ce nom. Delphi stocke les noms des pages standard dans des ressources liste de chaînes afin que les versions internationales du produit puissent nommer les pages dans leur langue. Si vous voulez installer un composant dans l'une des pages standard, vous obtiendrez la chaîne du nom de cette page en appelant la fonction *LoadStr* et en transmettant la constante représentant la ressource chaîne de cette page (par exemple *srSystem* pour la page Système).

Utilisation de la fonction RegisterComponents

Dans la procédure *Register*, appelez *RegisterComponents* pour recenser les composants dans le tableau des classes. *RegisterComponents* est une fonction qui présente deux paramètres : le nom de page de palette des composants et le tableau des classes de composants.

Mettez le nom de la page sur la palette des composants, où les composants doivent apparaître, dans le paramètre Page. Si la page nommée existe déjà, les composants sont ajoutés à cette page. Si elle n'existe pas, Delphi crée une nouvelle page palette ayant ce nom.

Appelez RegisterComponents depuis l'implémentation de la procédure Register dans une des unités qui définit les composants personnalisés. Les unités définissant les composants doivent alors être compilées en un paquet et ce dernier doit être installé avant l'ajout des composants personnalisés à la palette des composants.

```
procedure Register;
begin
  RegisterComponents('System', [TSystem1, TSystem2]);            {ajout à la page system }
  RegisterComponents('MyCustomPage',[TCustom1, TCustom2]);              { nouvelle page }
end;
```

Ajout de bitmaps à la palette

Chaque composant fait appel à un bitmap pour le représenter dans la palette. Si vous ne spécifiez pas de bitmap, Delphi utilise un bitmap par défaut.

Puisque les bitmaps d'une palette ne sont nécessaires qu'à la conception, ils ne sont pas compilés à l'intérieur de l'unité du composant. En revanche, ils doivent être fournis dans un fichier de ressources Windows portant le même nom que l'unité, mais avec l'extension .DCR (dynamic component resource). Vous pouvez créer ce fichier de ressources en utilisant un éditeur d'images dans Delphi. Chaque bitmap doit être un carré de 24 pixels de côté.

Pour chaque composant à installer, vous devez fournir un fichier de bitmaps, et dans chaque fichier de bitmaps, un bitmap pour chaque composant recensé. L'image bitmap a le même nom que le composant. Conservez le fichier de bitmaps dans le même répertoire que les fichiers compilés, afin que Delphi puisse localiser ces bitmaps lorsqu'il installe les composants dans la palette.

Par exemple, si vous créez un composant appelé *TMyControl* dans une unité nommée ToolBox, vous devez créer un fichier de ressources appelé TOOLBOX.DCR contenant un bitamp nommé TMYCONTROL. Les noms des ressources ne tiennent pas compte des différences majuscules/minuscules, mais sont en majuscules par convention.

Fournir l'aide pour vos composants

Lorsque vous sélectionnez un composant standard dans une fiche, ou une propriété ou un événement dans l'inspecteur d'objets, vous pouvez appuyer sur *F1* pour obtenir de l'aide concernant cet élément. Les développeurs pourront accéder au même type d'information sur vos composants si vous créez les fichiers d'aide appropriés.

Vous pouvez fournir un fichier d'aide de faible encombrement pour décrire vos composants et votre fichier d'aide devient partie intégrante du système d'aide global de Delphi.

Voir "Création du fichier d'aide" à la page 47-4 pour des informations sur la manière de composer le fichier d'aide à utiliser avec un composant.

Création du fichier d'aide

Vous pouvez utiliser l'outil de votre choix pour créer les fichiers d'aide Windows (au format .rtf). Delphi inclut le Microsoft Help Workshop, qui compile les fichiers d'aide et contient un guide en ligne destiné à l'auteur du système d'aide. Vous y trouverez toutes les informations nécessaires à la création des systèmes d'aide.

La composition de fichiers d'aide pour les composants s'effectue en plusieurs étapes :

- Création des entrées
- Aide contextuelle des composants
- Ajout des fichiers d'aide des composants

Création des entrées

Pour que l'aide associée à votre composant fonctionne de façon transparente avec celle des autres composants de la bibliothèque, vous devez respecter les conventions suivantes :

1 Chaque composant doit avoir une rubrique d'aide**.**

La rubrique associée au composant doit montrer dans quelle unité est déclaré le composant et fournir une brève description du composant. La rubrique du composant doit proposer des liens vers des fenêtres secondaires décrivant la position du composant dans la hiérarchie des objets et répertorier l'ensemble de ses propriétés, événements et méthodes. Les développeurs d'applications accéderont à cette rubrique en sélectionnant le composant dans une fiche et en appuyant sur *F1*. Pour avoir un exemple d'une rubrique associée à un composant, positionnez-vous sur un composant quelconque dans une fiche et appuyez sur *F1*.

La rubrique de composant doit avoir une note de bas de page # avec une valeur unique de rubrique. La note de bas de page # identifie de manière unique chaque rubrique du système d'aide.

La rubrique associée au composant doit avoir une note de bas de page "K" (qui indique les mots clé de recherche) comprenant le nom de la classe du composant. Par exemple, la note de bas de page des mots clés pour le composant *TMemo* contient "TMemo".

La rubrique associée au composant doit également comprendre une note de bas de page $ qui indique le titre de la rubrique. Le titre apparaît dans la boîte de dialogue des rubriques, la boîte de dialogue Signet et la fenêtre Historique.

2 Chaque composant doit inclure les rubriques de navigation secondaires suivantes :

- Une rubrique de hiérarchie offrant des liens vers chaque ancêtre du composant appartenant à sa hiérarchie.
- Une liste de toutes les propriétés disponibles dans le composant, avec des liens vers les entrées décrivant ces propriétés.
- Une liste de tous les événements disponibles dans le composant, avec des liens vers les entrées décrivant ces événements.
- Une liste de toutes les méthodes disponibles dans le composant, avec des liens vers les entrées décrivant ces méthodes.

Les liens vers les classes d'objets, les propriétés ou les événements dans le système d'aide de Delphi peuvent être réalisés à l'aide de Alinks. Alink opère

la liaison vers une classe d'objet en utilisant le nom de classe de l'objet suivi d'un caractère de soulignement et de la chaîne "object". Par exemple, pour réaliser un lien vers l'objet *TCustomPanel*, utilisez le code suivant :

```
!AL(TCustomPanel_object,1)
```

Pour réaliser un lien vers une propriété, une méthode ou un événement, faites précéder son nom par celui de l'objet qui l'implémente et par un caractère de soulignement. Par exemple, pour réaliser un lien vers la propriété *Text* implémentée par *TControl*, utilisez le code suivant :

```
!AL(TControl_Text,1)
```

Pour voir un exemple de rubriques de navigation secondaires, affichez l'aide d'un composant quelconque et cliquez sur les liens étiquetés hiérarchie, propriétés, méthodes ou événements.

3 Chaque propriété, événement ou méthode déclaré à l'intérieur du composant doit avoir une rubrique**.**

Une rubrique décrivant une propriété, un événement ou une méthode doit indiquer la déclaration de l'élément et décrire son rôle. Les développeurs d'applications accéderont à cette rubrique en sélectionnant l'élément dans l'inspecteur d'objets et en appuyant sur *F1*, ou en plaçant le curseur dans l'éditeur de code sur le nom de l'élément et en appuyant sur *F1*. Pour avoir un exemple de rubrique associée à une propriété, sélectionnez un élément quelconque dans l'inspecteur d'objets et appuyez sur *F1*.

Les rubriques de propriété, d'événement et de méthode doivent inclure une note de bas de page K qui indique le nom de la propriété, de l'événement et de la méthode, son nom et celui du composant. Ainsi, la propriété *Text* de *TControl* présente la note de bas de page K suivante :

```
Text,TControl;TControl,Text;Text,
```

Les rubriques de propriété, de méthode ou d'événement doivent également comporter une note de bas de page $ indiquant le titre de la rubrique, tel que TControl.Text.

Toutes ces rubriques doivent disposer d'un identificateur de rubrique unique à la rubrique, entré sous la forme d'une note de bas de page #.

Aide contextuelle des composants

Chaque rubrique de composant, de propriété, de méthode et d'événement doit avoir une note de bas de page A. La note de bas de page A permet d'afficher la rubrique lorsque l'utilisateur sélectionne un composant et appuie sur *F1*, ou lorsqu'il sélectionne une propriété ou un événement dans l'inspecteur d'objets et appuie sur *F1*. Les notes de bas de page A doivent suivre certaines conventions d'appellation :

Si la rubrique d'aide est destinée à un composant, la note de bas de page A comprend deux entrées séparées par un point-virgule selon la syntaxe suivante :

```
ComponentClass_Object;ComponentClass
```

où ComponentClass est le nom de la classe du composant.

Si la rubrique d'aide est destinée à une propriété ou à un événement, la note de bas de page A comprend trois entrées séparées par des points-virgules selon la syntaxe suivante :

```
ComponentClass_Element;Element_Type;Element
```

où *ComponentClass* est le nom de la classe du composant, *Element* est le nom de la propriété, de la méthode ou de l'événement et *Type* est la propriété, la méthode ou l'événement

Par exemple, en supposant une propriété BackgroundColor d'un composant TMyGrid, la note de bas de page A est

```
TMyGrid_BackgroundColor;BackgroundColor_Property;BackgroundColor
```

Ajout des fichiers d'aide des composants

Pour ajouter votre fichier d'aide à Delphi, utilisez l'utilitaire OpenHelp (appelé oh.exe) situé dans le répertoire bin ou en utilisant Aide | Personnaliser dans l'EDI.

Vous obtiendrez des informations sur le fichier OpenHelp.hlp sur l'utilisation de OpenHelp, ainsi que sur l'ajout de votre fichier d'aide au système d'aide.

Ajout d'éditeurs de propriétés

L'inspecteur d'objets permet par défaut de modifier tous les types de propriétés. Vous pouvez toutefois créer un éditeur pour des propriétés spécifiques en l'écrivant et en le recensant. Vous pouvez recenser les éditeurs de propriétés afin qu'ils s'appliquent uniquement aux propriétés des composants dont vous êtes l'auteur, ou à toutes les propriétés du type spécifié.

A la base, un éditeur de propriétés peut opérer selon deux modes : affichage sous la forme d'une chaîne texte permettant à l'utilisateur la modification de la valeur courante ; affichage d'une boîte de dialogue permettant des modifications d'une autre sorte. Selon la propriété en cours de modification, vous pourrez faire appel à l'un ou l'autre mode.

L'écriture d'un éditeur de propriété se déroule en cinq étapes :

1. Dérivation d'une classe éditeur de propriétés
2. Modification de la propriété sous une forme textuelle
3. Modification globale de la propriété
4. Spécification des attributs de l'éditeur
5. Recensement de l'éditeur de propriétés

Dérivation d'une classe éditeur de propriétés

La VCL et la CLX définissent plusieurs sortes d'éditeurs de propriétés, tous descendant de *TPropertyEditor*. Lorsque vous créez un éditeur de propriétés, votre classe éditeur de propriétés peut descendre directement de *TPropertyEditor* ou d'un des types d'éditeurs de propriétés décrits dans le tableau suivant. Les classes de l'unité DesignEditors peuvent être utilisées dans les applications VCL

et CLX. Néanmoins, certaines classes éditeur de propriété fournissent des boîtes de dialogue spécialisées et sont ainsi spécialisées VCL ou CLX. Elles peuvent être trouvées respectivement dans les unités WinEditors et CLXEditors.

Remarque Ce qui est absolument nécessaire pour un éditeur de propriété est qu'il dérive de *TBasePropertyEditor* et qu'il prenne en charge l'interface *IProperty*. Néanmoins, *TPropertyEditor* fournit une implémentation par défaut de l'interface *IProperty*.

La liste du tableau suivant n'est pas complète. Les unités WinEditors et CLXEditors définissent également certains éditeurs spécialisés utilisés exclusivement par certaines propriétés comme le composant *Name*. Les éditeurs de propriétés ci-dessous sont les plus utiles aux concepteurs de propriétés définies par l'utilisateur.

Tableau 47.1 Types d'éditeurs de propriétés prédéfinis

Type	Propriétés modifiables
TOrdinalProperty	Tous les éditeurs de valeurs ordinales (propriétés de type entier, caractères, énuméré) sont des descendants de *TOrdinalProperty*.
TIntegerProperty	Tous les types entiers y compris ceux prédéfinis ainsi que les intervalles utilisateur.
TCharProperty	Le type *Char* et les intervalles de valeurs *Char* tels que 'A'..'Z'.
TEnumProperty	Tous les types énumérés.
TFloatProperty	Tous les nombres à virgule flottante.
TStringProperty	Toutes les chaînes.
TSetElementProperty	Les éléments des ensembles comme valeurs booléennes.
TSetProperty	Tous les ensembles. Les ensembles ne sont pas directement modifiables mais peuvent être développés sous la forme d'une liste de propriétés que sont les éléments de l'ensemble.
TClassProperty	Classes. Affiche le nom de la classe et se développe pour afficher les propriétés de la classe.
TMethodProperty	Pointeurs sur des méthodes, le plus souvent des événements.
TComponentProperty	Les composants de la même fiche. Ne permet pas la modification des propriétés des composants, mais peut pointer sur un composant spécifique de type compatible.
TColorProperty	Les couleurs d'un composant. Montre si possible les constantes de couleurs ou à défaut leurs valeurs en hexadécimal. Une liste déroulante affiche les constantes de couleurs. Un double-clic a pour effet d'ouvrir la boîte de dialogue de sélection des couleurs.
TFontNameProperty	Les noms de fontes. La liste déroulante affiche toutes les fontes actuellement installées.
TFontProperty	Les fontes. Autorise le développement des propriétés d'une fonte particulière et offre l'accès à la boîte de dialogue des fontes.

L'exemple suivant montre la déclaration d'un éditeur de propriétés simple nommé *TMyPropertyEditor* :

```
type
  TFloatProperty = class(TPropertyEditor)
  public
    function AllEqual: Boolean; override;
```

```
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
  end;
```

Modification de la propriété sous une forme textuelle

Toutes les propriétés doivent fournir une représentation de type chaîne de leurs valeurs en vue de leur affichage dans l'inspecteur d'objets. Dans le cas de la plupart des propriétés, le développeur pourra saisir une nouvelle valeur lors de la conception. Les classes éditeur de propriétés fournissent des méthodes virtuelles que vous pouvez surcharger afin de convertir le texte de la propriété en sa valeur réelle.

Les méthodes à surcharger sont *GetValue* et *SetValue*. Votre éditeur de propriétés hérite également d'un ensemble de méthodes utilisées pour affecter et lire différentes sortes de valeurs comme le montre le tableau suivant.

Tableau 47.2 Méthodes pour lire et écrire les valeurs des propriétés

Type de propriété	Méthode Get	Méthode Set
Virgule flottante	GetFloatValue	SetFloatValue
Pointeur de méthode (événement)	GetMethodValue	SetMethodValue
Type ordinal	GetOrdValue	SetOrdValue
Chaîne	GetStrValue	SetStrValue

Lorsque vous surchargez une méthode *GetValue*, appelez l'une des méthodes "Get". Lorsque vous surchargez *SetValue*, appelez l'une des méthodes "Set".

Affichage de la valeur de la propriété

La méthode *GetValue* de l'éditeur de propriétés renvoie une chaîne représentant la valeur en cours de la propriété. L'inspecteur d'objets utilise cette chaîne dans la colonne des valeurs pour cette propriété. Par défaut, *GetValue* renvoie "*inconnu*".

Pour fournir une représentation sous une forme chaîne, vous devez surcharger la méthode *GetValue* de l'éditeur de propriétés.

Si la propriété n'est pas une valeur chaîne, votre méthode *GetValue* doit convertir la valeur en une chaîne.

Définition de la valeur de la propriété

La méthode *SetValue* de l'éditeur de propriétés accepte la chaîne saisie dans l'inspecteur d'objets, la convertit dans le type approprié, et définit la propriété. Si la chaîne n'est pas une valeur convenant à la propriété, *SetValue* doit déclencher et ignorer la valeur.

Pour lire la valeur d'une propriété, vous devez surcharger la méthode *SetValue* de l'éditeur de propriétés.

SetValue doit convertir la chaîne et la valider avant d'appeler une des méthodes.

Voici les méthodes *GetValue* et *SetValue* de *TIntegerProperty*. *Integer* est de type ordinal, ainsi *GetValue* appelle *GetOrdValue* et convertit le résultat en chaîne. *SetValue* convertit la chaîne en entier, effectue certains calculs d'intervalle et appelle *SetOrdValue*.

```
function TIntegerProperty.GetValue: string;
begin
  with GetTypeData(GetPropType)^ do
    if OrdType = otULong then // non signé
      Result := IntToStr(Cardinal(GetOrdValue))
    else
      Result := IntToStr(GetOrdValue);
end;

procedure TIntegerProperty.SetValue(const Value: string);
  procedure Error(const Args: array of const);
  begin
    raise EPropertyError.CreateResFmt(@SOutOfRange, Args);
  end;
var
  L: Int64;
begin
  L := StrToInt64(Value);
  with GetTypeData(GetPropType)^ do
    if OrdType = otULong then
    begin
      if (L < Cardinal(MinValue)) or (L > Cardinal(MaxValue)) then
        Error([Int64(Cardinal(MinValue)), Int64(Cardinal(MaxValue))]);
    end
    else if (L < MinValue) or (L > MaxValue) then
      Error([MinValue, MaxValue]);
  SetOrdValue(L);
end;
```

Les spécificités des exemples particuliers sont moins importantes qu'en principe : *GetValue* convertit la valeur en chaîne ; *SetValue* convertit la chaîne et valide la valeur avant d'appeler une des méthodes "Set".

Modification globale de la propriété

Si vous le souhaitez, vous pouvez fournir une boîte de dialogue pour la définition de la propriété. L'utilisation la plus courante des éditeurs de propriétés concerne les propriétés qui sont des classes. Un exemple est la propriété *Font*, qui a une boîte de dialogue éditeur associée permettant au développeur de choisir tous les attributs de fonte en même temps.

Pour fournir une boîte de dialogue de définition globale de la propriété, surchargez la méthode *Edit* de la classe éditeur de propriétés.

Les méthodes Edit utilisent les mêmes méthodes "Get" et "Set" utilisées dans les méthodes *GetValue* et *SetValue* ; une méthode *Edit* appelle à la fois une méthode "Get" et une méthode "Set". Comme l'éditeur est spécifique du type, il est

habituellement inutile de convertir les valeurs des propriétés en chaînes. L'éditeur traite généralement la valeur telle qu'elle a été récupérée.

Lorsque l'utilisateur clique sur le bouton '...' à côté de la propriété, ou double-clique sur la colonne des valeurs, l'inspecteur d'objets appelle la méthode *Edit* de l'éditeur de propriétés.

Pour votre implémentation de la méthode *Edit*, suivez ces étapes :

1. Construisez l'éditeur que vous utilisez pour cette propriété.
2. Lisez la valeur en cours et attribuez-la à la propriété en utilisant une méthode "Get".
3. Lorsque l'utilisateur sélectionne une nouvelle valeur, attribuez cette valeur à la propriété en utilisant une méthode "Set".
4. Détruisez l'éditeur.

Les propriétés *Color* trouvées dans la plupart des composants utilisent la boîte de dialogue de couleur standard Windows comme éditeur de propriétés. Voici la méthode *Edit* issue de *TColorProperty*, qui appelle la boîte de dialogue et utilise le résultat :

```
procedure TColorProperty.Edit;
var
  ColorDialog: TColorDialog;
begin
  ColorDialog := TColorDialog.Create(Application);                { construit l'éditeur }
  try
    ColorDialog.Color := GetOrdValue;                       { utilise la valeur existante }
    if ColorDialog.Execute then  { si l'utilisateur valide la boîte de dialogue par OK... }
      SetOrdValue(ColorDialog.Color);     { ...utilise le résultat pour définir la valeur }
  finally
    ColorDialog.Free;                                             { détruit l'éditeur }
  end;
end;
```

Spécification des attributs de l'éditeur

L'éditeur de propriétés doit fournir les informations permettant à l'inspecteur d'objets de déterminer les outils à afficher. Par exemple, l'inspecteur a besoin de savoir si la propriété a des sous-propriétés, ou s'il doit afficher la liste des valeurs possibles de la propriété.

Pour spécifier les attributs de l'éditeur, vous devez surcharger sa méthode *GetAttributes*.

GetAttributes renvoie un ensemble de valeurs de type *TPropertyAttributes* qui peut inclure une ou plusieurs des valeurs suivantes :

Tableau 47.3 Indicateurs des attributs des éditeurs de propriétés

Indicateur	Méthode associée	Signification si inclus
paValueList	GetValues	L'éditeur peut fournir une liste de valeurs énumérées.
paSubProperties	GetProperties	La propriété dispose de sous-propriétés qu'il est possible d'afficher.
paDialog	Edit	L'éditeur peut afficher une boîte de dialogue permettant de modifier globalement la propriété.
paMultiSelect	N/D	La propriété doit s'afficher lorsque l'utilisateur sélectionne plusieurs composants.
paAutoUpdate	SetValue	Mise à jour du composant après chaque modification au lieu d'attendre l'approbation de la valeur.
paSortList	N/D	L'inspecteur d'objets doit trier la liste de valeurs.
paReadOnly	N/D	La valeur de la propriété ne peut être modifiée lors de la conception.
paRevertable	N/D	Active l'élément de menu Revenir à hérité dans le menu contextuel de l'inspecteur d'objets. Cet élément de menu demande à l'éditeur d'annuler la valeur en cours de la propriété et de revenir à une valeur par défaut ou standard préalablement établie.
paFullWidthName	N/D	La valeur n'a pas besoin d'être affichée. L'inspecteur d'objets utilise toute sa largeur pour le nom de propriété.
paVolatileSubProperties	GetProperties	L'inspecteur d'objets récupère les valeurs de toutes les sous-propriétés à chaque modification de la valeur de la propriété.
paReference	GetComponentValue	La valeur est une référence à quelque chose d'autre. Utilisé conjointement avec paSubProperties, l'objet référencé devrait être affiché comme sous-propriétés de cette propriété.

Les propriétés *Color* sont plus polyvalentes que la plupart des autres propriétés, l'utilisateur dispose de plusieurs moyens pour sélectionner une couleur dans l'inspecteur d'objets : il peut taper une valeur, sélectionner dans une liste ou faire appel à l'éditeur personnalisé. C'est pourquoi l*a méthode GetAttributes* de *TColorProperty*, inclut plusieurs attributs dans la valeur qu'elle renvoie :

```
function TColorProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paMultiSelect, paDialog, paValueList, paRevertable];
end;
```

Recensement de l'éditeur de propriétés

Lorsque l'éditeur de propriétés est créé, vous devez le recenser dans Delphi. Le recensement d'un éditeur de propriétés associe un type de propriété et un éditeur spécifique. Vous pouvez recenser un éditeur pour toutes les propriétés d'un type particulier ou juste pour une propriété particulière d'un type de composant particulier.

Pour recenser un éditeur de propriétés, appelez une procédure *RegisterPropertyEditor*.

RegisterPropertyEditor prend quatre paramètres :

- Un pointeur de type information décrivant le type de la propriété à modifier.

  Il s'agit toujours d'un appel à la fonction intégrée *TypeInfo*, telle que `TypeInfo(TMyComponent)`.
- Le type du composant auquel s'applique cet éditeur. Si ce paramètre est **nil**, l'éditeur s'applique à toutes les propriétés d'un type donné.
- Le nom de la propriété. Ce paramètre n'est significatif que si le paramètre qui le précède spécifie un type particulier de composant. Dans ce cas, vous pouvez spécifier une propriété de ce type auquel s'applique l'éditeur.
- Le type d'éditeur de propriétés à utiliser pour modifier la propriété spécifiée.

Voici un extrait de la procédure qui recense les éditeurs des composants standard inclus dans la palette :

```
procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TComponent), nil, '', TComponentProperty);
  RegisterPropertyEditor(TypeInfo(TComponentName), TComponent, 'Name',
    TComponentNameProperty);
  RegisterPropertyEditor(TypeInfo(TMenuItem), TMenu, '', TMenuItemProperty);
end;
```

Les trois instructions de cette procédure couvrent les différentes utilisations de *RegisterPropertyEditor* :

- La première instruction est la plus typique. Elle recense l'éditeur de propriétés *TComponentProperty* pour toutes les propriétés de type *TComponent* (ou les descendants de *TComponent* qui n'ont pas d'éditeur spécifique recensé). Habituellement, vous créez un éditeur s'appliquant à un type particulier, puis vous souhaitez l'utiliser pour l'ensemble des propriétés de ce type. C'est pourquoi le deuxième et le troisième paramètres ont pour valeurs respectives **nil** et une chaîne vide.
- La deuxième instruction est le type de recensement le plus spécifique. Elle recense un éditeur pour une propriété spécifique et pour un type spécifique de composant. Dans notre exemple, l'éditeur s'applique à la propriété *Name* (de type *TComponentName*) de tous les composants.

- La troisième instruction est plus spécifique que la première, et moins que la deuxième. Elle recense un éditeur pour toutes les propriétés de type *TMenuItem* pour les composants de type *TMenu*.

Catégories de propriété

Dans l'IDE, l'inspecteur d'objets vous permet de masquer et d'afficher sélectivement des propriétés basées sur les catégories de propriété. Les propriétés des nouveaux composants personnalisés peuvent rentrer dans ce schéma en recensant des propriétés par catégories. Faites ceci lors du recensement du composant en appelant *RegisterPropertyInCategory* ou *RegisterPropertiesInCategory*. Utilisez *RegisterPropertyInCategory* pour recenser une seule propriété. Utilisez *RegisterPropertiesInCategory* pour recenser plusieurs propriétés dans un seul appel de fonction. Ces fonctions sont définies dans l'unité DesignIntf.

Notez qu'il n'est pas obligatoire de recenser des propriétés ni que toutes les propriétés d'un composant personnalisé soient recensées lorsque quelques-unes le sont. Toute propriété non explicitement associée à une catégorie est incluse dans la catégorie *TMiscellaneousCategory*. De telles propriétés sont affichées ou masquées dans l'inspecteur d'objets selon cette catégorisation par défaut.

En plus de ces deux fonctions de recensement de propriétés, il existe une fonction IsPropertyInCategory. Cette fonction est utile pour la création d'utilitaires de localisation, dans laquelle vous devez déterminer si une propriété est recensée dans une catégorie de propriété donnée.

Recensement d'une propriété à la fois

Vous pouvez recenser une propriété à la fois et l'associer à une catégorie de propriété en utilisant la fonction *RegisterPropertyInCategory*. *RegisterPropertyInCategory* est fournie dans quatre variations surchargées, chacune proposant un ensemble différent de critères pour l'identification de la propriété dans le composant personnalisé associé à la catégorie de propriété.

La première variation vous permet d'identifier la propriété selon son nom. La ligne ci-après recense une propriété associée à l'affichage visuel du composant, en identifiant la propriété par son nom, "AutoSize".

```
RegisterPropertyInCategory('Visual', 'AutoSize');
```

La deuxième variation identifie la propriété en utilisant le type de classe de composant et le nom de propriété caractéristiques. L'exemple ci-après recense (dans la catégorie THelpCategory) une propriété appelée "HelpContext" d'un composant de la classe personnalisée TMyButton.

```
RegisterPropertyInCategory('Help and Hints', TMyButton, 'HelpContext');
```

La troisième variation identifie la propriété en utilisant son type au lieu de son nom. L'exemple ci-après recense une propriété basée sur son type, Integer.

```
RegisterPropertyInCategory('Visual', TypeInfo(Integer));
```

La dernière variation utilise à la fois le type de la propriété et son nom pour identifier la propriété. L'exemple ci-après recense une propriété basée sur une combinaison de son type, *TBitmap* et de son nom, "Pattern".

```
RegisterPropertyInCategory('Visual', TypeInfo(TBitmap), 'Pattern');
```

Consultez la section Spécification de catégories de propriétés pour une liste des catégories de propriétés disponibles et une brève description de leur utilisation.

Recensement de plusieurs propriétés en une seule fois

Vous pouvez recenser plusieurs propriétés en une seule fois et les associer à une catégorie de propriété en utilisant la fonction RegisterPropertiesInCategory. RegisterPropertiesInCategory est fournie dans trois variations surchargées, chacune proposant un ensemble différent de critères pour l'identification de la propriété dans le composant personnalisé associé à la catégorie de propriété.

La première variation vous permet d'identifier des propriétés en fonction du nom ou du type de propriété. La liste est transmise sous la forme d'un tableau de constantes. Dans l'exemple ci-après, toute propriété ayant pour nom "Text" ou qui appartient à une classe de type *TEdit* est recensée dans la catégorie 'Localizable'.

```
RegisterPropertiesInCategory('Localizable', ['Text', TEdit]);
```

La deuxième variation vous permet de limiter les propriétés recensées à celles qui appartiennent à un composant spécifique. La liste des propriétés à recenser comprend seulement les noms, pas les types. Par exemple, le code suivant recense un nombre de propriétés dans la catégorie 'Help and Hints' pour tous les composants :

```
RegisterPropertiesInCategory('Help and Hints', TComponent, ['HelpContext', 'Hint',
'ParentShowHint', 'ShowHint']);
```

La troisième variation vous permet de limiter les propriétés recensées à celles possédant un type spécifique. Comme avec la seconde variation, la liste des propriétés à recenser peut n'inclure que les noms :

```
RegisterPropertiesInCategory('Localizable', TypeInfo(String), ['Text', 'Caption']);
```

Consultez la section Spécification de catégories de propriétés pour obtenir une liste des catégories de propriété disponibles ainsi qu'une brève description de leur utilisation.

Spécification de catégories de propriétés

Lorsque vous recensez les propriétés dans une catégorie, vous pouvez utiliser la chaîne de votre choix pour le nom de la catégorie. Si vous utilisez une chaîne qui n'a pas encore été utilisée, l'inspecteur d'objets génère une nouvelle classe de catégorie de propriétés avec ce nom. Vous pouvez néanmoins recenser des

propriétés dans des catégories intégrées. Les catégories de propriétés intégrées sont décrites dans le tableau suivant :

Tableau 47.4 Catégories de propriétés

Catégorie	Usage
Action	Propriétés relatives aux actions d'exécution ; les propriétés *Enabled* et *Hint* de *TEdit* se trouvent dans cette catégorie.
Database	Propriétés relatives aux opérations de bases de données ; les propriétés *DatabaseName* et *SQL* de *TQuery* se trouvent dans cette catégorie.
Drag, Drop, and Docking	Propriétés relatives aux opérations de glisser-déplacer et d'ancrage ; les propriétés *DragCursor* et *DragKind* de *TImage* se trouvent dans cette catégorie.
Help and Hints	Propriétés relatives à l'utilisation de l'aide en ligne ou des conseils ; les propriétés *HelpContext* et *Hint* de *TMemo* se trouvent dans cette catégorie.
Layout	Propriétés relatives à l'affichage visuel d'un contrôle à la conception ; les propriétés *Top* et *Left* de *TDBEdit* se trouvent dans cette catégorie.
Legacy	Propriétés relatives aux opérations obsolètes ; les propriétés *Ctl3D* et *ParentCtl3D* de *TComboBox* se trouvent dans cette catégorie.
Linkage	Propriétés relatives à l'association ou à la liaison d'un composant à un autre ; la propriété *DataSet* de *TDataSource* se trouve dans cette catégorie.
Locale	Propriétés relatives aux localisations internationales ; les propriétés *BiDiMode* et *ParentBiDiMode* de *TMainMenu* se trouvent dans cette catégorie.
Localizable	Propriétés qui peuvent nécessiter des modifications dans les versions localisées d'une application. De nombreuses propriétés chaîne (comme *Caption*) font partie de cette catégorie ainsi que les propriétés qui déterminent la taille et la position des contrôles.
Visual	Propriétés relatives à l'affichage visuel d'un contrôle à l'exécution ; les propriétés *Align* et *Visible* de *TScrollBox* se trouvent dans cette catégorie.
Input	Propriétés relatives à la saisie de données (il n'est pas nécessaire qu'elles soient relatives aux opérations de bases de données) ; les propriétés *Enabled* et *ReadOnly* de *TEdit* se trouvent dans cette catégorie.
Miscellaneous	Propriétés qui ne rentrent pas dans une catégorie ou n'ont pas besoin d'être mises dans des catégories (et les propriétés non explicitement recensées dans une catégorie spécifique) ; les propriétés *AllowAllUp* et *Name* de *TSpeedButton* se trouvent dans cette catégorie.

Utilisation de la fonction IsPropertyInCategory

Une application peut rechercher les propriétés recensées existantes afin de déterminer si une propriété donnée est toujours recensée dans une catégorie indiquée. Ceci peut être particulièrement utile dans des situations telles qu'un utilitaire de localisation qui vérifie la catégorisation des propriétés afin de préparer ses opérations de localisation. Deux variations surchargées de la

fonction IsPropertyInCategory sont disponibles, autorisant différents critères afin de déterminer si une propriété se trouve dans une catégorie.

La première variation vous permet de baser le critère de comparaison sur une combinaison du type de classe du composant propriétaire et du nom de la propriété. Dans la ligne de commande ci-après, pour que IsPropertyInCategory renvoie True, la propriété doit appartenir à un descendant *de TCustomEdit*, avoir le nom "Text" et se trouver dans la catégorie de propriété `'Localizable'`.

```
IsItThere := IsPropertyInCategory('Localizable', TCustomEdit, 'Text');
```

La deuxième variation vous permet de baser le critère de comparaison sur une combinaison du nom de classe du composant propriétaire et du nom de la propriété. Dans la ligne de commande ci-après, pour que IsPropertyInCategory renvoie True, la propriété doit être un descendant de *TCustomEdit*, avoir le nom"Text", et se trouver dans la catégorie de propriété `'Localizable'`.

```
IsItThere := IsPropertyInCategory('Localizable', 'TCustomEdit', 'Text');
```

Ajout d'éditeurs de composants

Les éditeurs de composants déterminent ce qui se passe lorsque vous double-cliquez sur le composant dans le concepteur et ajoutent des commandes au menu contextuel qui apparaît lorsque vous cliquez sur le composant avec le bouton droit. Ils peuvent également copier votre composant dans le Presse-papiers Windows dans des formats personnalisés.

Si vous n'attribuez pas d'éditeur à vos composants, Delphi utilise l'éditeur de composants par défaut. Ce dernier est implémenté par la classe *TDefaultEditor*. *TDefaultEditor* n'ajoute aucun nouvel élément au menu contextuel d'un composant. Lorsque vous double-cliquez sur le composant, *TDefaultEditor* recherche ses propriétés et génère le premier gestionnaire d'événement trouvé ou s'y rend.

Pour ajouter des éléments au menu contextuel, modifier le comportement du composant lorsque vous double-cliquez dessus ou ajouter de nouveaux formats de Presse-papiers, dérivez une nouvelle classe à partir de *TComponentEditor* et recensez-la pour qu'elle soit utilisée avec votre composant. Dans vos méthodes surchargées, vous pouvez utiliser la propriété *Component* de *TComponentEditor* pour accéder au composant en cours de modification.

L'ajout d'un éditeur de composants personnalisé comprend plusieurs étapes :

- Ajout d'éléments au menu contextuel
- Modification du comportement suite à un double-clic
- Ajout de formats de Presse-papiers
- Recensement d'un éditeur de composants

Ajout d'éléments au menu contextuel

Lorsque l'utilisateur clique avec le bouton droit sur le composant, les méthodes *GetVerbCount* et *GetVerb* de l'éditeur de composants sont appelées pour construire un menu contextuel. Vous pouvez surcharger ces méthodes pour ajouter des commandes (verbes) au menu contextuel.

L'ajout d'éléments au menu contextuel requiert ces étapes :

- Spécification d'éléments de menu
- Implémentation des commandes

Spécification d'éléments de menu

Surchargez la méthode *GetVerbCount* pour renvoyer le nombre de commandes que vous ajoutez au menu contextuel. Surchargez la méthode *GetVerb* pour renvoyer les chaînes qui doivent être ajoutées pour chacune de ces commandes. Lorsque vous surchargez *GetVerb*, ajoutez un "et" commercial (&) dans une chaîne afin que le caractère suivant apparaisse souligné dans le menu contextuel et fasse office de touche de raccourci pour la sélection de l'élément du menu. Veillez à ajouter des points de suspension (...) à la fin d'une commande si elle fait apparaître une boîte de dialogue. *GetVerb* possède un paramètre unique pour indiquer l'index de la commande.

Le code suivant surcharge les méthodes *GetVerbCount* et *GetVerb* pour ajouter deux commandes au menu contextuel.

```
function TMyEditor.GetVerbCount: Integer;
begin
  Result := 2;
end;

function TMyEditor.GetVerb(Index: Integer): String;
begin
  case Index of
    0: Result := '&DoThis ...';
    1: Result := 'Do&That';
  end;
end;
```

Remarque Veillez à ce que votre méthode *GetVerb* renvoie une valeur pour chaque index possible indiqué par *GetVerbCount*.

Implémentation des commandes

Lorsque la commande fournie par *GetVerb* est sélectionnée dans le concepteur, la méthode *ExecuteVerb* est appelée. Pour chaque commande que vous spécifiez dans la méthode *GetVerb*, implémentez une action dans la méthode *ExecuteVerb*. Vous pouvez accéder au composant en cours de modification à l'aide de la propriété *Component* de l'éditeur.

Par exemple, la méthode *ExecuteVerb* suivante implémente les commandes de la méthode *GetVerb* de l'exemple précédent.

```
procedure TMyEditor.ExecuteVerb(Index: Integer);
```

```
var
  MySpecialDialog: TMyDialog;
begin
  case Index of
    0: begin
         MyDialog := TMySpecialDialog.Create(Application);          { instantie l'éditeur }
         if MySpecialDialog.Execute then;                      { si l'utilisateur valide la
                                                              // boîte de dialogue par OK... }
          MyComponent.FThisProperty := MySpecialDialog.ReturnValue;{ ...utilise la valeur }
         MySpecialDialog.Free;                                    { détruit l'éditeur }
       end;
    1: That;                                                  { appelle la méthode That }
  end;
end;
```

Modification du comportement suite à un double-clic

Lorsque vous double-cliquez sur le composant, la méthode *Edit* du composant est appelée. Par défaut, la méthode *Edit* exécute la première commande ajoutée au menu contextuel. Ainsi, dans l'exemple précédent, le fait de double-cliquer sur le composant exécute la commande DoThis.

Même si l'exécution de la première commande est généralement une bonne idée, vous pouvez modifier ce comportement par défaut. Par exemple, vous pouvez définir un comportement différent si :

- vous n'ajoutez aucune commande au menu contextuel ;
- vous souhaitez afficher une boîte de dialogue qui combine plusieurs commandes lorsque l'utilisateur double-clique sur le composant.

Surchargez la méthode *Edit* pour spécifier un nouveau comportement lorsque l'utilisateur double-clique sur le composant. Par exemple, la méthode *Edit* suivante appelle une boîte de dialogue de fontes lorsque l'utilisateur double-clique sur le composant :

```
procedure TMyEditor.Edit;
var
  FontDlg: TFontDialog;
begin
  FontDlg := TFontDialog.Create(Application);
  try
    if FontDlg.Execute then
      MyComponent.FFont.Assign(FontDlg.Font);
  finally
    FontDlg.Free
  end;
end;
```

Remarque Si vous souhaitez qu'un double-clic sur le composant affiche l'éditeur de code d'un gestionnaire d'événement, utilisez *TDefaultEditor* comme classe de base pour votre éditeur de composants au lieu de *TComponentEditor*. Puis, au lieu de surcharger la méthode *Edit*, surchargez la méthode protégée *TDefaultEditor.EditProperty*. *EditProperty* recherche les gestionnaires d'événement

du composant et affiche le premier qu'il trouve. Vous pouvez modifier ce comportement pour visualiser un événement particulier. Par exemple :

```
procedure TMyEditor.EditProperty(PropertyEditor: TPropertyEditor;
  Continue, FreeEditor: Boolean)
begin
  if (PropertyEditor.ClassName = 'TMethodProperty') and
    (PropertyEditor.GetName = 'OnSpecialEvent') then
    // DefaultEditor.EditProperty(PropertyEditor, Continue, FreeEditor);
end;
```

Ajout de formats de Presse-papiers

Par défaut, lorsque l'utilisateur choisit Copier lorsqu'un composant est sélectionné dans l'EDI, le composant est copié dans le format interne de Delphi. Il peut ensuite être collé dans une autre fiche ou module de données. Votre composant peut copier d'autres formats dans le Presse-papiers en surchargeant la méthode *Copy*.

Par exemple, la méthode *Copy* suivante permet à un composant *TImage* de copier son image dans le Presse-papiers. Cette image est ignorée par l'EDI de Delphi, mais peut être collée dans d'autres applications.

```
procedure TMyComponent.Copy;
var
  MyFormat : Word;
  AData,APalette : THandle;
begin
  TImage(Component).Picture.Bitmap.SaveToClipBoardFormat(MyFormat, AData, APalette);
  ClipBoard.SetAsHandle(MyFormat, AData);
end;
```

Recensement d'un éditeur de composants

Une fois que l'éditeur de composants est défini, il peut être recensé pour être utilisé avec une classe composant particulière. Un éditeur de composants recensé est créé pour chaque composant de cette classe lorsqu'il est sélectionné dans le concepteur de fiche.

Pour associer un éditeur de composants à une classe composant, appelez *RegisterComponentEditor*. *RegisterComponentEditor* adopte le nom de la classe composant qui utilise l'éditeur et le nom de la classe éditeur de composants que vous avez définie. Par exemple, l'instruction suivante recense une classe éditeur de composants nommée *TMyEditor* en vue de son utilisation avec tous les composants de type *TMyComponent* :

```
RegisterComponentEditor(TMyComponent, TMyEditor);
```

Placez l'appel à *RegisterComponentEditor* dans la procédure *Register* où vous recensez votre composant. Par exemple, si un nouveau composant nommé *TMyComponent* et son éditeur de composants *TMyEditor* sont tous les deux

implémentés dans la même unité, le code suivant recense le composant et son association à l'éditeur de composants.

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TMyComponent);
  RegisterComponentEditor(classes[0], TMyEditor);
end;
```

Compilation des composants en paquets

Une fois vos composants recensés, vous devez les compiler en paquets avant de les installer dans l'EDI. Un paquet peut contenir un ou plusieurs composants ainsi que des éditeurs de propriétés personnalisés. Pour plus d'informations sur les paquets, voir chapitre 11, "Utilisation des paquets et des composants".

Pour créer et compiler un paquet, voir "Création et modification de paquets" à la page 11-7. Placez les unités de code source de vos composants personnalisés dans la liste Contains du paquet. Si vos composants dépendent d'autres paquets, incluez ces derniers dans la liste Requires.

Pour installer vos composants dans l'EDI, voir "Installation de paquets de composants" à la page 11-6.

Chapitre

48

Modification d'un composant existant

Le moyen le plus simple de créer un composant consiste à le dériver d'un composant qui réalise la presque totalité des fonctions souhaitées et lui apporter ensuite quelques modifications. L'exemple de ce chapitre modifie le composant mémo standard pour créer un mémo qui ne fait pas de saut de ligne par défaut.

La valeur de la propriété *WordWrap* du composant mémo est initialisée à *True*. Si vous utilisez fréquemment des mémos n'effectuant pas de saut de ligne, vous pouvez créer un nouveau composant mémo qui ne fait pas de saut de ligne par défaut.

Remarque Pour modifier les propriétés publiées ou enregistrer des gestionnaires d'événements spécifiques pour un composant existant, il est souvent plus simple d'utiliser un *modèle de composant* plutôt que de créer une classe.

La modification d'un composant existant se fait en deux étapes :

- Création et recensement du composant
- Modification de la classe composant

Création et recensement du composant

La création d'un composant débute toujours de la même façon : vous créez une unité, puis dérivez et recensez une classe composant avant de l'installer dans la palette des composants. Ce processus est décrit dans "Création d'un nouveau composant" à la page 40-9.

Pour notre exemple, suivez la procédure générale de création d'un composant en tenant compte des spécificités suivantes :

- Nommez l'unité du composant *Memos*.

- Dérivez un nouveau type de composant appelé *TWrapMemo*, descendant de *TMemo*.
- Recensez *TWrapMemo* sur la page Exemples de la palette des composants.

Le fichier en-tête que vous obtenez doit ressembler à ceci :

```
unit Memos;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, StdCtrls;
type
  TWrapMemo = class(TMemo)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TWrapMemo]);
end;
end.
```

Si vous compilez et installez maintenant le nouveau composant, il se comportera exactement comme son ancêtre, *TMemo*. Dans la section suivante, vous effectuerez une simple modification à votre composant.

Modification de la classe composant

Une fois la nouvelle classe composant créée, vous pouvez lui apporter presque toutes les modifications que vous voulez. Dans notre exemple, vous allez changer la valeur par défaut d'une propriété du composant mémo. Cela implique deux changements mineurs dans la classe composant :

- Surcharge du constructeur.
- Spécification de la nouvelle valeur par défaut de la propriété.

Le constructeur définit la valeur de la propriété. La valeur par défaut indique à Delphi quelles valeurs stocker dans le fichier fiche (.dfm pour la VCL et .xfm pour la CLX). Delphi ne stocke que les valeurs qui diffèrent de la valeur par défaut, c'est pourquoi il est important d'effectuer les deux étapes.

Surcharge du constructeur

Lorsque vous placez un composant dans une fiche au moment de la conception ou lorsqu'une application en cours d'exécution construit un composant, le constructeur du composant définit les valeurs des propriétés. Quand un composant est chargé depuis un fichier fiche, l'application définit toutes les propriétés qui ont été modifiées lors de la conception.

Remarque Lorsque vous surchargez un constructeur, le nouveau constructeur doit appeler le constructeur reçu en héritage avant toute autre action. Pour plus d'informations, voir "Surcharge des méthodes" à la page 41-9.

Dans notre exemple, votre nouveau composant doit surcharger le constructeur transmis en héritage par *TMemo* en attribuant la valeur *False* à la propriété *WordWrap*. Pour ce faire, ajoutez le constructeur surchargé à la prédéclaration, puis écrivez le nouveau constructeur dans la partie **implémentation** de l'unité :

```
type
  TWrapMemo = class(TMemo)
  public                                        { les constructeurs sont toujours publics }
    constructor Create(AOwner: TComponent); override;          { cette syntaxe est toujours
                                                                          // identique }
  end;
⋮
constructor TWrapMemo.Create(AOwner: TComponent);        { ceci va après l'implémentation }
begin
  inherited Create(AOwner);                            { Faites TOUJOURS ceci en premier ! }
  WordWrap := False;                                  { définit la nouvelle valeur désirée }
end;
```

Vous pouvez maintenant installer le nouveau composant sur la palette des composants et l'ajouter à une fiche. Remarquez que la propriété *WordWrap* est dorénavant initialisée à *False*.

Si vous changez une valeur de propriété initiale, vous devez aussi désigner cette valeur comme étant celle par défaut. Si vous échouez à faire correspondre la valeur définie par le constructeur à celle spécifiée par défaut, Delphi ne peut pas stocker, ni restaurer la valeur correcte.

Spécification de la nouvelle valeur par défaut de la propriété

Lorsque Delphi stocke la description d'une fiche dans un fichier fiche, il ne stocke que les valeurs des propriétés différentes des valeurs par défaut. La taille du fichier fiche reste minime et le chargement est plus rapide. Si vous créez une propriété ou si vous changez la valeur par défaut d'une propriété existante, c'est une bonne idée de mettre à jour la déclaration de la propriété en y incluant la nouvelle valeur par défaut. Les fichiers fiche ainsi que le chargement et les valeurs par défaut sont expliqués en détail dans le chapitre 47, "Accessibilité des composants au moment de la conception".

Pour changer la valeur par défaut d'une propriété, redéclarez le nom de la propriété, suivi de la directive **default** et de la nouvelle valeur par défaut. Il n'est pas nécessaire de redéclarer la propriété entière mais uniquement le nom et la valeur par défaut.

Pour le composant mémo de saut à la ligne automatique, redéclarez la propriété *WordWrap* dans la partie **published** de la déclaration d'objet, avec la valeur *False* par défaut :

```
type
  TWrapMemo = class(TMemo)
```

```
  .
  .
  .
published
  property WordWrap default False;
end;
```

Spécifier la valeur par défaut de la propriété n'affecte en rien le fonctionnement du composant. Vous devez toujours initialiser la valeur dans le constructeur du composant. La redéclaration de la valeur par défaut assure que Delphi connaît quand *WordWrap* doit être écrit dans le fichier fiche.

Chapitre

49

Création d'un composant graphique

Un contrôle graphique est un composant simple. Un contrôle purement graphique ne reçoit jamais la focalisation et n'a donc pas besoin de handle de fenêtre. Les utilisateurs peuvent quand même manipuler le contrôle avec la souris, mais il n'y a pas d'interface clavier.

TShape, le composant graphique présenté dans ce chapitre, correspond au composant forme inclus dans la page Supplément de la palette des composants. Bien que le composant que nous allons créer soit identique au composant forme standard, vous devrez lui donner un nom différent pour éviter des doublons d'identificateur. Ce chapitre appelle son composant forme *TSampleShape* et illustre toutes les étapes de sa création :

- Création et recensement du composant
- Publication des propriétés héritées
- Ajout de fonctionnalités graphiques

Création et recensement du composant

La création d'un composant débute toujours de la même façon. Vous créez une unité et vous recensez le composant avant de l'installer dans la palette des composants. Ce processus est décrit dans "Création d'un nouveau composant" à la page 40-9.

Pour cet exemple, suivez la procédure générale de création d'un composant en tenant compte de ces spécifités :

- Appelez l'unité du composant *Shapes*.
- Dérivez un nouveau type de composant appelé *TSampleShape*, descendant de *TGraphicControl*.
- Recensez *TSampleShape* sur la page Exemples de la palette des composants.

L'unité que vous obtenez doit ressembler à ceci :

```
unit Shapes;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
  TSampleShape = class(TGraphicControl)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponent('Samples', [TSampleShape]);
end;
end.
```

Publication des propriétés héritées

Lorsque vous dérivez un type de composant, vous choisissez parmi les propriétés et les événements déclarés dans la partie **protected** de la classe ancêtre ceux que vous voulez rendre disponibles aux utilisateurs du nouveau composant. *TGraphicControl* publie toutes les propriétés qui permettent au composant de fonctionner en tant que contrôle, donc les seules fonctionnalités que vous devez publier sont celles dont vous avez besoin pour répondre aux événements de souris et aux opérations glisser-déplacer.

La publication des propriétés et des événements reçus en héritage est expliquée dans "Publication des propriétés héritées" à la page 42-3 et "Rendre visibles des événements" à la page 43-6. Ces deux processus impliquent la redéclaration du nom des propriétés dans la partie **published** de la déclaration de classe.

S'agissant de notre contrôle forme, vous devez publier les trois événements associés à la souris ainsi que les deux propriétés associées aux opérations glisser-déplacer :

```
type
  TSampleShape = class(TGraphicControl)
  published
    property DragCursor;        { propriétés glisser-déplacer }
    property DragMode;
    property OnDragDrop;        { événements glisser-déplacer }
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;       { événements de souris }
    property OnMouseMove;
    property OnMouseUp;
  end;
```

Le contrôle forme de notre exemple rend les interactions associées à la souris et aux opérations glisser-déplacer accessibles à l'utilisateur.

Ajout de fonctionnalités graphiques

Lorsque votre composant graphique a été déclaré et qu'ont été publiées toutes les propriétés reçues en héritage que vous voulez rendre disponibles, vous pouvez ajouter les fonctionnalités graphiques qui caractériseront votre composant :

1 Détermination de ce qui doit être dessiné.
2 Dessin de l'image du composant.

En outre, s'agissant du contrôle forme de notre exemple, vous allez ajouter certaines propriétés qui serviront aux développeurs d'applications pour personnaliser l'apparence du contrôle lors de la conception.

Détermination de ce qui doit être dessiné

Un contrôle graphique est capable de changer son apparence pour refléter un changement de condition, y compris une intervention de l'utilisateur. Un contrôle graphique qui aurait toujours le même aspect ne devrait pas être un composant. Si vous voulez une image statique, importez une image au lieu d'utiliser un contrôle.

Généralement, l'apparence d'un contrôle graphique dépend de plusieurs propriétés. Par exemple, le contrôle jauge dispose de propriétés qui déterminent sa forme et son orientation et si la représentation de la progression est numérique ou graphique. De même, notre contrôle forme doit disposer d'une propriété qui détermine le type de forme qu'il doit dessiner.

Pour donner à votre contrôle une propriété qui détermine la forme dessinée, ajoutez une propriété intitulée *Shape*. Ce processus se déroule en trois étapes :

1 Déclaration du type de la propriété.
2 Déclaration de la propriété.
3 Ecriture de la méthode d'implémentation.

La création de propriété est expliquée en détail chapitre 42, "Création de propriétés"

Déclaration du type de la propriété

Lorsque vous déclarez une propriété dont le type est défini par l'utilisateur, le type de la propriété doit être déclaré avant la classe qui inclut cette propriété. Les types énumérés sont fréquemment employés par les propriétés.

S'agissant de notre contrôle forme, vous aurez besoin d'un type énuméré avec un élément défini pour chaque forme que le contrôle est en mesure de dessiner.

Ajoutez la définition de type suivante avant la déclaration de classe du contrôle forme.

```
type
  TSampleShapeType = (sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
    sstEllipse, sstCircle);
  TSampleShape = class(TGraphicControl) { existe déjà }
```

Vous pouvez maintenant utiliser ce type pour déclarer une nouvelle propriété dans la classe.

Déclaration de la propriété

Généralement, pour déclarer une propriété, vous déclarez un champ privé pour stocker les données de la propriété puis vous spécifiez les méthodes pour lire et/ou écrire sa valeur. Souvent, la méthode pour lire la valeur n'est pas nécessaire car un simple pointage sur la valeur stockée suffit.

S'agissant de notre contrôle forme, vous aurez à déclarer un champ contenant la forme courante, puis à déclarer une propriété qui lit ce champ et l'écrit via un appel de méthode.

Ajoutez les déclarations suivantes dans *TSampleShape* :

```
type
  TSampleShape = class(TGraphicControl)
  private
    FShape: TSampleShapeType;     { donnée membre pour contenir la valeur de la propriété }
    procedure SetShape(Value: TSampleShapeType);
  published
    property Shape: TSampleShapeType read FShape write SetShape;
  end;
```

Il ne vous reste plus qu'à ajouter l'implémentation de *SetShape*.

Ecriture de la méthode d'implémentation

Lorsque la partie **read** ou **write** d'une définition de propriété utilise une méthode plutôt qu'un accès direct aux données stockées de la propriété, vous devez implémenter ces méthodes.

Ajoutez l'implémentation de la méthode *SetShape* à la partie **implémentation** de l'unité :

```
procedure TSampleShape.SetShape(Value: TSampleShapeType);
begin
  if FShape <> Value then                        { ignore s'il n'y a pas eu de changement }
  begin
    FShape := Value;                                        { stocke la nouvelle valeur }
    Invalidate;                                  { force le dessin avec la nouvelle forme }
  end;
end;
```

Surcharge du constructeur et du destructeur

Pour changer les valeurs par défaut des propriétés et initialiser les classes appartenant à votre composant, vous devez surcharger le constructeur et le destructeur reçus en héritage. Dans les deux cas, n'oubliez pas d'appeler la méthode reçue en héritage.

Modification des valeurs par défaut des propriétés

La taille par défaut d'un contrôle graphique étant réduite, vous pouvez modifier sa largeur et sa hauteur dans le constructeur. La modification des valeurs par défaut des propriétés est abordée plus en détail chapitre 48, "Modification d'un composant existant".

Dans notre exemple, le contrôle forme définit sa taille par un carré de 65 pixels de côté.

Ajoutez le constructeur surchargé dans la déclaration de la classe composant :

```
type
  TSampleShape = class(TGraphicControl)
  public                                              { constructeurs toujours publics }
    constructor Create(AOwner: TComponent); override          { surcharger la directive }
  end;
```

1 Redéclarez les propriétés *Height* et *Width* avec leurs nouvelles valeurs par défaut :

```
type
  TSampleShape = class(TGraphicControl)
  ⋮
  published
    property Height default 65;
    property Width default 65;
  end;
```

2 Ecrivez le nouveau constructeur dans la partie **implémentation** de l'unité :

```
constructor TSampleShape.Create(AOwner: TComponent);
   begin
     inherited Create(AOwner);  { appelle toujours le constructeur hérité }
     Width := 65;
     Height := 65;
   end;
```

Publication du crayon et du pinceau

Par défaut, un canevas dispose d'un crayon fin et noir et d'un pinceau plein et blanc. Pour permettre aux développeurs de changer le crayon et le pinceau, vous devez leur fournir des classes pour les manipuler lors de la conception, puis copier les classes dans le canevas lors des opérations de dessin. Des classes telles qu'un crayon ou un pinceau auxiliaire sont appelées *classes ayant un propriétaire* car elles appartiennent au composant qui est responsable de leur création et de leur destruction.

La gestion des classes ayant un propriétaire se déroule en quatre étapes :

1 Déclaration des champs de la classe.
2 Déclaration des propriétés d'accès.
3 Initialisation des classes ayant un propriétaire.
4 Définition des propriétés des classes ayant un propriétaire.

Déclaration des champs de classe

Chaque classe appartenant à un composant doit avoir un champ déclaré dans ce composant. Le champ de classe garantit que le composant dispose toujours d'un pointeur sur l'objet qui lui appartient afin de lui permettre de le détruire avant de se détruire lui-même. Généralement, un composant initialise les objets dont il est propriétaire dans son constructeur et les détruit dans son destructeur.

Les champs de classe des objets ayant un propriétaire sont presque toujours déclarés **private**. Si des applications (ou d'autres composants) ont besoin d'accéder aux objets ayant un propriétaire, vous devez pour cela déclarer des propriétés publiées ou publiques.

Ajoutez des champs de classe pour le crayon et le pinceau de votre contrôle forme :

```
type
  TSampleShape = class(TGraphicControl)
  private           { les données membres sont presque toujours privées }
    FPen: TPen;     { donnée membre pour l'objet crayon }
    FBrush: TBrush; { donnée membre pour l'objet pinceau }
    ⋮
  end;
```

Déclaration des propriétés d'accès

Vous pouvez fournir les accès aux objets appartenant à un composant en déclarant des propriétés de même type que ces objets. Cela offre aux développeurs un moyen d'accéder aux objets lors de la conception ou de l'exécution. Généralement, la partie **read** de la propriété ne fait que référencer le champ de classe, alors que la partie **write** appelle une méthode qui permet au composant de réagir aux changements apportés à l'objet.

Ajoutez des propriétés à votre contrôle forme pour fournir les accès aux champs du crayon et du pinceau. Vous allez également déclarer les méthodes pour réagir aux changements de crayon ou de pinceau.

```
type
  TSampleShape = class(TGraphicControl)
  ⋮
  private                                          { ces méthodes doivent être privées }
    procedure SetBrush(Value: TBrush);
    procedure SetPen(Value: TPen);
  published                                  { les rend disponible lors de la conception }
    property Brush: TBrush read FBrush write SetBrush;
    property Pen: TPen read FPen write SetPen;
  end;
```

Vous devez ensuite écrire les méthodes *SetBrush* et *SetPen* dans la partie implémentation de l'unité :

```
procedure TSampleShape.SetBrush(Value: TBrush);
begin
  FBrush.Assign(Value);                 { remplace le pinceau existant par le paramètre }
end;
```

```
procedure TSampleShape.SetPen(Value: TPen);
begin
  FPen.Assign(Value);                    { remplace le crayon existant par le paramètre }
end;
```

Affecter directement le contenu de *Value* à *FBrush*...

```
FBrush := Value;
```

...écraserait le pointeur interne de *FBrush*, ferait perdre de la mémoire et créerait une série de problèmes de propriété.

Initialisation des classes ayant un propriétaire

Si vous ajoutez des classes dans votre composant, le constructeur de ce dernier doit les initialiser pour que l'utilisateur puisse interagir avec ces objets lors de l'exécution. De même, le destructeur du composant doit également détruire les objets appartenant au composant avant de détruire ce dernier.

Comme vous avez déclaré un crayon et un pinceau dans le contrôle forme, vous devez les initialiser dans le constructeur du contrôle forme et les détruire dans son destructeur :

1 Construisez le crayon et le pinceau dans le constructeur du contrôle forme :

```
constructor TSampleShape.Create(AOwner: TComponent);
  begin
    inherited Create(AOwner);          { appelle toujours le contrôle hérité }
    Width := 65;
    Height := 65;
    FPen := TPen.Create;               { construit le crayon }
    FBrush := TBrush.Create;           { construit le pinceau }
  end;
```

2 Ajoutez le destructeur surchargé dans la déclaration de la classe composant :

```
type
    TSampleShape = class(TGraphicControl)
    public                             { les destructeurs sont toujours publics}
      constructor Create(AOwner: TComponent); override;
      destructor Destroy; override;    { ne pas oublier la surcharge de directive }
    end;
```

3 Ecrivez le nouveau destructeur dans la partie **implémentation** de l'unité :

```
destructor TSampleShape.Destroy;
  begin
    FPen.Free;                         { détruit l'objet crayon }
    FBrush.Free;                       { détruit l'objet pinceau }
    inherited Destroy;                 { appelle aussi toujours le destructeur hérité }
  end;
```

Définition des propriétés des classes ayant un propriétaire

L'une des dernières étapes de la gestion des classes crayon et pinceau consiste à provoquer un nouveau dessin du contrôle forme si le crayon ou le pinceau est modifié. Comme les classes crayon et pinceau disposent toutes les deux d'un

événement *OnChange*, vous pouvez créer une méthode dans le contrôle forme et faire pointer les deux événements *OnChange* sur cette méthode.

Ajoutez la méthode suivante au contrôle forme et effectuez la mise à jour du constructeur du composant pour affecter aux événements du crayon et du pinceau cette nouvelle méthode :

```
type
  TSampleShape = class(TGraphicControl)
  published
    procedure StyleChanged(Sender: TObject);
  end;
⋮
implementation
⋮
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                    { appelez toujours le constructeur hérité }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                          { construit le crayon }
  FPen.OnChange := StyleChanged;              { affecte la méthode à l'événement OnChange }
  FBrush := TBrush.Create;                                      { construit le pinceau }
  FBrush.OnChange := StyleChanged;            { affecte la méthode à l'événement OnChange }
end;

procedure TSampleShape.StyleChanged(Sender: TObject);
begin
  Invalidate;                                          { efface et redessine le composant }
end;
```

Ces modifications faites, le composant se redessine pour refléter tout changement du crayon ou du pinceau.

Dessin de l'image du composant

L'essentiel d'un contrôle graphique se résume à sa façon de dessiner son image à l'écran. Le type abstrait *TGraphicControl* définit une méthode appelée *Paint* que vous devez surcharger pour peindre l'image voulue dans votre contrôle.

La méthode *Paint* de votre contrôle forme doit accomplir plusieurs tâches :

- Utiliser le crayon et le pinceau sélectionnés par l'utilisateur.
- Utiliser la forme sélectionnée.
- Ajuster les coordonnées pour que les carrés et les cercles utilisent une largeur et une hauteur identiques.

La surcharge de la méthode *Paint* nécessite deux étapes :

1. Ajout de *Paint* dans la déclaration du composant.
2. Insertion de la méthode *Paint* dans la partie **implémentation** de l'unité.

S'agissant de notre contrôle forme, vous devez ajouter la déclaration suivante à la déclaration de classe :

```
type
  TSampleShape = class(TGraphicControl)
  ⋮
  protected
    procedure Paint; override;
  ⋮
  end;
```

Vous devez ensuite écrire la méthode dans la partie **implémentation** de l'unité :

```
procedure TSampleShape.Paint;
begin
  with Canvas do
  begin
    Pen := FPen;                                          { copie le crayon du composant }
    Brush := FBrush;                                      { copie le pinceau du composant }
    case FShape of
      sstRectangle, sstSquare:
        Rectangle(0, 0, Width, Height);                   { dessine les rectangles et carrés }
      sstRoundRect, sstRoundSquare:
        RoundRect(0, 0, Width, Height, Width div 4, Height div 4);     { dessine des formes
                                                                         // arrondies }
      sstCircle, sstEllipse:
        Ellipse(0, 0, Width, Height);                     { dessine des formes arrondies }
    end;
  end;
end;
```

Paint est appelée à chaque fois que le contrôle doit mettre à jour son image. Les contrôles sont dessinés lorsqu'ils s'affichent pour la première fois ou lorsqu'une fenêtre qui se trouvait au-dessus disparaît. En outre, vous pouvez forcer le dessin en appelant *Invalidate*, comme le fait la méthode *StyleChanged*.

Adaptation du dessin de la forme

Le contrôle forme standard effectue une tâche supplémentaire que ne fait pas encore le contrôle forme de notre exemple : il gère les carrés et les cercles ainsi que les rectangles et les ellipses. Pour ce faire, vous devez écrire le code qui trouve le côté le plus petit et centre l'image.

Voici une méthode *Paint* parfaitement adaptée aux carrés et aux ellipses :

```
procedure TSampleShape.Paint;
var
  X, Y, W, H, S: Integer;
begin
  with Canvas do
  begin
    Pen := FPen;                                          { copie le crayon du composant }
    Brush := FBrush;                                      { copie le pinceau du composant }
    W := Width;                                           { utilise la largeur du composant }
    H := Height;                                          { utilise la hauteur du composant }
```

```
    if W < H then S := W else S := H;  { enregistre du plus petit pour les cercles/carrés }

    case FShape of                          { ajuste la hauteur, la largeur et la position }
      sstRectangle, sstRoundRect, sstEllipse:
        begin
          X := 0;                  { l'origine est l'angle supérieur gauche de ces formes }
          Y := 0;
        end;
      sstSquare, sstRoundSquare, sstCircle:
        begin
          X := (W - S) div 2;                                  { centre horizontalement... }
          Y := (H - S) div 2;                                     { ...puis verticalement }
          W := S;                { utilise la dimension la plus petite pour la largeur... }
          H := S;                                                 { ...et pour la hauteur }
        end;
    end;

    case FShape of
      sstRectangle, sstSquare:
        Rectangle(X, Y, X + W, Y + H);             { dessine les rectangles et les carrés }
      sstRoundRect, sstRoundSquare:
        RoundRect(X, Y, X + W, Y + H, S div 4, S div 4);   { dessine des formes arrondies }
      sstCircle, sstEllipse:
        Ellipse(X, Y, X + W, Y + H);                       { dessine les formes arrondies }
    end;
  end;
end;
```

Chapitre

50

Personnalisation d'une grille

Delphi fournit plusieurs composants abstraits que vous pouvez utiliser comme points de départ pour personnaliser vos composants. Les grilles et les boîtes liste sont les plus importants. Dans ce chapitre, nous allons voir comment créer un petit calendrier en partant du composant grille de base *TCustomGrid*.

La création du calendrier nécessite les étapes suivantes :

- Création et recensement du composant
- Publication des propriétés héritées
- Modification des valeurs initiales
- Redimensionnement des cellules
- Remplissage des cellules
- Navigation de mois en mois et d'année en année
- Navigation de jour en jour

Le composant calendrier résultant est pratiquement identique au composant *TCalendar* contenu dans la page Exemples de la palette des composants.

Création et recensement du composant

La création d'un composant débute toujours de la même façon. Vous créez une unité et vous recensez le composant avant de l'installer dans la palette des composants. Ce processus est décrit dans "Création d'un nouveau composant" à la page 40-9.

Dans cet exemple, suivez la procédure générale de création d'un composant, avec les spécificités suivantes :

- Appelez l'unité du composant *CalSamp*.
- Dérivez un nouveau type de composant appelé *TSampleCalendar*, descendant de *TCustomGrid*.

- Recensez *TSampleCalendar* dans la page Exemples de la palette des composants.

L'unité résultante dérivant de *TCustomGrid* dans la VCL devrait ressembler à ceci :

```
unit CalSamp;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Grids;

type
  TSampleCalendar = class(TCustomGrid)
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TSampleCalendar]);
end;

end.
```

Si la dérivation s'effectue à partir de la version CLX de *TCustomGrid*, seule la clause **uses** est différente, elle présente alors les unités CLX.

Si vous installez le composant calendrier maintenant, vous verrez qu'il apparaît sur la page Exemples. Les seules propriétés disponibles sont les propriétés de contrôle les plus basiques. L'étape suivante consiste à rendre disponible certaines des propriétés plus spécialisées aux utilisateurs du calendrier.

Remarque Bien que vous puissiez installer le composant calendrier exemple que vous venez de compiler, n'essayez pas de le placer tout de suite sur une fiche. Le composant *TCustomGrid* contient une méthode *DrawCell* abstraite qui doit être redéclarée avant que les objets d'instance puissent être créés. La surcharge de la méthode *DrawCell* est décrite dans "Remplissage des cellules".

Publication des propriétés héritées

Le composant grille abstrait, *TCustomGrid*, fournit de nombreuses propriétés **protected**. Vous pouvez choisir parmi ces propriétés celles que vous voulez rendre accessibles aux utilisateurs du contrôle calendrier.

Pour rendre accessibles aux utilisateurs de vos composants les propriétés protégées qu'ils reçoivent en héritage, vous devez redéclarer ces propriétés dans la partie **published** de la déclaration de vos composants.

S'agissant du contrôle calendrier, vous devez publier les propriétés et les événements, comme ci-dessous :

```
type
  TSampleCalendar = class(TCustomGrid)
```

```
  published
    property Align;  { propriétés publiées }
    property BorderStyle;
    property Color;
    property Font;
    property GridLineWidth;
    property ParentColor;
    property ParentFont;
    property OnClick;  { événements publiés }
    property OnDblClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
  end;
```

Il existe bien d'autres propriétés ne s'appliquant pas à un calendrier qui sont publiables, par exemple la propriété *Options* qui permet à l'utilisateur de choisir les lignes de la grille à dessiner.

Si vous installez le composant calendrier modifié dans la palette des composants et l'utilisez dans une application, vous trouverez bien d'autres propriétés et événements opérationnels. Nous allons maintenant commencer à ajouter de nouvelles fonctionnalités au composant.

Modification des valeurs initiales

Un calendrier est essentiellement une grille avec un nombre fixe de lignes et de colonnes, ne contenant pas nécessairement des dates. Les propriétés *ColCount* et *RowCount* de la grille n'ont donc pas été publiées, car il est peu probable que les utilisateurs du calendrier voudront afficher autre chose que les sept jours de la semaine. Vous devez néanmoins définir les valeurs initiales de ces propriétés en fonction des sept jours de la semaine.

Pour changer les valeurs initiales des propriétés du composant, vous devez surcharger le constructeur afin qu'il affecte les valeurs voulues. Le constructeur doit être virtuel.

Souvenez-vous que vous devez ajouter le constructeur à la partie **public** de la déclaration de la classe du composant, puis écrire le nouveau constructeur dans la partie **implémentation** de l'unité du composant. La première instruction du nouveau constructeur doit toujours être un appel au constructeur hérité.

```
type
  TSampleCalendar = class(TCustomGrid
  public
    constructor Create(AOwner: TComponent); override;
  ⋮
  end;
⋮
constructor TSampleCalendar.Create(AOwner: TComponent);
```

```
begin
  inherited Create(AOwner);                                  { appelle le constructeur hérité }
  ColCount := 7;                                                  { toujours 7 jours/semaine }
  RowCount := 7;                                          { toujours 6 semaines plus les titres }
  FixedCols := 0;                                                  { aucun libellé de ligne }
  FixedRows := 1;                                             { une ligne pour les noms de jour }
  ScrollBars := ssNone;                                          { pas de défilement nécessaire }
  Options := Options - [goRangeSelect] + [goDrawFocusSelected];     {désactive la sélection
                                                                          // d'intervalle }
end;
```

Le calendrier a dorénavant sept colonnes et sept lignes, avec la ligne de titre fixe (ou qui ne défile pas).

Redimensionnement des cellules

VCL Lorsqu'un utilisateur ou une application modifie la taille d'une fenêtre ou d'un contrôle, Windows envoie le message *WM_SIZE* à la fenêtre ou au contrôle concerné pour lui permettre d'ajuster les paramètres nécessaires afin de dessiner ultérieurement son image dans la nouvelle taille. Votre composant VCL peut répondre à ce message en modifiant la taille des cellules de façon à ce qu'elles s'inscrivent dans les limites du contrôle. Pour répondre au message *WM_SIZE*, vous devez ajouter au composant une méthode de gestion du message.

La création d'une méthode de gestion de message est décrite en détail dans "Création de nouveaux gestionnaires de messages" à la page 46-5.

Dans notre exemple, le contrôle calendrier devant répondre au message *WM_SIZE*, vous devez ajouter au contrôle une méthode protégée appelée *WMSize* et indexée par le message *WM_SIZE*, puis écrire la méthode de calcul de la taille des cellules qui permettra à toutes d'être visibles :

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure WMSize(var Message: TWMSize); message WM_SIZE;
  ⋮
  end;
⋮
procedure TSampleCalendar.WMSize(var Message: TWMSize);
var
  GridLines: Integer;                                        { variable locale temporaire }
begin
  GridLines := 6 * GridLineWidth;       { calcule de taille combinée de toutes les lignes }
  DefaultColWidth := (Message.Width - GridLines) div 7;       { définit la nouvelle largeur
                                                                  // de cellule par défaut }
  DefaultRowHeight := (Message.Height - GridLines) div 7;          { ainsi que sa hauteur }
end;
```

Maintenant lorsque le calendrier est redimensionné, il affiche toutes les cellules dans la taille maximum avec laquelle ils peuvent rentrer dans le contrôle.

CLX Dans la CLX, les changements de taille d'une fenêtre ou d'un contrôle sont notifiés automatiquement par un appel à la méthode protégée *BoundsChanged*. Votre composant CLX peut répondre à cette notification en modifiant la taille des cellules afin qu'elles tiennent toutes dans les limites du contrôle.

Dans ce cas, le contrôle calendrier doit redéfinir *BoundsChanged* afin qu'elle calcule la taille de cellule adéquate pour que toutes les cellules soient visibles avec la nouvelle taille :

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure BoundsChanged; override;
  ⋮
  end;
⋮
procedure TSampleCalendar.BoundsChanged;
var
  GridLines: Integer;                                    { variable locale temporaire }
begin
  GridLines := 6 * GridLineWidth;     { calcul de la taille combinée de toutes les lignes }
  DefaultColWidth := (Width - GridLines) div 7;  { nouvelle largeur de cellule par défaut }
  DefaultRowHeight := (Height - GridLines) div 7;{ nouvelle hauteur de cellule par défaut }
  inherited; { appelle la méthode héritée }
end;
```

Remplissage des cellules

Un contrôle grille se remplit cellule par cellule. S'agissant du calendrier, cela revient à calculer une date (si elle existe) pour chaque cellule. Le dessin par défaut des cellules de la grille s'opère dans une méthode virtuelle intitulée *DrawCell*.

Pour remplir le contenu des cellules de la grille, vous devez surcharger la méthode *DrawCell*.

Les cellules de titre de la ligne fixe sont ce qu'il y a de plus facile à remplir. La bibliothèque d'exécution contient un tableau avec l'intitulé raccourci des jours et il vous faut donc insérer l'intitulé approprié à chaque colonne :

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
      override;
  end;
⋮
procedure TSampleCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect;
  AState: TGridDrawState);
begin
  if ARow = 0 then
    Canvas.TextOut(ARect.Left, ARect.Top, ShortDayNames[ACol + 1]);
                                                        { utilise les chaînes RTL }
end;
```

Suivi de la date

Pour que le contrôle calendrier soit utile, les utilisateurs ainsi que les applications doivent disposer d'un moyen de définir la date, le mois et l'année. Delphi stocke les dates et les heures dans des variables de type *TDateTime*. *TDateTime* est une représentation numérique encodée des dates et des heures particulièrement pratique pour être manipulée par un programme mais peu commode à interpréter par un utilisateur.

Vous pouvez donc stocker la date du calendrier sous une forme encodée et fournir un accès direct à cette valeur lors de l'exécution, mais vous pouvez aussi fournir les propriétés *Day*, *Month* et *Year* que l'utilisateur du composant peut définir lors de la conception.

Le suivi de la date dans le calendrier comprend les traitements suivants :

- Stockage interne de la date
- Accès au jour, au mois et à l'année
- Génération des numéros de jours
- Sélection du jour en cours

Stockage interne de la date

Pour stocker la date du calendrier, vous devez avoir un champ contenant la date, ainsi qu'une propriété accessible à l'exécution seulement qui fournit un accès à cette date.

L'ajout de la date interne au calendrier requiert trois étapes :

1 Déclarez une donnée membre privée pour contenir la date :

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FDate: TDateTime;
  ⋮
```

2 Initialisez la donnée membre date dans le constructeur :

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                          { déjà ici }
  ⋮                                  { d'autres initialisations ici }
  FDate := Date;                      { prend la date active du RTL }
end;
```

3 Déclarez une propriété à l'exécution pour accéder à la date encodée :

Vous aurez besoin d'une méthode pour définir la date car sa définition entraîne la mise à jour de l'image sur l'écran du contrôle :

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    procedure SetCalendarDate(Value: TDateTime);
  public
```

```
    property CalendarDate: TDateTime read FDate write SetCalendarDate;
  ⋮
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;               { définit la nouvelle valeur date }
  Refresh;                        { met à jour l'image à l'écran }
end;
```

Accès au jour, au mois et à l'année

Une date encodée numériquement est adaptée aux applications, mais l'utilisateur préférera manipuler jour, mois et année. En créant des propriétés, vous offrez un accès aux dates stockées et encodées numériquement.

Les éléments d'une date (jour, mois, année) sont des entiers. La modification de chacun d'entre eux nécessite l'encodage de la date. Vous pouvez éviter la duplication du code en partageant les méthodes d'implémentation entre les trois propriétés. Autrement dit, vous pouvez écrire deux méthodes, l'une pour lire un élément et l'autre pour l'écrire, et utiliser ces méthodes pour lire et écrire les trois propriétés.

Pour fournir un accès lors de la conception aux éléments jour, mois et année, procédez de la façon suivante :

1 Déclarez les trois propriétés, en attribuant à chacune un numéro unique d'**index** :

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  ⋮
```

2 Déclarez et écrivez les méthodes d'implémentation, définissant les différents éléments pour chaque valeur d'index :

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    function GetDateElement(Index: Integer): Integer;       { notez le paramètre Index }
    procedure SetDateElement(Index: Integer; Value: Integer);
  ⋮
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);         { éclate la date encodée en éléments }
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;
```

```
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
  AYear, AMonth, ADay: Word;
begin
  if Value > 0 then                          { tous les éléments doivent être positifs }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);{ récupère les éléments courants de la date }
    case Index of                            { définit le nouvel élément selon l'index }
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);          { encodage de la date modifiée }
    Refresh;                                    { mise à jour du calendrier visible }
  end;
end;
```

Vous pouvez maintenant définir le jour, le mois et l'année du calendrier lors de la conception à partir de l'inspecteur d'objets, ou à l'exécution à partir du code. Bien que vous n'ayez pas encore ajouté le code pour dessiner les dates dans les cellules, vous disposez maintenant de toutes les données nécessaires.

Génération des numéros de jours

Insérer les numéros des jours dans le calendrier nécessite plusieurs considérations. Le nombre de jours dans le mois dépend à la fois du mois et de l'année. Le jour de la semaine qui débute le mois dépend aussi du mois et de l'année. Utilisez la fonction *IsLeapYear* pour déterminer si l'année est bissextile. Utilisez le tableau *MonthDays* dans l'unité SysUtils pour obtenir le nombre de jours dans le mois.

Une fois récupérées les informations concernant les années bissextiles et le nombre de jours par mois, vous pouvez calculer l'endroit de la grille où s'insère chaque date. Le calcul dépend du premier jour du mois.

Comme vous devez considérer le décalage du premier jour du mois, par rapport à l'origine de la grille, pour chaque cellule à remplir, le meilleur choix consiste à calculer ce nombre après chaque changement de mois ou d'année, et de s'y reporter à chaque fois. Vous pouvez stocker cette valeur dans un champ de classe, puis mettre à jour ce champ à chaque modification de la date.

Pour remplir les cellules avec les numéros de jour appropriés, procédez de la façon suivante :

1 Ajoutez à la classe une donnée membre décalage du premier jour du mois, ainsi qu'une méthode pour mettre à jour la valeur de cette donnée membre :

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FMonthOffset: Integer;                                  { stocke le décalage }
    ⋮
  protected
    procedure UpdateCalendar; virtual;          { propriété pour l'accès au décalage }
  end;
```

```
⋮
procedure TSampleCalendar.UpdateCalendar;
var
  AYear, AMonth, ADay: Word;
  FirstDate: TDateTime;                                  { date du premier jour du mois }
begin
  if FDate <> 0 then                { ne calcule le décalage que si la date est valide }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);          { récupère les éléments de la date }
    FirstDate := EncodeDate(AYear, AMonth, 1);           { date du premier jour du mois }
    FMonthOffset := 2 - DayOfWeek(FirstDate);       { génère le décalage dans la grille }
  end;
  Refresh;                                             { toujours repeindre le contrôle }
end;
```

2 Ajoutez les instructions au constructeur et aux méthodes *SetCalendarDate* et *SetDateElement* qui appellent la nouvelle méthode de mise à jour à chaque changement de date :

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                                              { existait déjà }
  ⋮                                                  { d'autres initialisations ici }
  UpdateCalendar;                                        { définit le bon décalage }
end;

procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;                                                        { existait déjà }
  UpdateCalendar;                                   { appelait précédemment Refresh }
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
  ⋮
    FDate := EncodeDate(AYear, AMonth, ADay);               { encode la date modifiée }
    UpdateCalendar;                                 { appelait précédemment Refresh }
  end;
end;
```

3 Ajoutez une méthode au calendrier renvoyant le numéro du jour à partir des coordonnées ligne/colonne d'une cellule qui lui sont transmises :

```
function TSampleCalendar.DayNum(ACol, ARow: Integer): Integer;
begin
  Result := FMonthOffset + ACol + (ARow - 1) * 7;  { calcule le jour pour cette cellule}
  if (Result < 1) or (Result > MonthDays[IsLeapYear(Year), Month]) then
    Result := -1;                                          { renvoie -1 si incorrect }
end;
```

N'oubliez pas d'ajouter la déclaration de *DayNum* à la déclaration de type du composant.

4 Vous pouvez désormais calculer l'endroit où s'affichent les dates, et mettre à jour *DrawCell* pour remplir les cellules :

```
procedure TCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
var
```

```
    TheText: string;
    TempDay: Integer;
  begin
    if ARow = 0 then                                { s'il s'agit de la ligne de titre ...}
      TheText := ShortDayNames[ACol + 1]                      { utilise le nom du jour }
    else begin
      TheText := '';                                          { cellule vide par défaut }
      TempDay := DayNum(ACol, ARow);              { récupère le numéro pour cette cellule }
      if TempDay <> -1 then TheText := IntToStr(TempDay);     { utilise ce numéro s'il est
                                                                                  // valide }
    end;
    with ARect, Canvas do
      TextRect(ARect, Left + (Right - Left - TextWidth(TheText)) div 2,
        Top + (Bottom - Top - TextHeight(TheText)) div 2, TheText);
  end;
```

Si maintenant vous réinstallez le composant calendrier et le placez dans une fiche, les informations correspondant au mois en cours apparaîtront.

Sélection du jour en cours

Maintenant que les numéros des jours s'affichent dans les cellules du calendrier, il devient intéressant de savoir positionner la surbrillance sur le jour en cours. Comme la sélection se positionne implicitement sur la cellule située en haut et à gauche, vous devez définir les propriétés *Row* et *Column* au moment de la construction initiale du calendrier ainsi qu'à chaque changement de date.

Pour positionner la sélection dans la cellule du jour en cours, modifiez la méthode *UpdateCalendar* pour définir *Row* et *Column* avant d'appeler *Refresh* :

```
procedure TSampleCalendar.UpdateCalendar;
begin
  if FDate <> 0 then
  begin
    ⋮ { existing statements to set FMonthOffset }
    Row := (ADay - FMonthOffset) div 7 + 1;
    Col := (ADay - FMonthOffset) mod 7;
  end;
  Refresh; { déjà ici }
end;
```

Notez que vous réutilisez la variable *ADay* précédemment définie lors du décodage de la date.

Navigation de mois en mois et d'année en année

Les propriétés sont particulièrement utiles pour manipuler les composants, en particulier lors de la conception. Mais lorsque des manipulations fréquentes ou instinctives font intervenir plusieurs propriétés, il paraît judicieux de fournir des méthodes pour les gérer. Le passage au "mois suivant" dans notre calendrier est un exemple de ce type. Le bouclage sur les douze mois avec l'incrémentation de l'année est à la fois une caractéristique simple et commode pour le programmeur qui utilise le composant.

Le seul inconvénient à l'encapsulation des manipulations les plus fréquentes sous la forme de méthodes est le suivant : les méthodes ne sont accessibles qu'à l'exécution. Néanmoins, de telles manipulations ne sont fastidieuses que lorsqu'elles sont souvent répétées, ce qui est rarement le cas au moment de la conception.

S'agissant du calendrier, ajoutez les quatre méthodes suivantes pour gérer le passage de mois en mois et d'année en année. Chacune de ces méthodes utilise la fonction *IncMonth* de façon légèrement différente pour incrémenter ou décrémenter *CalendarDate* de mois en mois ou d'année en année. Après avoir incrémenté ou décrémenté *CalendarDate*, décodez la valeur de la date pour remplir les propriétés Year, Month et Day avec les nouvelles valeurs correspondantes .

```
procedure TCalendar.NextMonth;
begin
  DecodeDate(IncMonth(CalendarDate, 1), Year, Month, Day);
end;

procedure TCalendar.PrevMonth;
begin
  DecodeDate(IncMonth(CalendarDate, -1), Year, Month, Day);
end;

procedure TCalendar.NextYear;
begin
  DecodeDate(IncMonth(CalendarDate, 12), Year, Month, Day);
end;

procedure TCalendar.PrevYear;
begin
  DecodeDate(CalendarDate, -12), Year, Month, Day);
end;
```

N'oubliez pas d'ajouter les déclarations des nouvelles méthodes à la déclaration de la classe.

Désormais, si vous créez une application qui utilise le composant calendrier, vous pourrez facilement implémenter le passage de mois en mois ou d'année en année.

Navigation de jour en jour

A l'intérieur d'un même mois, il existe deux moyens évidents pour naviguer parmi les jours. Le premier consiste à utiliser les touches de direction et le deuxième à répondre aux clics de la souris. Le composant grille standard les gère tous les deux indistinctement en tant que clics de souris. Autrement dit, le déplacement avec les touches de direction est pris en compte comme un clic sur une cellule adjacente.

Le processus de navigation de jour en jour comprend :

- Déplacement de la sélection
- Fourniture d'un événement OnChange
- Exclusion des cellules vides

Déplacement de la sélection

Le comportement reçu en héritage d'une grille gère le déplacement de la sélection en réponse aux touches de direction enfoncées ou aux clics de souris. Pour modifier le jour sélectionné, vous devez modifier le comportement implicite.

Pour gérer les déplacements à l'intérieur du calendrier, vous devez surcharger la méthode *Click* de la grille.

Lorsque vous surchargez une méthode telle que *Click*, en dépendance étroite avec les interactions de l'utilisateur, vous devez pratiquement toujours inclure un appel à la méthode reçue en héritage pour ne pas perdre le comportement standard.

Le code suivant est une méthode *Click* surchargée pour la grille calendrier. N'oubliez pas d'ajouter la déclaration de *Click* à *TSampleCalendar*, en incluant après la directive **override**.

```
procedure TSampleCalendar.Click;
var
  TempDay: Integer;
begin
  inherited Click;                          { n'oubliez pas d'appeler la méthode héritée ! }
  TempDay := DayNum(Col, Row);         { récupère le numéro du jour de la cellule cliquée }
  if TempDay <> -1 then Day := TempDay;                  { change le jour s'il est valide }
end;
```

Fourniture d'un événement OnChange

Les utilisateurs de votre calendrier ont maintenant la possibilité de changer la date. Il paraît donc judicieux de répondre à ces changements.

Ajoutez un événement *OnChange* à *TSampleCalendar*.

1 Déclarez l'événement ainsi qu'un champ pour le stocker et une méthode virtuelle pour l'appeler :

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FOnChange: TNotifyEvent;
  protected
    procedure Change; dynamic;
  ⋮
  published
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
  ⋮
```

2 Ecrivez la méthode *Change* :

```
procedure TSampleCalendar.Change;
begin
  if Assigned(FOnChange) then FOnChange(Self);
end;
```

3 Ajoutez les instructions appelant *Change* à la fin des méthodes *SetCalendarDate* et *SetDateElement* :

```
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;
  UpdateCalendar;
  Change;                                              { seule nouvelle instruction }
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
    ⋮                                { instructions définissant les valeurs des éléments }
    FDate := EncodeDate(AYear, AMonth, ADay);
    UpdateCalendar;
    Change;                                                  { ceci est nouveau }
  end;
end;
```

Les applications qui utilisent le composant calendrier peuvent maintenant répondre aux changements en associant des gestionnaires à l'événement *OnChange*.

Exclusion des cellules vides

Tel qu'il est actuellement, le calendrier déplace la sélection vers une cellule vide sans changer la date. Il devient intéressant d'empêcher la sélection des cellules vides.

Pour déterminer si une cellule est sélectionnable, vous devez surcharger la méthode *SelectCell* de la grille.

SelectCell est une fonction qui accepte deux paramètres ligne et colonne et qui renvoie une valeur booléenne indiquant si la cellule spécifiée est sélectionnable.

Vous pouvez surcharger *SelectCell* pour qu'elle renvoie *False* si la cellule ne contient pas une date valide :

```
function TSampleCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if DayNum(ACol, ARow) = -1 then Result := False        { -1 indique une date incorrecte }
  else Result := inherited SelectCell(ACol, ARow);     { sinon, utilise la valeur héritée }
end;
```

Désormais, si l'utilisateur clique sur une cellule vide ou tente de s'y déplacer à l'aide des touches de direction, le calendrier ne modifie pas la sélection en cours.

Chapitre

51

Contrôles orientés données

Lorsque vous souhaitez vous connecter avec des bases de données, travaillez avec les contrôles *orientés données.* C'est grâce à ces contrôles que l'application établit un lien avec une partie spécifique d'une base de données. Parmi les contrôles sensibles aux données de Delphi, citons les libellés, les boîtes de saisie, les boîtes liste, les boîtes à options, les contrôles de référence et les grilles. Vous avez également la possibilité de construire vos propres contrôles orientés données. Pour plus d'informations sur l'utilisation des contrôles orientés données, voir chapitre 15, "Utilisation de contrôles de données".

Il existe différents niveaux d'orientation données. Le plus élémentaire fonctionne en lecture seulement, permet de *scruter des données* et reflète l'état d'une base de données. L'orientation données modifiables, permettant de *modifier les données*, est plus complexe car l'utilisateur peut changer les valeurs stockées dans la base en manipulant le contrôle. Notez également que le degré d'implication de la base de données peut varier du cas le plus simple, un lien établi avec un seul champ, aux cas plus complexes faisant intervenir des contrôles à enregistrements multiples.

Ce chapitre illustre d'abord le cas le plus simple, en créant un contrôle en lecture simple qui est lié à un seul champ d'un ensemble de données. Le contrôle spécifique utilisé sera le calendrier *TSampleCalendar* créé dans le chapitre 50, "Personnalisation d'une grille". Vous pouvez aussi utiliser le contrôle calendrier standard de la page Exemples de la palette des composants, *TCalendar*.

Ce chapitre continue ensuite avec une explication sur la manière de faire d'un nouveau contrôle pour scruter les données un contrôle de modification des données.

Création d'un contrôle pour scruter les données

La création d'un contrôle calendrier orienté données, que ce soit un contrôle en lecture seulement ou un contrôle grâce auquel l'utilisateur peut changer les données sous-jacentes, fait intervenir les étapes suivantes :

- Création et recensement du composant
- Ajout du lien aux données
- Réponse aux changements de données

Création et recensement du composant

La création d'un composant débute toujours de la même façon. Vous créez une unité et vous recensez le composant avant de l'installer dans la palette des composants. Ce processus est décrit dans "Création d'un nouveau composant" à la page 40-9.

Pour notre exemple, suivez la procédure générale de création d'un composant en tenant compte des spécificités suivantes :

- Appelez l'unité du composant *DBCal*.
- Dérivez une nouvelle classe composant appelée *TDBCalendar*, dérivée du composant VCL *TSampleCalendar*. Le chapitre 50, "Personnalisation d'une grille", montre comment créer le composant *TSampleCalendar*.
- Recensez *TDBCalendar* dans la page Exemples de la palette des composants.

L'unité que vous obtenez doit ressembler à ceci :

```
unit DBCal;

interface

uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Grids, Calendar;

type
  TDBCalendar = class(TSampleCalendar)
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TDBCalendar]);
end;

end.
```

Vous pouvez à présent commencer à transformer le nouveau calendrier en scruteur de données.

Fonctionnement du contrôle en lecture seulement

Puisque votre calendrier scruteur de données ne fonctionnera qu'en lecture (par rapport aux données), il est opportun de rendre le contrôle lui-même accessible en lecture seulement. Ainsi, l'utilisateur ne s'attendra pas à voir répercuter dans la base de données une modification qu'il aurait apporté au contrôle.

Rendre le calendrier accessible en lecture seulement fait intervenir deux étapes :

- Ajout de la propriété ReadOnly.
- Autorisation des mises à jour nécessaires.

Si vous démarrez avec le composant *TCalendar* de la page Exemples de Delphi au lieu de *TSampleCalendar*, le contrôle a déjà une propriété *ReadOnly*. Vous pouvez donc ignorer ces étapes.

Ajout de la propriété ReadOnly

En ajoutant une propriété *ReadOnly*, vous fournissez le moyen de rendre le contrôle accessible en lecture seulement au moment de la conception. Si la valeur de cette propriété est *True*, toutes les cellules du contrôle perdront la capacité à être sélectionnées.

1 Ajoutez la déclaration de la propriété ainsi qu'une donnée membre **private** pour contenir la valeur :

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FReadOnly: Boolean;                                    { champ de stockage interne }
  public
    constructor Create(AOwner: TComponent); override;     { doit surcharger pour définir
                                                            // les valeurs par défaut }
  published
    property ReadOnly: Boolean read FReadOnly write FReadOnly default True;
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);               { appelez toujours le constructeur hérité ! }
  FReadOnly := True;                                  { définit la valeur par défaut }
end;
```

2 Surchargez la méthode *SelectCell* pour inhiber la sélection si le contrôle est accessible en lecture seulement. L'utilisation de *SelectCell* est expliquée dans "Exclusion des cellules vides" à la page 50-13.

```
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if FReadOnly then Result := False                  { sélection impossible si accès en
                                                                        // lecture seule }
  else Result := inherited SelectCell(ACol, ARow); { sinon, utilise la méthode héritée }
end;
```

N'oubliez pas d'ajouter la déclaration de *SelectCell* à la déclaration de la classe de *TDBCalendar*, et ajoutez la directive **override**.

Si vous ajoutez maintenant le calendrier à une fiche, vous vous rendez compte que le composant ignore les clics de souris et les frappes de touches. Et il ne met plus à jour la position de la sélection lorsque vous changez la date.

Autorisation des mises à jour nécessaires

Le calendrier accessible en lecture seulement utilise la méthode *SelectCell* pour effectuer toutes sortes de modifications, y compris l'affectation de valeurs aux propriétés *Row* et *Col*. La méthode *UpdateCalendar* définit *Row* et *Col* à chaque changement de date mais, dans la mesure où *SelectCell* n'autorise aucune modification, la position de la sélection reste inchangée, même si la date est modifiée.

Pour outrepasser cette interdiction de toute modification, vous devez ajouter au calendrier un indicateur booléen interne et n'autoriser les modifications que si la valeur de cet indicateur est *true* :

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FUpdating: Boolean;                                  { indicateur privé à usage interne }
  protected
    function SelectCell(ACol, ARow: Longint): Boolean; override;
  public
    procedure UpdateCalendar; override;                      { notez la directive override }
  end;
⋮
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if (not FUpdating) and FReadOnly then Result := False       { sélection possible si mise
                                                                          // à jour }
  else Result := inherited SelectCell(ACol, ARow);    { sinon, utilise la méthode héritée }
end;

procedure TDBCalendar.UpdateCalendar;
begin
  FUpdating := True;              { définit l'indicateur pour permettre les mises à jour }
  try
    inherited UpdateCalendar;                                  { mise à jour habituelle }
  finally
    FUpdating := False;                               { réinitialise toujours l'indicateur }
  end;
end;
```

Le calendrier n'autorise toujours pas les modifications directes de l'utilisateur mais les modifications de la date effectuées via les propriétés de date sont prises en compte. Vous disposez maintenant d'un contrôle en lecture seulement tout à fait opérationnel. Vous voilà prêt à ajouter les fonctionnalités servant à scruter les données.

Ajout du lien aux données

La connexion entre un contrôle et une base de données est gérée par une classe appelée *lien de données*. La classe lien de données, qui connecte un contrôle à un seul champ d'une base de données, est *TFieldDataLink* (VCL ou CLX). Il existe également des liens de données vers des tables entières.

Un contrôle orienté données est *propriétaire* de sa classe lien de données. Autrement dit, le contrôle est responsable de la construction et de la destruction du lien de données. Pour des détails sur la gestion des classes ayant un propriétaire, voir chapitre 49, "Création d'un composant graphique".

Pour établir un lien de données en tant que classe ayant un propriétaire, respectez ces étapes :

1 Déclaration du champ de classe de la classe

2 Déclaration des propriétés d'accès

3 Initialisation du lien de données

Déclaration du champ de classe

Un composant utilise un champ pour chacune des classes dont il est le propriétaire, comme cela est expliqué dans "Déclaration des champs de classe" à la page 49-6. Dans ce cas, le calendrier a besoin d'un champ de type *TFieldDataLink* pour son lien de données.

Déclarez un champ pour le lien de données du calendrier :

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FDataLink: TFieldDataLink;
  ⋮
  end;
```

Avant de compiler l'application, vous devez ajouter DB et DBCtrls à la clause **uses** de l'unité.

Déclaration des propriétés d'accès

Tout contrôle orienté données dispose d'une propriété *DataSource* indiquant la classe source de données qui fournit les données au contrôle. En outre, un contrôle qui accède à un champ unique a besoin d'une propriété *DataField* pour spécifier ce champ dans la source de données.

Contrairement aux propriétés d'accès des classes ayant un propriétaire que nous avons vues avec l'exemple du chapitre 49, "Création d'un composant graphique", ces propriétés d'accès ne donnent pas accès aux classes ayant un propriétaire elles-mêmes, mais plutôt aux propriétés correspondantes de la classe ayant un propriétaire. Autrement dit, vous allez créer des propriétés qui autorisent le contrôle et son lien de données à partager la même source et le même champ.

Déclarez les propriétés *DataSource* et *DataField* ainsi que leurs méthodes d'implémentation, puis écrivez ces méthodes en tant que simples "boîtes à lettres" vers les propriétés correspondantes de la classe lien de données :

Exemple de déclaration des propriétés d'accès

```
type
  TDBCalendar = class(TSampleCalendar)
  private                                  { les méthodes d'implémentation sont private }
    ...
    function GetDataField: string;                  { renvoie le nom du champ de données }
    function GetDataSource: TDataSource; { renvoie une référence sur la source de données }
    procedure SetDataField(const Value: string);     { affecte le nom du champ de données }
    procedure SetDataSource(Value: TDataSource); { affecte une nouvelle source de données }
  published                       { rend les propriétés accessibles lors de la conception }
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
⋮
function TDBCalendar.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

function TDBCalendar.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TDBCalendar.SetDataField(const Value: string);
begin
  FDataLink.FieldName := Value;
end;

procedure TDBCalendar.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
end;
```

Maintenant que sont établis les liens entre le calendrier et son lien de données, il reste une étape importante à franchir. Vous devez construire la classe lien de données au moment de la construction du contrôle calendrier et le détruire avant de détruire ce même contrôle.

Initialisation du lien de données

Un contrôle orienté données doit avoir accès à son lien de données pendant toute sa durée de vie, il doit donc construire l'objet lien de données dans son propre constructeur et le détruire avant de se détruire lui-même.

Surchargez les méthodes *Create* et *Destroy* du calendrier pour construire et détruire l'objet lien de données :

```
type
  TDBCalendar = class(TSampleCalendar)
  public                          { les constructeurs et destructeurs sont toujours publics }
```

```
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    ⋮
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  FDataLink := TFieldDataLink.Create;                { construit l'objet lien de données }
  FDataLink.Control := self;              { informe le lien de données sur le calendrier }
  FReadOnly := True;                                                    { existe déjà }
end;

destructor TDBCalendar.Destroy;
begin
  FDataLink.Free;          { détruit toujours d'abord les objets ayant un propriétaire... }
  inherited Destroy;                                { ...puis appelle le destructeur hérité }
end;
```

Vous avez maintenant un lien de données complet. Il vous reste à indiquer au contrôle les données qu'il doit lire dans le champ lié. La section suivante vous explique comment procéder.

Réponse aux changements de données

Lorsqu'un contrôle a un lien de données et les propriétés précisant la source et le champ des données, il doit répondre aux changements des données de ce champ provoqués soit par un déplacement vers un autre enregistrement, soit par une modification du champ.

Les classes lien de données ont toutes un événement intitulé *OnDataChange*. Lorsque la source de données indique un changement dans ses données, l'objet lien de données appelle le gestionnaire attaché à son événement *OnDataChange*.

Pour mettre à jour un contrôle en réponse à une modification des données, vous devez attacher un gestionnaire à l'événement *OnDataChange* du lien de données.

Dans notre exemple, vous allez ajouter une méthode au calendrier, puis la désigner comme gestionnaire de l'événement *OnDataChange* du lien de données.

Déclarez et implémentez *la méthode DataChange*, puis associez-la à l'événement *OnDataChange* dans le constructeur. Dans le destructeur, détachez le gestionnaire *OnDataChange* avant de détruire l'objet.

```
type
  TDBCalendar = class(TSampleCalendar)
  private { this is an internal detail, so make it private }
    procedure DataChange(Sender: TObject);       { doit avoir des paramètres corrects pour
                                                                          // l'événement }
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);             { appelle toujours d'abord le constructeur hérité }
  FReadOnly := True;                                                    { existe déjà }
```

```
  FDataLink := TFieldDataLink.Create;                { construit l'objet lien de données }
  FDataLink.OnDataChange := DataChange;           { attache le gestionnaire à l'événement }
end;

destructor TDBCalendar.Destroy;
begin
  FDataLink.OnDataChange := nil;      { détache le gestionnaire avant de détruire l'objet }
  FDataLink.Free;           { détruit toujous d'abord les objets ayant un propriétaire... }
  inherited Destroy;                               { ...puis appelle le destructeur hérité }
end;

procedure TDBCalendar.DataChange(Sender: TObject);
begin
  if FDataLink.Field = nil then                     { s'il n'y a pas de champ attribué... }
    CalendarDate := 0                                   { ...définit une date incorrecte }
  else CalendarDate := FDataLink.Field.AsDateTime;    { sinon, définit le calendrier par la
                                                                                  // date }
end;
```

Vous avez maintenant un contrôle de parcours des données.

Création d'un contrôle de modification de données

Lorsque vous créez un contrôle permettant de modifier les données, vous créez et recensez le composant puis lui ajoutez un lien de données, comme pour les contrôles permettant de scruter les données. Vous devez également répondre aux changements de données dans le champ sous-jacent, mais vous devez prendre en considération quelques points supplémentaires.

Par exemple, vous souhaitez sans doute que votre contrôle réponde aux événements clavier et souris. Votre contrôle doit répondre lorsque l'utilisateur change le contenu du contrôle. Lorsque l'utilisateur quitte le contrôle, les changements effectués dans le contrôle doivent être répercutés dans l'ensemble de données.

Le contrôle permettant la modification des données décrit ici est le même que le contrôle calendrier décrit dans la première partie de ce chapitre. Le contrôle est modifié de telle sorte qu'il permette l'édition en plus de la consultation des données du champ lié.

Voici les étapes à suivre pour modifier un contrôle existant et en faire un contrôle permettant la modification des données :

- Modification de la valeur par défaut de FReadOnly.
- Gestion des messages liés à la souris ou au clavier.
- Mise à jour de la classe lien de données sur un champ.
- Modification de la méthode Change.
- Mise à jour de l'ensemble de données.

Modification de la valeur par défaut de FReadOnly

Comme il s'agit d'un contrôle permettant la modification des données, la propriété *ReadOnly* doit être *False* par défaut. Pour qu'elle soit *False*, modifiez la valeur de *FReadOnly* dans le constructeur :

```
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  ⋮
  FReadOnly := False;  { définit la valeur par défaut }
  ⋮
end;
```

Gestion des messages liés à la souris ou au clavier

Lorsque l'utilisateur commence à se servir du contrôle, celui-ci reçoit de Windows les messages indiquant la manipulation de la souris (*WM_LBUTTONDOWN*, *WM_MBUTTONDOWN*, ou *WM_RBUTTONDOWN*), ou le message indiquant la manipulation du clavier (*WM_KEYDOWN*). En cas d'utilisation de la CLX, la notification provient du système d'exploitation sous la forme d'événements système. Pour permettre à un contrôle de répondre à ces messages, vous devez écrire les gestionnaires des réponses à ces messages.

- Réponse aux messages indiquant la manipulation de la souris
- Réponse aux messages indiquant la manipulation du clavier

Réponse aux messages indiquant la manipulation de la souris

Une méthode *MouseDown* est une méthode protégée de l'événement *OnMouseDown* d'un contrôle. Le contrôle lui-même appelle *MouseDown* en réponse au message Windows indiquant la manipulation de la souris. Lorsque vous surchargez la méthode *MouseDown héritée*, vous pouvez inclure du code apportant d'autres réponses en plus de l'appel à l'événement *OnMouseDown*.

Pour surcharger *MouseDown*, ajoutez la méthode *MouseDown* à la classe *TDBCalendar* :

```
type
  TDBCalendar = class(TSampleCalendar);
   ⋮
  protected
    procedure MouseDown(Button: TButton, Shift: TShiftState, X: Integer, Y: Integer);
      override;
   ⋮
  end;

procedure TDBCalendar.MouseDown(Button: TButton; Shift: TShiftState; X, Y: Integer);
var
  MyMouseDown: TMouseEvent;
begin
  if not ReadOnly and FDataLink.Edit then
    inherited MouseDown(Button, Shift, X, Y)
  else
```

```
  begin
    MyMouseDown := OnMouseDown;
    if Assigned(MyMouseDown then MyMouseDown(Self, Button, Shift, X, Y);
  end;
end;
```

Lorsque *MouseDown* répond à un message indiquant la manipulation de la souris, la méthode *MouseDown* héritée est appelée uniquement si la propriété *ReadOnly* du contrôle est *False* et si l'objet lien de données est en mode édition, c'est-à-dire si le champ peut être modifié. Si le champ ne peut être modifié, le code mis par le programmeur dans le gestionnaire de l'événement *OnMouseDown*, s'il en existe un, est exécuté.

Réponse aux messages indiquant la manipulation du clavier

Une méthode *KeyDown* est une méthode protégée de l'événement *OnKeyDown* d'un contrôle. Le contrôle lui-même appelle *KeyDown* en réponse au message Windows indiquant la manipulation du clavier. Lorsque vous surchargez la méthode *KeyDown héritée*, vous pouvez inclure le code qui apporte d'autres réponses en plus de l'appel à l'événement *OnKeyDown*.

Pour surcharger *KeyDown*, suivez ces instructions :

1 Ajoutez une méthode *KeyDown* à la classe *TDBCalendar* :

```
type
    TDBCalendar = class(TSampleCalendar);
     ⋮
    protected
      procedure KeyDown(var Key: Word; Shift: TShiftState; X: Integer; Y: Integer);
        override;
     ⋮
    end;
```

2 Implémentez la méthode *KeyDown* :

```
procedure KeyDown(var Key: Word; Shift: TShiftState);
var
  MyKeyDown: TKeyEvent;
begin
  if not ReadOnly and (Key in [VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT, VK_END,
    VK_HOME, VK_PRIOR, VK_NEXT]) and FDataLink.Edit then
    inherited KeyDown(Key, Shift)
  else
  begin
    MyKeyDown := OnKeyDown;
    if Assigned(MyKeyDown) then MyKeyDown(Self, Key, Shift);
  end;
end;
```

Lorsque *KeyDown* répond à un message indiquant la manipulation du clavier, la méthode *KeyDown* héritée est appelée uniquement si la propriété *ReadOnly* du contrôle vaut *False*, si la touche appuyée est une des touches de déplacement du curseur et si l'objet lien de données est en mode édition, c'est-à-dire si le champ peut être modifié. Si le champ ne peut être modifié ou si une autre touche a été

pressée, le code mis par le programmeur dans le gestionnaire de l'événement *OnKeyDown*, s'il en existe un, est exécuté.

Mise à jour de la classe lien de données sur un champ

Il existe deux types de modification des données :

- Le changement de la valeur d'un champ doit se répercuter dans le contrôle orienté données
- Le changement dans le contrôle orienté données doit se répercuter dans la valeur du champ

Le composant *TDBCalendar* a déjà une méthode *DataChange* qui gère les modifications de la valeur du champ dans l'ensemble de données en assignant cette valeur à la propriété *CalendarDate*. La méthode *DataChange* est le gestionnaire de l'événement *OnDataChange*. Ainsi le composant calendrier est capable de gérer le premier type de modification des données.

De manière semblable, la classe lien de données sur un champ a aussi un événement *OnUpdateData* qui se produit lorsque l'utilisateur modifie le contenu du contrôle orienté données. Le contrôle calendrier a une méthode *UpdateData* qui devient le gestionnaire de l'événement *OnUpdateData*. *UpdateData* assigne au champ lien de données la valeur modifiée dans le contrôle orienté données.

1 Pour répercuter dans la valeur du champ une modification effectuée sur la valeur du calendrier, ajoutez une méthode *UpdateData* à la section **private** du composant calendrier :

```
type
     TDBCalendar = class(TSampleCalendar);
     private
       procedure UpdateData(Sender: TObject);
      ⋮
     end;
```

2 Implémentez la méthode *UpdateData* :

```
procedure UpdateData(Sender: TObject);
   begin
     FDataLink.Field.AsDateTime := CalendarDate;   { définit le champ lien par la date du
calendrier }
   end;
```

3 Dans le constructeur de *TDBCalendar*, affectez la méthode *UpdateData* à l'événement *OnUpdateData* :

```
constructor TDBCalendar.Create(AOwner: TComponent);
   begin
     inherited Create(AOwner);
     FReadOnly := True;
     FDataLink := TFieldDataLink.Create;
     FDataLink.OnDataChange := DataChange;
     FDataLink.OnUpdateData := UpdateData;
   end;
```

Modification de la méthode Change

La méthode *Change* du *TDBCalendar* est appelée chaque fois qu'est définie une nouvelle valeur de date. *Change* appelle le gestionnaire de l'événement *OnChange*, s'il existe. L'utilisateur du composant peut écrire du code dans le gestionnaire de l'événement *OnChange* afin de répondre aux modifications de la date.

Lorsque la date du calendrier change, l'ensemble de données sous-jacent doit être averti de ce changement. Vous pouvez le faire en surchargeant la méthode *Change et en ajoutant une ligne de code de plus.* Voici les étapes à suivre :

1 Ajoutez une nouvelle méthode *Change* au composant *TDBCalendar* :

```
type
     TDBCalendar = class(TSampleCalendar);
     private
       procedure Change; override;
      ⋮
     end;
```

2 Ecrivez la méthode *Change*, appelant la méthode *Modified* qui informe l'ensemble de données que celles-ci ont changé, puis appelle la méthode *Change* héritée :

```
procedure TDBCalendar.Change;
begin
  FDataLink.Modified;                    { appelle la méthode Modified }
  inherited Change;                  { appelle la méthode Change héritée }
end;
```

Mise à jour de l'ensemble de données

A ce point, une modification dans le contrôle orienté données a changé les valeurs dans la classe du lien de données sur un champ. La dernière étape de la création d'un contrôle permettant la modification des données consiste à mettre à jour l'ensemble de données avec la nouvelle valeur. Cela doit se produire après que la personne ayant changé la valeur du contrôle quitte ce contrôle en cliquant à l'extérieur de celui-ci ou en appuyant sur la touche. Ce processus fonctionne de manière différente dans la VCL et la CLX.

VCL La VCL possède des ID de message définis pour les opérations sur les contrôles. Par exemple, le message *CM_EXIT* est envoyé au contrôle lorsque l'utilisateur quitte celui-ci. Vous pouvez écrire un gestionnaire qui réponde à ce message. Et ensuite, lorsque l'utilisateur quitte le contrôle, la méthode *CMExit*, gestionnaire du message *CM_EXIT*, répondra en mettant à jour l'enregistrement dans l'ensemble de données avec les valeurs modifiées dans la classe lien de données sur un champ. Pour plus d'informations sur les gestionnaires de messages, voir chapitre 46, "Gestion des messages".

Pour mettre à jour l'ensemble de données depuis un gestionnaire de message, suivez ces instructions :

1 Ajoutez le gestionnaire de message au composant *TDBCalendar* :

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure CMExit(var Message: TWMNoParams); message CM_EXIT;
   ⋮
  end;
```

2 Implémentez la méthode *CMExit* afin qu'elle ressemble à ceci :

```
procedure TDBCalendar.CMExit(var Message: TWMNoParams);
begin
  try
    FDataLink.UpdateRecord;  {indiquer au lien de données d'actualiser la base de données }
  except
    on Exception do SetFocus;                  { si échec, ne pas perdre la focalisation }
 end;
  inherited;
end;
```

CLX *Dans la CLX, TWidgetControl* possède une méthode *DoExit* protégée qui est appelée quand la focalisation en saisie quitte le contrôle. Cette méthode appelle le gestionnaire de l'événement *OnExit*. Vous pouvez redéfinir cette méthode pour actualiser l'enregistrement dans l'ensemble de données avant la génération du gestionnaire d'événement *OnExit*.

Pour mettre à jour l'ensemble de données quand l'utilisateur quitte le contrôle, suivez les étapes ci-après :

1 Ajoutez un overrride à la méthode *DoExit* du composant *TDBCalendar* :

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure DoExit; override;
   ⋮
  end;
```

2 Implémentez la méthode *DoExit* afin qu'elle ressemble à ceci :

```
procedure TDBCalendar.CMExit(var Message: TWMNoParams);
begin
  try
    FDataLink.UpdateRecord;  {indiquer au lien de données d'actualiser la base de données }
  except
    on Exception do SetFocus;                  { si échec, ne pas perdre la focalisation }
 end;
  inherited;                    { laisser la méthode héritée générer un événement OnExit }
end;
```

Chapitre

52

Transformation d'une boîte de dialogue en composant

Il est pratique de transformer une boîte de dialogue fréquemment sollicitée en un composant que vous pourrez ajouter dans la palette des composants. Ainsi, vos composants boîte de dialogue fonctionneront exactement comme ceux des boîtes de dialogue standard. L'objectif ici est de créer un composant simple qu'un utilisateur peut ajouter à un projet et dont il peut définir les propriétés lors de la conception.

La transformation d'une boîte de dialogue en composant nécessite les étapes suivantes :

1. Définition de l'interface du composant
2. Création et recensement du composant
3. Création de l'interface du composant
4. Test du composant

A l'exécution, le composant "enveloppe" de Delphi, associé à la boîte de dialogue crée et exécute celle-ci en lui transmettant les données spécifiées par l'utilisateur. Le composant boîte de dialogue est donc à la fois réutilisable et personnalisable.

Dans ce chapitre, vous allez voir comment créer un composant enveloppe autour de la fiche générique A propos de... disponible dans le référentiel d'objets de Delphi.

Remarque Copiez les fichiers ABOUT.PAS et ABOUT.DFM dans votre répertoire de travail.

Il n'y a pas grand chose à dire concernant la conception de la boîte de dialogue enveloppée par un composant. Dans un tel contexte, n'importe quelle fiche peut fonctionner comme une boîte de dialogue.

Définition de l'interface du composant

Avant de créer le composant pour votre boîte de dialogue, vous devez décider de la façon dont il sera utilisé par les développeurs. Vous devez créer une interface entre votre boîte de dialogue et les applications qui l'utilisent.

Par exemple, considérons les propriétés des composants associés aux boîtes de dialogue standard. Elles autorisent le développeur à définir l'état initial de la boîte de dialogue, tel que le titre ou le paramétrage initial des contrôles, et renvoient toutes les informations nécessaires lorsque la boîte de dialogue se ferme. Les seules interactions directes ne se produisent qu'avec les propriétés du composant enveloppe et pas avec les contrôles individuels de la boîte de dialogue.

L'interface doit donc contenir suffisamment d'informations pour que la fiche boîte de dialogue s'affiche selon les indications du développeur et qu'elle renvoie les informations nécessaires à l'application. Les propriétés du composant enveloppe peuvent être vues comme les données permanentes d'une boîte de dialogue transitoire.

Dans le cas de votre boîte A propos de, aucune information n'est renvoyée. Les propriétés de l'enveloppe contiennent donc uniquement les informations nécessaires pour afficher correctement la boîte A propos de. Puisqu'il y a quatre champs distincts dans cette boîte sur lesquels l'application peut agir, il vous faut fournir quatre propriétés de type chaîne pour les paramétrer.

Création et recensement du composant

La création d'un composant débute toujours de la même façon. Vous créez une unité et vous recensez le composant avant de l'installer dans la palette des composants. Ce processus est décrit dans "Création d'un nouveau composant" à la page 40-9.

Pour notre exemple, suivez la procédure générale de création d'un composant en tenant compte des spécificités suivantes :

- Nommez l'unité du composant *AboutDlg*.
- Dérivez un nouveau type de composant appelé *TAboutBoxDlg*, descendant de *TComponent*.
- Recensez *TAboutBoxDlg* sur la page Exemples de la palette des composants.

L'unité que vous obtenez doit ressembler à ceci :

```
unit AboutDlg;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
  TAboutBoxDlg = class(TComponent)
  end;
procedure Register;
implementation
```

```
procedure Register;
begin
  RegisterComponents('Samples', [TAboutBoxDlg]);
end;
end.
```

Pour l'instant, le nouveau composant possède uniquement les fonctionnalités intégrées à *TComponent*. C'est le composant non visuel le plus simple. Dans la section suivante, vous allez créer l'interface entre le composant et la boîte de dialogue.

Création de l'interface du composant

Voici les étapes nécessaires à la création de l'interface du composant :

1. Inclusion de l'unité de la fiche
2. Ajout des propriétés de l'interface
3. Ajout de la méthode Execute

Inclusion de l'unité de la fiche

Pour que votre composant enveloppe puisse initialiser et afficher la boîte de dialogue enveloppée, vous devez ajouter les fichiers de l'unité de la fiche dans la clause **uses** de l'unité du composant enveloppe.

Ajoutez *About* à la clause **uses** de l'unité *AboutDlg*.

La clause **uses** ressemble maintenant à ceci :

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms,
  About;
```

L'unité fiche déclare toujours une instance de la classe de la fiche. Dans le cas de la boîte A propos de, la classe de la fiche est *TAboutBox*, et l'unité *About* doit inclure la déclaration suivante :

```
var
  AboutBox: TAboutBox;
```

Ainsi en ajoutant *About* à la clause **uses**, vous rendez disponible *AboutBox* au composant enveloppe.

Ajout des propriétés de l'interface

Avant de poursuivre, vous devez déterminer les propriétés de votre composant enveloppe nécessaires pour permettre aux développeurs d'utiliser votre boîte de dialogue en tant que composant dans leurs applications. Puis, ajoutez les déclarations de ces propriétés à la déclaration de classe du composant.

Les propriétés d'un composant enveloppe sont sensiblement plus simples à écrire que celles d'un composant standard. Souvenez-vous que vous ne faites que créer des données permanentes que l'enveloppe et la boîte de dialogue peuvent

échanger. En définissant ces données sous la forme de propriétés, vous donnez aux développeurs la possibilité de définir des données au moment de la conception qui, lors de l'exécution, seront transmises par l'enveloppe à la boîte de dialogue.

La déclaration d'une propriété d'interface nécessite deux ajouts à la déclaration de classe du composant :

- Un champ de classe privé qui est une variable utilisée par l'enveloppe pour stocker la valeur de la propriété.
- La déclaration **published** de la propriété elle-même qui indique son nom et le champ à utiliser pour le stockage.

De telles propriétés d'interface n'ont pas besoin de méthodes d'accès. Elles accèdent directement aux données stockées. Par convention, le champ qui stocke la valeur de la propriété porte le même nom que la propriété, mais précédé de la lettre *F*. Le champ et la propriété *doivent avoir le même* type.

Par exemple, la déclaration d'une propriété d'interface de type entier appelée *Year*, est la suivante :

```
type
  TMyWrapper = class(TComponent)
  private
    FYear: Integer;                { donnée membre pour les données de la propriété Year }
  published
    property Year: Integer read FYear write FYear;         { la propriété et son stockage }
  end;
```

S'agissant de votre boîte A propos de, vous devez disposer de quatre propriétés de type string pour le nom du produit, les informations de version, les informations de copyright et les commentaires éventuels.

```
type
  TAboutBoxDlg = class(TComponent)
  private
    FProductName, FVersion, FCopyright, FComments: string;  { déclare les données membres }
  published
    property ProductName: string read FProductName write FProductName;
    property Version: string read FVersion write FVersion;
    property Copyright: string read FCopyright write FCopyright;
    property Comments: string read FComments write FComments;
  end;
```

Si vous installez votre composant dans la palette des composants et si vous le placez dans une fiche, vous pourrez définir les propriétés, et ces valeurs apparaîtront de façon permanente dans la fiche. Ainsi, lors de l'exécution de la boîte de dialogue qu'il enveloppe, le composant pourra les utiliser.

Ajout de la méthode Execute

Il vous reste à définir dans l'interface du composant les moyens d'ouvrir la boîte de dialogue et de récupérer un résultat lorsqu'elle se ferme. Comme pour les composants des boîtes de dialogue standard, vous utiliserez une fonction

booléenne appelée *Execute* qui renvoie *True* si l'utilisateur clique sur OK, ou *False* s'il annule la boîte de dialogue.

La déclaration de la méthode *Execute* ressemble toujours à ceci :

```
type
  TMyWrapper = class(TComponent)
  public
    function Execute: Boolean;
  end;
```

L'implémentation minimale de la méthode *Execute* doit construire la fiche de la boîte de dialogue, puis afficher celle-ci en tant que boîte de dialogue modale avant de renvoyer *True* ou *False*, selon la valeur renvoyée par *ShowModal*.

Voici l'implémentation minimale de la méthode *Execute* pour une fiche boîte de dialogue de type *TMyDialogBox* :

```
function TMyWrapper.Execute: Boolean;
begin
  DialogBox := TMyDialogBox.Create(Application);                    { construit la fiche }
  try
    Result := (DialogBox.ShowModal = IDOK);        { exécute; définit le résultat selon la
                                                                      //façon de fermer }
  finally
    DialogBox.Free;                                                 { restitue la fiche }
  end;
end;
```

Notez l'utilisation d'un bloc **try..finally** qui vérifie que l'application restitue l'objet boîte de dialogue même si une exception se produit. En général, lorsque vous construisez un objet de cette manière, vous devez utiliser un bloc **try..finally** pour protéger le bloc de code et être sûr que l'application libère toutes les ressources qu'elle alloue.

En pratique, il y a davantage de code dans le bloc **try..finally**. Plus spécifiquement, avant l'appel à *ShowModal*, le composant enveloppe doit définir certaines propriétés de la boîte de dialogue en fonction de ses propres propriétés d'interface. A l'inverse, quand *ShowModal* rend la main, le composant enveloppe définit certaines de ses propriétés d'interface en fonction du résultat de l'exécution de la boîte de dialogue.

Dans le cas de la boîte A propos de, vous devez utiliser les quatre propriétés d'interface du composant enveloppe pour définir le contenu des libellés de la boîte de dialogue. Comme cette dernière ne renvoie aucune information à l'application, il n'y a rien de particulier à faire après l'appel à *ShowModal*.

Dans la partie publique de la classe *TAboutDlg*, ajoutez la déclaration pour la méthode *Execute* :

```
type
  TAboutDlg = class(TComponent)
public
  function Execute: Boolean;
end;

function TAboutBoxDlg.Execute: Boolean;
```

```
begin
  AboutBox := TAboutBox.Create(Application);            { construit la boîte A propos de }
  try
    if ProductName = '' then                          { si le nom du produit est vide... }
      ProductName := Application.Title; { ...utilise à la place le titre de l'application }
    AboutBox.ProductName.Caption := ProductName;              { copie le nom du produit }
    AboutBox.Version.Caption := Version;                   { copie les infos de version }
    AboutBox.Copyright.Caption := Copyright;              { copie les infos de copyright }
    AboutBox.Comments.Caption := Comments;                     { copie les commentaires }
    AboutBox.Caption := 'About ' + ProductName;           { définit le titre de la boîte }
    with AboutBox do begin
      ProgramIcon.Picture.Graphic := Application.Icon;                  { copie l'icône }
      Result := (ShowModal = IDOK);                     { exécute et définit le résultat }
    end;
  finally
    AboutBox.Free;                           { restitue la boîte de dialogue A propos de }
  end;
end;
```

Test du composant

Une fois le composant boîte de dialogue installé, vous pouvez l'utiliser comme n'importe quelle autre boîte de dialogue commune, en le plaçant sur une fiche et en l'exécutant. Un moyen rapide de vérifier le fonctionnement de la boîte A propos de consiste à ajouter un bouton de commande dans une fiche et à exécuter la boîte de dialogue lorsque l'utilisateur clique sur ce bouton.

Par exemple, si vous avez créé une boîte de dialogue A propos de, et si vous l'avez ajouté à la palette des composants, vous pouvez tester son fonctionnement en suivant les étapes ci-dessous :

1. Créez un nouveau projet.
2. Placez un composant A propos de dans la fiche principale.
3. Placez un bouton de commande dans la fiche.
4. Double-cliquez sur le bouton de commande pour créer un gestionnaire d'événements vide.
5. Dans le gestionnaire d'événements, entrez la ligne de code suivante :

   ```
   AboutBoxDlg1.Execute;
   ```

6. Exécutez l'application.

Lorsque la fiche principale apparaît, cliquez sur le bouton de commande. La boîte A propos de s'affiche avec l'icône projet par défaut et Project1 comme titre. Choisissez OK pour fermer la boîte de dialogue.

Vous pouvez pousser plus loin le test du fonctionnement du composant en définissant les différentes propriétés du composant A propos de et en exécutant une nouvelle fois l'application.

Index

Symboles

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z