31

Object persistence and databases

Executing an object-oriented application means creating and manipulating a certain number of objects. What happens to these objects when the current execution terminates? *Transient* objects will disappear with the current session; but many applications also need *persistent* objects, which will stay around from session to session. Persistent objects may need to be shared by several applications, raising the need for *databases*.

In this overview of persistence issues and solutions we will examine the three approaches that O-O developers have at their disposal for manipulating persistent objects. They can rely on **persistence mechanisms** from the programming language and development environment to get object structures to and from permanent storage. They can combine object technology with databases of the most commonly available kind (not O-O): **relational databases**. Or they can use one of the newer **object-oriented database systems**, which undertake to transpose to databases the basic ideas of object technology.

This chapter describes these techniques in turn, providing an overview of the technology of O-O databases with emphasis on two of the best-known products. It ends with a more futuristic discussion of the fate of database ideas in an O-O context.

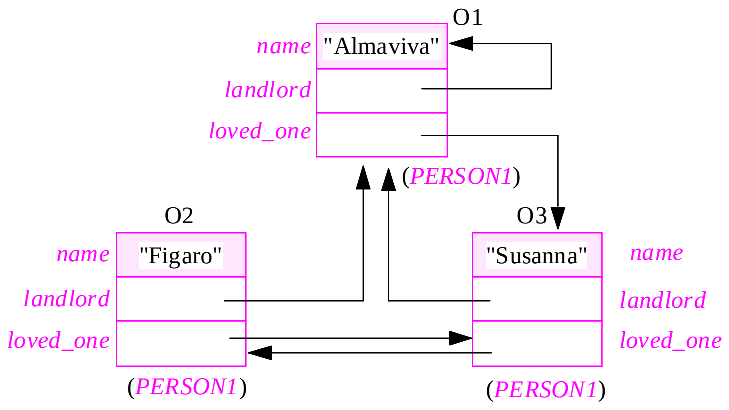
31.1 PERSISTENCE FROM THE LANGUAGE

For many persistence needs it suffices to have, associated with the development environment, a set of mechanisms for storing objects in files and retrieving them from files. For simple objects such as integers and characters, we can use input-output facilities similar to those of traditional programming.

Storing and retrieving object structures

See "Deep storage: a first view of persistence", page 250.

As soon as composite objects enter the picture, it is not sufficient to store and retrieve individual objects since they may contain references to other objects, and an object deprived of its dependents would be inconsistent. This observation led us in an earlier chapter to the *Persistence Closure* principle, stating that any storage and retrieval mechanism must handle, together with an object, all its direct and indirect dependents. The following figure served to illustrate the issue:



The need for persistence closure

The Persistence Closure principle stated that any mechanism that stores O1 must also store all the objects to which it refers, directly or indirectly; otherwise when you retrieve the structure you would get a meaningless value ("*dangling reference*") in the *loved_one* field for O1.

We saw the mechanisms of class *STORABLE* which provide the corresponding facilities: *store* to store an object structure and *retrieved* to access it back. This is a precious mechanism, whose presence in an O-O environment is by itself a major advantage over traditional environments. The earlier discussion gave a typical example of use: implementing the SAVE facility of an editor. Here is another, from ISE's own practice. Our compiler performs several passes on representations of the software text. The first pass creates an internal representation, known as an Abstract Syntax Tree (AST). Roughly speaking, the task of the subsequent passes is to add more and more semantic information to the AST (to "decorate the tree") until there is enough to generate the compiler's target code. Each pass finishes by a *store*; the next pass starts by retrieving the AST through *retrieved*.

The *STORABLE* mechanism works not only on files but also on network connections such as sockets; it indeed lies at the basis of the *Net* client-server library.

Storable format variants

Procedure *store* has several variants. One, *basic_store*, stores objects to be retrieved by the same system running on the same machine architecture, as part of the same execution or of a later one. These assumptions make it possible to use the most compact format possible for representing objects.

Another variant, *independent_store*, removes all these assumptions; the object representation is platform-independent and system-independent. It consequently takes a little more space, since it must use a portable data representation for floating-point and other numerical values, and must include some elementary information about the classes of the system. But it is precious for client-server systems, which must exchange potentially large and complex collections of objects among machines of widely different architectures, running entirely different systems. For example a workstation server and a PC client can run two different applications and communicate through the *Net* library, with the server application performing the fundamental computations and the client application taking care of the user interface thanks to a graphical library such as *Vision*.

Note that the storing part is the only one to require several procedures — *basic_store*, *independent_store*. Even though the implementation of retrieval is different for each format, you will always use a single feature *retrieved*, whose implementation will detect the format actually used by the file or network data being retrieved, and will automatically apply the appropriate retrieval algorithm.

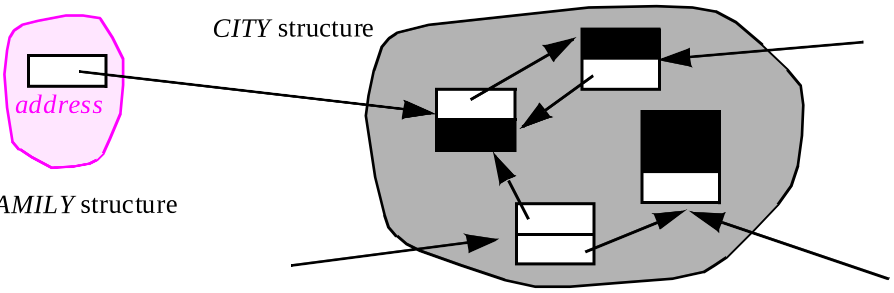
31.2 BEYOND PERSISTENCE CLOSURE

The Persistence Closure principle is, in theory, applicable to all forms of persistence. It makes it possible, as we saw, to preserve the consistency of objects stored and retrieved.

In some practical cases, however, you may need to adapt the data structure before letting it be applied by mechanisms such as *STORABLE* or the O-O database tools reviewed later in this chapter. Otherwise you may end up storing more than you want.

The problem arises in particular because of shared structures, as in this setup:

Small structure with reference to big shared structure



A relatively small data structure needs to be archived. Because it contains one or more references to a large shared structure, the Persistence Closure principle requires archiving that structure too. In some cases you may not want this. For example, as illustrated by the figure, you could be doing some genealogical research, or other processing on objects representing persons; a person object might, through an *address* field, reference a much bigger set of objects representing geographical information. A similar situation occurs in ISE's *ArchiText* product, which enables users to manipulate structured documents, such as programs or specifications. Each document, like the *FAMILY* structure in the figure, contains a reference to a structure representing the underlying grammar, playing the role of the *CITY* structure; we may want to store a document but not the grammar, which already exists elsewhere and may be shared by many documents.

In such cases you may want to “cut out” the references to the shared structure before storing the referring structure. This is, however, a delicate process. First, you must as always make sure that at retrieval time the objects will still be consistent — satisfy their invariants. But there is also a practical problem: to avoid complication and errors, you do not really want to modify the original structure; only in the stored version should references be cut out.

See “Deferred classes as partial implementations: the notion of behavior class”, page 504.

Once again the techniques of object-oriented software construction provide an elegant solution, based on the ideas of *behavior class* reviewed in the discussion of inheritance. One of the versions of the storing procedure, *custom_independent_store*, has the same effect as *independent_store* by default, but also lets any descendant of a library class *ACTIONABLE* redefine a number of procedures which do nothing by default, such as *pre_store* which will be executed just before an object is stored and *post_store* which will be executed after. So you can for example have *pre_store* perform

preserve; *address* := *Void*

where *preserve*, also a feature of *ACTIONABLE*, copies the object safely somewhere. Then *post_action* will perform a call to

restore

which restores the object from the preserved copy.

For this common case it is in fact possible to obtain the same effect through a call of the form

store_ignore ("*address*")

where *ignore* takes a field name as argument. Since the implementation of *store_ignore* may simply skip the field, avoiding the two-way copy of *preserve* and *restore*, it will be more efficient in this case, but the *pre_store*-*post_store* mechanism is more general, allowing any actions before and after storage. Again, you must make sure that these actions will not adversely affect the objects.

You may in fact use a similar mechanism to remove an inconsistency problem arising at retrieval time; it suffices to redefine the procedure *post_retrieve* which will be executed just before the retrieved object rejoins the community of approved objects. For example an application might redefine *post_retrieve*, in the appropriate class inheriting from *ACTIONABLE*, to execute something like

address := *my_city_structure*.*address_value* (…)

hence making the object presentable again before it has had the opportunity to violate its class invariant or any informal consistency constraint.

There are clearly some rules associated with the *ACTIONABLE* mechanism; in particular, *pre_store* must not perform any change of the data structure unless *post_store* corrects it immediately thereafter. You must also make sure that *post_retrieve* will perform the necessary actions (often the same as those of *post_store*) to correct any inconsistency introduced into the stored structure by *pre_store*. Used under these rules, the mechanism lets you remain faithful to the spirit of the Persistent Closure principle while making its application more flexible.

31.3 SCHEMA EVOLUTION

A general issue arises in all approaches to O-O persistence. Classes can change. What if you change a class of which instances exist somewhere in a persistent store? This is known as the schema evolution problem.

The word *schema* comes from the relational database world, where it describes the architecture of a database: its set of relations (as defined in the next section) with, for every relation, what we would call its type — number of fields and type of each. In an O-O context the schema will also be the set of types, given here by the classes.

Although some development environments and database systems have provided interesting tools for O-O schema evolution, none has yet provided a fully satisfactory solution. Let us define the components of a comprehensive approach.

An object's **generating class** (*or* **generator**) *is the class of which it is a direct instance. See "Basic form", page 219.*

Some precise terminology will be useful. **Schema evolution** occurs if at least one class used by a system that attempts to retrieve some objects (the **retrieving system**) differs from its counterpart in the system that stored these objects (the **storing system**). Object retrieval mismatch, or just **object mismatch** for short, occurs when the retrieving system actually retrieves a particular object whose own generating class was different in the storing system. Object mismatch is an individual consequence, for one particular object, of the general phenomenon of schema evolution for one or more classes.

Remember that in spite of the terms "storing system" and "retrieving system" this whole discussion is applicable not only to storage and retrieval using files or databases, but also to object transmission over a network, as with the *Net* library. In such a case the more accurate terms would be "sending system" and "receiving system".

Exercise E31.1, page 1062, asks you to study the consequences of removing this assumption.

To keep the discussion simple, we will make the usual assumption that a software system does not change while it is being executed. This means in particular that all the instances of a class stored by a particular system execution refer to the same version of the class; so at retrieval time either all of them will produce an object mismatch, or none of them will. This assumption is not too restrictive; note in particular that it does not rule out the case of a database that contains instances of many different versions of the same class, produced by different system executions.

Naïve approaches

We can rule out two extreme approaches to schema evolution:

- You might be tempted to forsake previously stored objects (schema *revolution*!). The developers of the new application will like the idea, which makes their life so much easier. But the *users* of the application will not be amused.
- You may offer a migration path from old format to new, requiring a one-time, en masse conversion of old objects. Although this solution may be acceptable in some cases, it will not do for a large persistent store or one that must be available continuously.

What we really need is a way to convert old objects **on the fly** as they are retrieved or updated. This is the most general solution, and the only one considered in the rest of this discussion.

If you happen to need en-masse conversion, an on-the-fly mechanism will trivially let you do it: simply write a small system that retrieves all the existing objects using the new classes, applying on-the-fly conversion as needed, and stores everything.

On-the-fly object conversion

The mechanics of on-the-fly conversion can be tricky; we must be particularly careful to get the details right, lest we end up with corrupted objects and corrupted databases.

First, an application that retrieves an object and has a different version of its generating class may not have the rights to update the stored objects, which may be just as well since *other* applications may still use the old version. This is not, however, a new problem. What counts is that the objects manipulated by the application be consistent with their own class descriptions; an on-the-fly conversion mechanism will ensure this property. Whether to write back the converted object to the database is a separate question — a classical question of access privilege, which arises as soon as several applications, or even several sessions of the same application, can access the same persistent data. Database systems, object-oriented or not, have proposed various solutions

Regardless of write-back aspects, the newer and perhaps more challenging problem is how each application will deal with an obsolete object. Schema evolution involves three separate issues — detection, notification and correction:

- **Detection** is the task of catching object mismatches (cases in which a retrieved object is obsolete) at retrieval time.
- **Notification** is the task of making the retrieving system aware of the object mismatch, so that it will be able to react appropriately, rather than continuing with an inconsistent object (a likely cause of major trouble ahead!).
- **Correction** is the task, for the retrieving system, of bringing the mismatched object to a consistent state that will make it a correct instance of the new version of its class — a citizen, or at least a permanent resident, of its system of adoption.

All three problems are delicate. Fortunately, it is possible to address them separately.

Detection

We can define two general categories of detection policy: **nominal** and **structural**.

In both cases the problem is to detect a mismatch between two versions of an object's generating class: the version used by the system that stored the object, and the version used by the system which retrieves it.

In the nominal approach, each class version is identified by a version name. This assumes some kind of registration mechanism, which may have two variants:

- If you are using a configuration management system, you can register each new version of the class and get a version name in return (or specify the version name yourself).

- More automatic schemes are possible, similar to the automatic identification facility of Microsoft's OLE 2, or the techniques used to assign "dynamic IP addresses" to computers on the Internet (for example a laptop that you plug in temporarily into a new network). These techniques are based on random number assignments, with numbers so large as to make the likelihood of a clash infinitesimal.

Either solution requires some kind of central registry. If you want to avoid the resulting hassle, you will have to rely on the structural approach. The idea here is to associate with each class version a **class descriptor** deduced from the actual structure of the class, as defined by the class declaration, and to make sure that whenever a persistent mechanism stores objects it *also stores the associated class descriptors*. (Of course if you store many instances of a class you will only need to store one copy of the class descriptor.) Then the detection mechanism is simple: just compare the class descriptor of each retrieved object with the new class descriptor. If they are different, you have an object mismatch.

What goes into a class descriptor? There is some flexibility; the answer is a tradeoff between efficiency and reliability. For efficiency, you will not want to waste too much space for keeping class information in the stored structure, or too much time for comparing descriptors at retrieval time; but for reliability you will want to minimize the risk of missing an object mismatch — of treating a retrieved object as up-to-date if it is in fact obsolete. Here are various possible strategies:

C1 • At one extreme, the class descriptor could just be the class name. This is generally insufficient: if the generator of an object in the storing system has the same name as a class in the retrieving system, we will accept the object even though the two classes may be totally incompatible. Trouble will inevitably follow.

C2 • At the other extreme, we might use as class descriptor the entire class text — perhaps not as a string but in an appropriate internal form (abstract syntax tree). This is clearly the worst solution for efficiency, both in space occupation and in descriptor comparison time. But it may not even be right for reliability, since some class changes are harmless. Assume for example the new class text has added a routine, but has not changed any attribute or invariant clause. Then nothing bad can happen if we consider a retrieved object up-to-date; but if we detect an object mismatch we may cause some unwarranted trouble (such as an exception) in the retrieving system.

C3 • A more realistic approach is to make the class descriptor include the class name and the list of its attributes, each characterized by its name and its type. As compared to the nominal approach, there is still the risk that two completely different classes might have both the same name and the same attributes, but (unlike in case C1) such chance clashes are extremely unlikely to happen in practice.

C4 • A variation on C3 would include not just the attribute list but also the whole class invariant. With the invariant you should be assured that the addition or removal of a routine, which will not yield a detected object mismatch, is harmless, since if it changed the semantics of the class it would affect the invariant.

C3 is the minimum reasonable policy, and in usual cases seems a good tradeoff, at least to start.

Notification

What should happen when the detection mechanism, nominal or structural, has caught an object mismatch?

We want the retrieving system to know, so that it will be able to take the appropriate correction actions. A library mechanism will address the problem. Class *GENERAL* (ancestor of all classes) must include a procedure

```
correct_mismatch is
    do
        …See full version below …
    end
```

correct in this procedure name is not an adjective but a verb, as in "Correct this mismatch, fast!". See "Grammatical categories", page 881.

with the rule that any detection of an object mismatch will cause a call to *correct_mismatch* on the temporarily retrieved version of the object. Any class can redefine the default version of *correct_mismatch*; like a creation procedure, and like any redefinition of the default exception handling procedure *default_rescue*, any redefinition of *correct_mismatch* must ensure the invariant of the class.

What should the default version of *correct_mismatch* do? It may be tempting, in the name of unobtrusiveness, to give it an empty body. But this is not appropriate, since it would mean that by default object retrieval mismatches will be ignored — leading to all kinds of possible abnormal behavior. The better global default is to raise an exception:

```
correct_mismatch is
            -- Handle object retrieval mismatch.
    do
        raise_mismatch_exception
    end
```

where the procedure called in the body does what its name suggests. It might cause some unexpected exceptions, but this is better than letting mismatches go through undetected. A project that wants to override this default behavior, for example to execute a null instruction rather than raise an exception, can always redefine *correct_mismatch*, at its own risk, in class *ANY*. (As you will remember, developer-defined classes inherit from *GENERAL* not directly but through *ANY*, which a project or installation can customize.)

"THE GLOBAL INHERITANCE STRUCTURE", page 580.

For more flexibility, there is also a feature *mismatch_information* of type *ANY*, defined as a once function, and a procedure *set_mismatch_information* (*info*: *ANY*) which resets its value. This makes it possible to provide *correct_mismatch* with more information, for example about the various preceding versions of a class.

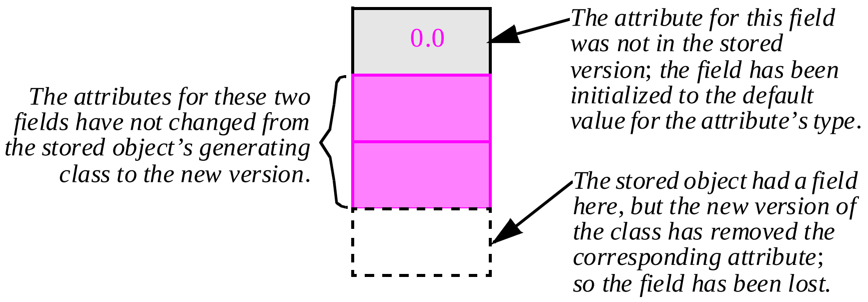
If you do expect object mismatches for a certain class, you will not want the default exception behavior for that class: instead you will redefine *correct_mismatch* so as to update the retrieved object. This is our last task: correction.

Correction

How do we correct a object that has been found, upon retrieval, to cause a mismatch? The answer requires a careful analysis, and a more sophisticated approach than has usually been implemented by existing systems or proposed in the literature.

The precise situation is this: the retrieval mechanism (through feature *retrieved* of class *STORABLE*, a database operation, or any other available primitive) has created a new object in the retrieving system, deduced from a stored object with the same generating class; but it has also detected a mismatch. The new object is in a temporary state and may be inconsistent; it may for example have lost a field which was present in the stored object, or gained a field not present in the original. Think of it as a foreigner without a visa.

Object mismatch


0.0
The attribute for this field was not in the stored version; the field has been initialized to the default value for the attribute's type.
The attributes for these two fields have not changed from the stored object's generating class to the new version.
The stored object had a field here, but the new version of the class has removed the corresponding attribute; so the field has been lost.

See "The role of creation procedures", page 372.

Such an object state is similar to the intermediate state of an object being created — outside of any persistence consideration — by a creation instruction !! *x*•*make* (...), just after the object's memory cell has been allocated and initialized to default values, but just before *make* has been called. At that stage the object has all the required components but is not yet ready for acceptance by the community since it may have inconsistent values in some of its fields; it is, as we saw, the official purpose of a creation procedure *make* to override default initializations as may be needed to ensure the invariant.

Let us assume for simplicity that the detection technique is structural and based on attributes (that is to say, policy C3 as defined earlier), although the discussion will transpose to the other solutions, nominal or structural. The mismatch is a consequence of a change in the attribute properties of the class. We may reduce it to a combination of any number of *attribute additions* and *attribute removals*. (If a class change is the replacement of the type of an attribute, we can consider it as a removal followed by an addition.) The figure above shows one addition and one removal.

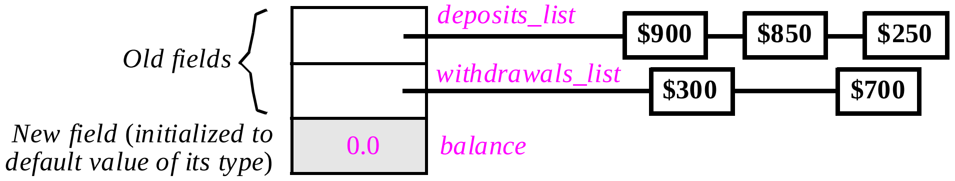
Attribute removal does not raise any apparent difficulty: if the new class does not include a certain attribute present in the old class, the corresponding object fields are not needed any more and we may simply discard them. In fact procedure *correct_mismatch* does not need to do anything for such fields, since the retrieval mechanism, when creating a tentative instance of the new class, will have discarded them; the figure shows this for the bottom field — rather, non-field — of the illustrated object.

We might of course be a bit more concerned about the discarded fields; what if they were really needed, so that the object will not make sense without them? This is where having a more elaborate *detection* policy, such as structural policy C4 which takes the invariant into account, would be preferable.

The more delicate case is when the new class has **added** an attribute, which yields a new field in the retrieved objects, as illustrated by the top field of the object in the preceding figure. What do we do with such a field? We must initialize it somehow. In the systems I have seen offering some support for schema evolution and object conversion, the solution is to use a conventional default as initialization value (the usual choices: zero for numbers, empty for strings). But, as we know from earlier discussions of similar problems — arising for example in the context of inheritance — this may be *very* wrong!

See "Uniform Access", page 55, and "Definition and example", page 364.

Our standard example was a class *ACCOUNT* with attributes *deposits_list* and *withdrawals_list*; assume that a new version adds an attribute *balance* and a system using this new version attempts to retrieve an instance created from the previous version.



Retrieving an account object

(*What is wrong with this picture*?)

The purpose of adding the *balance* attribute is clear: instead of having to recompute an account's balance on demand we keep it in the object and update it whenever needed. The new class invariant reflects this through a clause of the form

balance = deposits_list.total – withdrawals_list.total

But if we apply the default initialization to a retrieved object's *balance* field, we will get a badly inconsistent result, whose balance field does not agree with the record of deposits and withdrawals. On the above figure, *balance* is zero as a result of the default initialization; to agree with the deposits and withdrawals shown, it should be 1000 dollars.

Hence the importance of having the *correct_mismatch* mechanism. In such a case the class will simply redefine the procedure as

```
correct_mismatch is
                -- Handle object retrieval mismatch by correctly setting up balance
    do
        balance := deposits_list.total – withdrawals_list.total
    end
```

If the author of the new class has not planned for this case, the default version of *correct_mismatch* will raise an exception, causing the application to terminate abnormally unless a **retry** (providing another recovery possibility) handles it. This is the right outcome, since continuing execution could destroy the integrity of the execution's object structure — and, worse yet, of the persistent object structure, for example a database. In the earlier metaphor, we will reject the object unless we can assign it a proper immigration status.

31.4 FROM PERSISTENCE TO DATABASES

Using *STORABLE* ceases to be sufficient for true database applications. Its limitations have been noted in the earlier discussion: there is only one entry object; there is no support for content-based queries; each call to *retrieved* re-creates the entire structure, with no sharing of objects between successive calls. In addition, there is no support in *STORABLE* for letting different client applications access the same persistent data simultaneously.

Although various extensions of the mechanism can alleviate or remove some of these problems, a full-fledged solution requires taking advantage of database technology.

O-O or not, a set of mechanisms for storing and retrieving data items ("objects" in a general sense) deserves being called a database management system if it supports the following features:

- Persistence: objects can outlive the termination of individual program sessions using them, as well as computer failures.
- Programmable structure: the system treats objects as structured data connected by clearly defined relations. Users of the system can group a set of objects into a collection, called a database, and define the structure of a particular database.
- Arbitrary size: there is no built-in limit (such as could result from a computer's main memory size or addressing capability) to the number of objects in a database.
- Access control: users can "own" objects and define access rights to them.
- Property-based querying: mechanisms enable users and programs to find database objects by specifying their abstract properties rather than their location.
- Integrity constraints: users can define some semantic constraints on objects and have the database system enforce these constraints.
- Administration: tools are available to monitor, audit, archive and reorganize the database, add users, remove users, print out reports.
- Sharing: several users or programs can access the database simultaneously.
- Locking: users or programs can obtain exclusive access (read only, read and write) to one or more objects.
- Transactions: it is possible to define a sequence of database operations, called a transaction, with the guarantee that either the whole transaction will be executed normally or, if it fails, it will not have visibly affected the state of the database.

  The standard transaction example is a money transfer from a bank account to another, requiring two operations — debiting the first account and crediting the second — which must either succeed together or fail together. If they fail, any partial modification, such as debiting the first account, must be canceled; this is called *rolling back* the transaction.

The features listed are not exhaustive; they reflect what most current commercial systems offer, and what users have come to expect.

31.5 OBJECT-RELATIONAL INTEROPERABILITY

By far the most common form of database systems today is the **relational** kind, based on ideas developed by E. F. Codd in a 1970 article.

Definitions

A relational database is a set of *relations*, each containing a set of *tuples* (or *records*). A relation is also known as a *table* and a tuple as a *row* because it is convenient to present a relation in tabular form, as in

The BOOKS relation

title	*date*	*pages*	*author*
"The Red and the Black"	1830	341	"STENDHAL"
"The Charterhouse of Parma"	1839	307	"STENDHAL"
"Madame Bovary"	1856	425	"FLAUBERT"
"Eugénie Grandet"	1833	346	"BALZAC"

Each tuple is made of a number of *fields*. All the tuples in a relation have the same number and types of fields; in the example the first and last fields are strings, the other two are integers. Each field is identified by a name: *title*, *date* and so on in the above *BOOKS* example. The field names, or equivalently the columns, are known as *attributes*.

Some authors, notably Date, use "attribute name" for attribute and "attribute" for field.

Relational databases are usually *normalized*, meaning among other things that every field is a simple value (such as an integer, a real, a string, a date); it cannot be a reference to another tuple.

Operations

The relational model of databases comes with a *relational algebra* which defines a number of operations on relations. Three typical operations are selection, projection and join.

Selection yields a relation containing a subset of the rows of a given relation, based on some condition on the fields. Applying the selection condition "*pages* less than 400" to *BOOKS* yields a relation made of *BOOKS*'s first, second and last tuples.

The projection of a relation along one or more attributes is obtained by ignoring all the other fields, and removing any duplicate rows in the result. If we project the above relation along its last attribute we obtain a one-field relation with three tuples, "STENDHAL", "FLAUBERT" and "BALZAC"; if we project it along its first three attributes the result is a three-field relation, deduced from the above by removing the last column.

The join of two relations is a composite relation obtained by selecting type-compatible attributes in each of them and combining rows that match for these attributes. Assume that we also have a relation *AUTHORS*:

The AUTHORS relation

name	*real_name*	*birth*	*death*
"BALZAC"	"Honoré de Balzac"	1799	1850
"FLAUBERT"	"Gustave Flaubert"	1821	1880
"PROUST"	"Marcel Proust"	1871	1922
"STENDHAL"	"Henri Beyle"	1783	1842

Then the join of *BOOKS* and *AUTHORS* on the matching attributes *author* and *name* is the following relation:

Join of BOOKS and AUTHORS relations on author and name fields

title	*date*	*pages*	*author/name*	*real_name*	*birth*	*death*
"The Red and the Black"	1830	341	"STENDHAL"	"Henri Beyle"	1783	1842
"The Charterhouse of Parma"	1839	307	"STENDHAL"	"Henri Beyle"	1783	1842
"Madame Bovary"	1856	425	"FLAUBERT"	"Gustave Flaubert"	1821	1880
"Eugénie Grandet"	1833	346	"BALZAC"	"Honoré de Balzac"	1799	1850

Queries

The relational model permits queries — one of the principal database requirements of our earlier list — through a standardized language called SQL, with two forms: one to be used directly by humans, the other ("embedded SQL") to be used by programs. Using the first form, a typical SQL query is

```
select title, date, pages from BOOKS
```

yielding the titles, dates and page numbers of all recorded books. As you will have noted, such a query is, in the relational algebra, a projection. Another example is

```
select title, date, pages, author where pages < 400
```

corresponding in the relational algebra to a selection. The query

```
select
    title, date, pages, author, real_name, birth, date
from AUTHORS, BOOKS where
    author = name
```

is internally a join, yielding the same result as the join example given earlier.

Using relational databases with object-oriented software

The concepts of relational databases, as just sketched, bear a marked resemblance to the basic model of O-O computation. We can associate a relation with a class, and a tuple of that relation with an object — an instance of that class. We need a class library to provide us with the operations of relational algebra (corresponding to embedded SQL).

A number of object-oriented environments provide such a library for C++, Smalltalk or (with the *Store* library) the notation of this book. This approach, which we may call object-relational interoperability, has been used successfully by many developments. It is appropriate in either of the following circumstances:

- You are writing an object-oriented system which must use and possibly update existing corporate data, stored in relational databases. In such a case there is no other choice than using an object-relational interface.
- Your O-O software needs to store object structures simple enough to fit nicely in the relational view of things. (Reasons why it might *not* fit are explained next.)

If your persistence requirements fall outside of these cases, you will experience what the literature calls an *impedance mismatch* between the data model of your software development (object-oriented) and the data model of your database system (relational). You may then find it useful to take a look at the newest development in the database field: object-oriented database systems.

31.6 OBJECT-ORIENTED DATABASE FUNDAMENTALS

The rise of object-oriented databases has been fueled by three incentives:

D1 • The desire to provide object-oriented software developers with a persistence mechanism compatible with their development method — to remove the impedance mismatches.

D2 • The need to overcome conceptual limitations of relational databases.

D3 • The attempt to offer more advanced database facilities, not present in earlier systems (relational or not), but made possible and necessary by the general technological advance of the computer field.

The first incentive is the most obvious for someone whose background is O-O software development when he comes to the persistence question. But it is not necessarily the most important. The other two are pure database concerns, independent of the development method.

To study the concept of O-O database let us start by examining the limitations of relational systems (D2) and how they can fail to meet the expectations of an O-O developer (D1), then move on to innovative contributions of the O-O database movement.

Where relational databases stop

It would be absurd to deny the contribution of relational database systems. (In fact, whereas the first publications on O-O databases in the eighties tended to be critical of relational technology, the more recent trend is to describe the two approaches as complementary.) Relational systems have been one of the principal components in the growth of information technology since the seventies, and will be around for a long time. They are well adapted to situations involving data, possibly large amounts thereof, where

R1 • The structure of the data is regular: all objects of a given type have the same number and types of components.

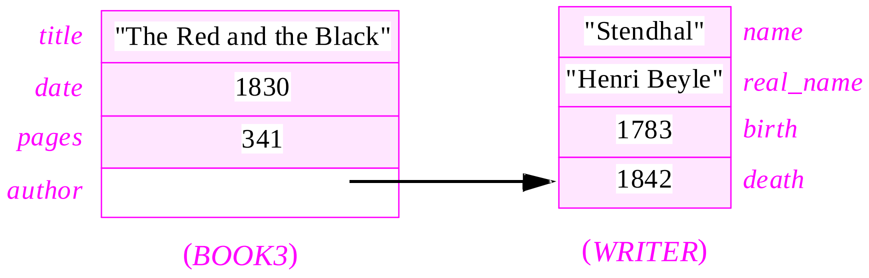
R2 • The structure is simple: the component types all belong to a small set of predefined possibilities.

R3 • These types are drawn from a small group of predefined possibilities (integers, strings, dates…), each with fixed space requirements.

A typical example is a census or taxpayer database with many objects representing persons, each made of a fixed set of components for the name (string), date of birth (date), address (string), salary (integer) and a few more properties.

Property R3 rules out many multimedia, CAD-CAM and image processing applications, where some data elements, such as image bitmaps, are of highly variable sizes, and sometimes very large. It also precludes, as a result of the "normal form" requirements enforced by existing commercial tools, the possibility for an object to refer to another object. This is of course a dramatic limitation when compared to what we have come to taking for granted in the discussions of this book: whenever we had

An object with a reference to another object



the object-oriented model made it easy to access indirect properties of an object, such as *redblack*.*author*.*birth_year* (yielding *1783* if *redblack* is attached to the object on the left of the figure). A relational description will not be able to represent the reference field *author*, whose value is the denotation of another object.

There is a workaround in the relational model, but it is heavy and impractical. To represent the above situation, you will have two relations, *BOOKS* and *AUTHORS*, as introduced a few pages back. Then, to connect the two relations, you may perform a **join**, which was also shown in the first part of this discussion, using matching fields *author* for the first relation and *name* from the second.

To answer questions such as “What is the birth year of the author of *The Red and the Black*?” the relational implementation will have to compute joins, projections etc.; here we can use the join seen earlier and then project along the *date* attribute.

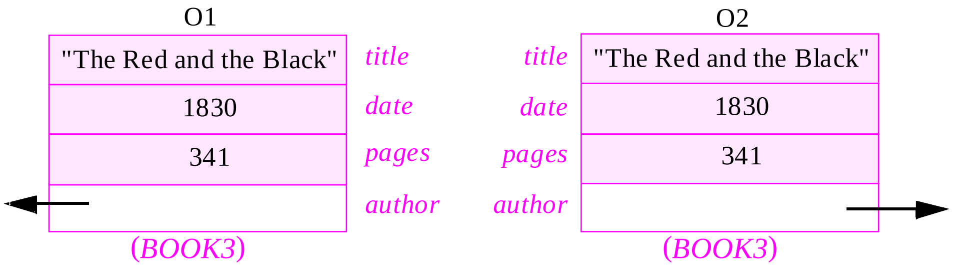
The join example was on page 1049.

This technique works and is widely used, but it is only applicable for simple schemes. The number of join operations would quickly become prohibitive in a system that must regularly handle queries with many indirections, as “How many rooms are there in the previous house of the manager of the department from which the lady who graduated at the top of my wife’s youngest maternal uncle’s undergraduate class was reassigned when the parent company went through its second round of venture funding?” — no particular problem in an O-O system’s run-time network of objects.

Object identity

The simplicity of the relational model follows in part from the identification of objects with their values. A relation (table) is a subset of $A \times B \times \dots$ for some sets A, B, …, where $\times$ represents cartesian product; in other words each one of the elements of the relation — each object — is a tuple *<a1, b1, …>* where *a1* is an element of *A* and so on. But such an object has no existence other than its value; in particular, inserting an object into a relation has no effect if the relation already has an identical tuple. For example inserting *<"The Red and the Black", 1830, 341, "STENDHAL">* into the above *BOOKS* relation does not change the relation. This is very different from the dynamic model of O-O computation, where we can have two identical objects:

Separate but equal



(*Both bottom references are attached to the same object.*)

As you will remember, *equal* (*obj1*, *obj2*) will have value true if *obj1* and *obj2* are references attached to these objects, but *obj1 = obj2* will yield false.

Being identical is not the same as being the same (ask any identical twins). This ability to distinguish between the two notions is part of the modeling power of object technology. It relies on the notion of **object identity**: any object has an existence independent of its contents.

Visitors to the Imperial Palace in Kyoto are told both that the buildings are very ancient and that each is rebuilt every hundred years or so. With the notion of object identity there is no contradiction: the object is the same even if its contents have changed.

You are the same individual as ten years ago even if none of the molecules that made up your body then remains in it now.

We can express object identity in the relational model, of course: just add to every object a special key field, guaranteed to be unique among objects of a given type. But we have to take care of it explicitly. With the O-O model, object identity is there by default.

In non-persistent O-O software construction, support for object identity is almost accidental: in the simplest implementation, each object resides at a certain address, and a reference to the object uses that address, which serves as immutable object identity. (This is not true any more in implementations, such as ISE's, which may move objects around for effective garbage collection; object identity is then a more abstract concept.) With persistence, object identify becomes a distinctive factor of the object-oriented model.

Maintaining object identity in a shared databases raises new problems: every client that needs to create objects must obtain a unique identity for them; this means that the module in charge of assigning identities must be a shared resource, creating a potential bottleneck in a highly concurrent setup.

The threshold model

After [Zdonik 1990].

From the preceding observations follows what has been called the threshold model of object-oriented databases: the minimum set of properties that a database system must satisfy if it deserves at all to be called O-O. (More advanced features, also desirable, will be discussed next.) There are four requirements for meeting the threshold model: **database, encapsulation, object identity** and **references**. The system must:

T1 • Provide database functionality, as defined earlier in this chapter.

T2 • Support encapsulation, that is to say allow hiding the internal properties of objects and make them accessible through an official interface.

T3 • Associate with each object an identification that is unique in the database.

T4 • Allow an object to contain references to other objects.

Notable in this list is the absence of some object-oriented mechanisms that we know are indispensable to the method, in particular inheritance. But this is not as strange as might appear at first. All depends on what you expect from a database system. A system at the threshold level might be a good **O-O database engine**, providing a set of mechanisms for storing, retrieving and traversing object structures, but leaving any higher knowledge about the semantics of these objects, such as the inheritance relations, to the design and programming language and the development environment.

The experience of early O-O database systems confirms that the database engine approach is reasonable. Some of the first systems went to the other extreme and had a complete "data model" with an associated O-O language supporting inheritance, genericity, polymorphism and so on. The vendors found that these languages were competing with O-O design and programming languages, and tended to *lose* such competitions (since a database language, will likely be less general and practical than one designed from the start as a universal programming language); they scurried in most cases to replace these proprietary offerings with interfaces to the main O-O languages.

Additional facilities

Beyond the threshold model a number of facilities are desirable. Most commercial systems offer at least some of them.

The first category includes direct support for more advanced properties of the O-O method: inheritance (single or multiple), typing, dynamic binding. This does not require more elaboration for the readers of this book. Other facilities, reviewed next, include: object versioning, schema evolution, long transactions, locking, object-oriented queries.

Object versioning

Object versioning is the ability to retain earlier states of an object after procedure calls have changed the state. This is particularly important as a result of concurrent accesses. Assume that an object O1 contains a reference to an object O2. A client changes some fields of O1, other than the reference. Another client changes O2. Then if the first client attempts to follow the reference, it may find a version of O2 that is inconsistent with O1.

Some O-O database systems address this problem by treating every object modification as the creation of a new object, thereby maintaining access to older versions.

Class versioning and schema evolution

Objects are not the only elements to require versioning: over time, their generating classes may change too. This is the problem of schema evolution, discussed at the beginning of this chapter. Only a few O-O database systems provide full support for schema evolution.

Long transactions

The concept of transaction has always been important in database systems, but classical transaction mechanisms have been directed towards *short* transactions: those which begin and end with a single operation performed by a single user during a single session of a computer system. The archetypal example, cited at the beginning of this chapter, is transferring a certain amount of money from one bank account to another; it is a transaction, since it requires an all-or-nothing outcome: either both operations (debiting one account and crediting the other) succeed, or both fail. The time it will take is on the order of seconds (less if we ignore user interaction).

Applications in the general idea of *design* of complex systems, such as CAD-CAM (computer-aided design and manufacturing of engineering products) and computer-aided software engineering, raise the need of *long* transactions, whose duration may be on the order of days or even months. During the design of a car, for example, one of the engineering teams may have to check out the carburetor part to perform some changes, and check it back in a week or two later. Such an operation has all the properties of a transaction, but the techniques developed for short transactions are not directly applicable.

The field of software development itself has obvious demand for long transactions, arising each time several people or teams work on a common set of modules. Interestingly, database technology has not been widely applied (in spite of many suggestions in the literature) to software development. The software field has instead developed for its own purposes a set of *configuration management* tools which address the specific issues of software component management, but also duplicate some standard database functions, most of the time without the benefit of database technology. This situation, surprising at first look, has a most likely explanation: the absence of support for long transactions in traditional database management systems.

Although long transactions may not conceptually require object technology, recent efforts to support them have come from O-O database systems, some of which offer a way to check any object in and out of a database.

Locking

Any database management system must provide some form of locking, to ensure safe concurrent access and updating. Early O-O database systems supported *page-level* locking, where the operating system determines the scope of a lock; this is inconvenient for large objects (which may extend over several pages) and small objects (which may fit several to a page, so that locking one will also lock the others). Newer systems provide *object-level* locking, letting a client application lock any object individually.

Recent efforts have tried hard to *minimize* the amount of locking that occurs in actual executions, since locking may cause contention and slow down the operation of the database. **Optimistic locking** is the general name for a class of policies which try to avoid placing a lock on an object a priori, but instead execute the possibly contentious operations on a copy, then wait as long as possible to update the master copy, locking it and reconciling conflicting updates at that time if necessary. We will see below an advanced form of optimistic locking in the Matisse case.

Queries

Database systems, it was recalled earlier, support queries. Here object-oriented systems can offer more flexibility than relational ones in the presence of schema evolution. Changing the schema of a relational database often means that you must change the query texts too and recompile them if appropriate. In an O-O database, the queries are relative to objects; you query the instances of a certain class with respect to some of their features. Here *instance* has, at least on option, its general sense covering both direct instances of a class and instances of its proper descendants; so if you add a descendant to a class the original queries on that class will be able to retrieve instances of the new descendant.

31.7 O-O DATABASE SYSTEMS: EXAMPLES

Since the mid-eighties a number of object-oriented database products have appeared. Some of the best-known product names are Gemstone, Itasca, Matisse, Objectivity, ObjectStore, Ontos, O_2, Poet, Versant. More recently a few companies such as UniSQL have introduced object-relational systems in an effort to reconcile the best of both approaches; the major relational database vendors are also proposing or announcing combined solutions, such as Informix's Illustra (based in part on UC Berkeley's POSTGRES project) and Oracle's announced Oracle 8 system.

To facilitate interoperability, a number of O-O database vendors have joined forces in the *Object Database Management Group*, which has proposed the ODMG standard to unify the general interface of O-O databases and their query language.

Let us take a look at two particularly interesting systems, Matisse and Versant.

Matisse

MATISSE, from ADB Inc., is an object-oriented database system with support for C, C++, Smalltalk and the notation of this book.

The official spelling is all upper case.

Matisse is a bold design with many non-conventional ideas. It is particularly geared towards large databases with a rich semantic structure and can manipulate very large objects such as images, films and sounds. Although it supports basic O-O concepts such as multiple inheritance, Matisse refrains from imposing too many constraints on the data model and instead serves as a powerful O-O database engine in the sense defined earlier in this chapter. Some of the strong points are:

- An original representation technique that makes it possible to split an object — especially a large object — over several disks, so as to optimize access time.
- Optimized object placement on disks.
- An automatic duplication mechanism providing a software solution to hardware fault tolerance: objects (rather than the disks themselves) can be mirrored across several disks, with automatic recovery in case of a disk failure.
- A built-in object versioning mechanism (see below).
- Support for transactions.
- Support for a client-server architecture in which a central server manages data for a possibly large number of clients, which keep a "cache" of recently accessed objects.

Matisse uses an original approach to the problem of minimizing locks. The mutual exclusion rule enforced by many systems is that several clients may read an object at once, but as soon as one client starts writing no other client may read or write. The reason, discussed in the concurrency chapter, is to preserve object integrity, as expressed by class invariants. Permitting two clients to write simultaneously could make the object inconsistent; and if a client is in the middle of writing, the object may be in an unstable state (one that does not satisfy the invariant), so that another client reading it may get an inconsistent result.

Writer-writer locks are clearly inevitable. Some systems, however, make it possible to breach the reader-writer exclusion by permitting read operations to occur even in the presence of a write lock. Such operations are appropriately called *dirty reads*.

Matisse, whose designers were clearly obsessed with the goal of minimizing locks, has a radical solution to this issue, based on object management: *no write operations*. Instead of modifying an existing object, a write operation (one, that is, which appears as such to the client software) will create a new object. As a result, it is possible to read objects without any locking: you will access a certain version of the database, unaffected by write operations that may occur after you start the read. You are also able to access a number of objects with the guarantee that they will all belong to the same version of the database, whereas with a more traditional approach you would have to use global locks or transactions, and incur the resulting performance penalties, to achieve the same result.

A consequence of this policy is the ability to go back to earlier versions of an object or of the database. By default, older versions are kept, but the system provides a "version collector" to get rid of unwanted versions.

Matisse provides interesting mechanisms for managing relations. If a class such as *EMPLOYEE* has an attribute *supervisor*: *MANAGER*, Matisse will on request maintain the inverse links automatically, so that you can access not only the supervisor of an employee but also all the employees managed by a supervisor. In addition, the query facilities can retrieve objects through associated keywords.

Versant

Versant, from Versant Object Technology, is an object-oriented database system with support for C++, Smalltalk and the notation of this book. Its data model and interface language support many of the principal concepts of O-O development, such as classes, multiple inheritance, feature redefinition, feature renaming, polymorphism and genericity.

Versant is one of the database systems conforming to the ODMG standard. It is meant for client-server architectures and, like Matisse, allows caching of the most recently accessed information, at the page level on the server side and at the object level for clients.

The design of Versant has devoted particular attention to locking and transactions. Locks can be placed on individual objects. An application can request a read lock, an update lock or a write lock. Update locks serve to avoid deadlock: if you have a read lock and want to write, you should first request an update lock, which will be granted only if no other client has done so; this still lets other clients read, until you request a write lock, which you are guaranteed to get. Going directly from read lock to write lock could cause deadlock: two clients each waiting indefinitely for the other to release its lock.

The transaction mechanism provides for both short and long transactions; an application may check out an object for any period. Object versioning is supported, as well as optimistic locking.

The query mechanism makes it possible to query all instances of a class, including instances of its proper descendants. As noted earlier, this makes it possible to add a class without having to redefine the queries applying to its previously existing ancestors.

Another interesting Versant capability is the event notification mechanism, which you can use to make sure that certain events, such as object update and deletion, will cause applications to receive a notification, enabling them to execute any associated actions that they may have defined for that purpose.

Versant provides a rich set of data types, including a set of predefined collection classes. It permits schema evolution, with the convention that new fields are initialized to default values. A set of indexing and query mechanism is available.

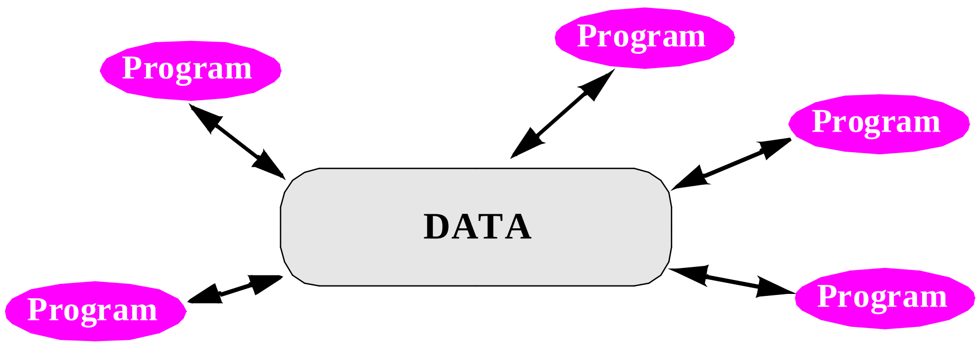
31.8 DISCUSSION: BEYOND O-O DATABASES

Let us conclude this review of persistence issues with a few musings on possible future evolutions. The observations that follow are tentative rather than final; they are meant to prompt further reflection rather than to provide concrete answers.

Is “O-O database” an oxymoron?

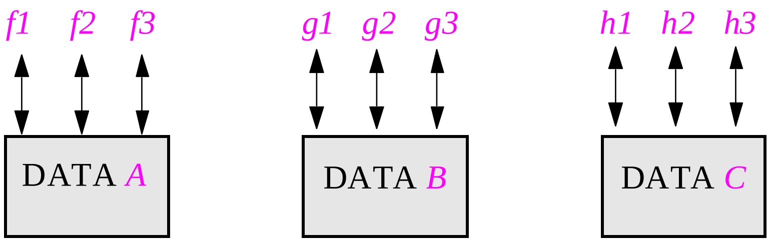
The notion of database proceeds from a view of the world in which the Data sit in the middle, and various programs are permitted to access and modify such Data:

The database view



In object technology, however, we have learned to understand data as being entirely defined by the applicable operations:

The O-O view



The two views seem incompatible! The notion of data existing independently of the programs that manipulate them (“data independence”, a tenet reaffirmed in the first few pages of every database textbook) is anathema to the object-oriented developer. Should we then consider that “object-oriented database” is an oxymoron?

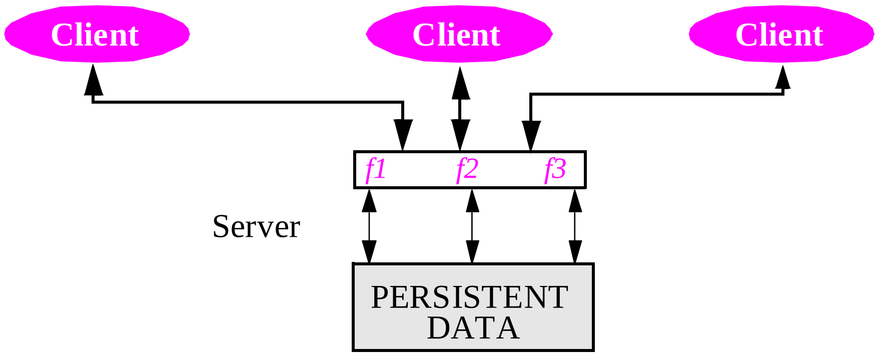
“FROM PERSISTENCE TO DATABASES”, 31.4, page 1047.

Perhaps not, but it may be worthwhile to explore how, in a dogmatic O-O context, we could obtain the effect of databases without really having databases. If we define (simplifying to the barest essentials the definition of databases given earlier in this chapter)

DATABASE = PERSISTENCE + SHARING

the dogmatic view would consider the second component, data sharing, as incompatible with O-O ideas, and focus on persistence only. Then we would address the sharing needs through a different technique: concurrency! The picture becomes

Separating persistence from sharing



On concurrency and the **separate** *mechanism see chapter 30.*

Following O-O principles, the persistent data are implemented as a set of objects — instances of some abstract data types — and controlled by a certain server system. Client systems that need to manipulate the data will do so through the server; because the setup requires sharing and concurrent access, the clients will treat the server as **separate** in the sense defined by the discussion of concurrency. For example:

```
flights: separate FLIGHT_DATABASE; …

flight_details (f: separate FLIGHT_DATABASE;
                    rf: REQUESTED_FLIGHTS): FLIGHT is
    do
        Result := f.flight_details (rf)
    end

reserve (f: separate FLIGHT_DATABASE; r: RESERVATION) is
    do
        f.reserve (r); status := f.status
    end
```

Then the server side requires no sharing mechanism, only a general persistence mechanism. We may also need tools and techniques to handle such matters as object versioning, which are indeed *persistence* rather than *database* issues.

The persistence mechanism could then become extremely simple, shedding much of the baggage of databases. We might even consider that *all objects are persistent by default*; transient objects become the exception, handled by a mechanism that generalizes garbage collection. Such an approach, inconceivable when database systems were invented, becomes less absurd with the constant decrease of storage costs and the growing availability of 64-bit virtual address spaces where, it has been noted, "*one could create a new 4-gigabyte object, the size of a full address space on a conventional 32-bit processor, once a second for 136 years and not exhaust the available namespace. This is sufficient to store all the data associated with almost any application during its entire lifetime*."

Quotation from [Sombrero-Web].

All this is speculative, and provides no proof that we should renounce the traditional notion of database. There is no need to rush and sell your shares of O-O database companies yet. Consider this discussion as an intellectual exercise: an invitation to probe further into the widely accepted notion of O-O database, examining whether the current approach truly succeeds in removing the dreaded *impedance mismatches* between the software development method and the supporting data storage mechanisms.

Unstructured information

A final note on databases. With the explosion of the World-Wide Web and the appearance of content-based search tools (of which some well-known examples, at the time of writing, are AltaVista, Web Crawler and Yahoo) it has become clear that we can access data successfully even in the absence of a database.

Database systems require that before you store any data for future retrieval you first convert it into a strictly defined format, the database schema. Recent studies, however, show that 80% of the electronic data in companies is unstructured (that is to say, resides outside of databases, typically in text files) even though database systems have been around for many years. This is where content-based tools intervene: from user-defined criteria involving characteristic words and phrases, they can retrieve data from unstructured or minimally structured documents. Almost anyone who has tried these tools has been bedazzled by the speed at which they can retrieve information: a second or two suffices to find a needle in a bytestack of thousands of gigabytes. This leads to the inevitable question: do we really need structured databases?

The answer is still yes. Unstructured and structured data will coexist. But databases are no longer the only game in town; more and more, sophisticated query tools will be able to retrieve information even if it is not in the exact format that a database would require. To write such tools, of course, object technology is our best bet.

31.9 KEY CONCEPTS STUDIED IN THIS CHAPTER

- An object-oriented environment should allow objects to be persistent — to remain in existence after the session creating them has terminated.
- A persistence mechanism should offer *schema evolution* to convert retrieved objects on the fly if their generating class has changed ("object mismatch"). This involves three tasks: detection, notification, correction. By default, a mismatch should cause an exception.
- Beyond persistence, many applications need database support, offering concurrent access to clients.

- Other properties of databases include querying, locking and transactions.
- It is possible to use O-O development in conjunction with relational databases, through a simple correspondence: classes to relations, objects to tuples.
- To gain full use of object technology and avoid impedance mismatches between the development and the data model, you may use object-oriented databases.
- Two interesting O-O database systems were studied: Matisse, providing original solutions for object versioning and redundancy, and Versant, providing advanced locking and transaction mechanisms.
- In a more tentative part of the discussion, some questions were raised as to the true compatibility of database principles with the O-O view, and the need for accessing unstructured as well as structured data.

31.10 BIBLIOGRAPHICAL NOTES

The original paper on the relational model is [Codd 1970]; there are many books on the topic. Probably the best-known database textbook, with particular emphasis on the relational model, is [Date 1995], the sixth edition of a book originally published in the mid-seventies. Another useful general-purpose text is [Elmasri 1989].

[Waldén 1995] contains a detailed practical discussion of how to make object-relational interoperability work. [Khoshafian 1986] brought the question of object identity to the forefront of O-O database discussions.

A good starting point for understanding the goals of object-oriented database systems and reading some of the original papers is [Zdonik 1990], a collection of contributions by some of the pioneers in the field, whose introductory chapter is the source of the "threshold model" concept used in the present chapter. The widely circulated "O-O Database System Manifesto" [Atkinson 1989], the result of the collaboration of a number of experts, has been influential in defining the goals of the O-O database movement. There are now a number of textbooks on the topic; some of the best known, in order of publication, are: [Kim 1990], [Bertino 1993], [Khoshafian 1993], [Kemper 1994], [Loomis 1995]. For further, regularly updated references, Michael Ley's on-line bibliography of database systems [Ley-Web] is precious. Klaus Dittrich's group at the University of Zürich maintains a "mini-FAQ" about O-O databases at *http://www.ifi.unizh.ch/groups/dbtg/ObjectDB/ODBminiFAQ.html*. [Cattell 1993] describes the ODMG standard. For an appraisal, somewhat jaded, of the achievements and failures of O-O databases by one of the pioneers of the field, see [Stein 1995].

This chapter has benefited from important comments by Richard Bielak, particularly on schema evolution, Persistence Closure, queries in O-O databases, Versant and Sombrero. Its presentation of Versant is based on [Versant 1994], that of Matisse on [ADB 1995] (see also *http://www.adb.com/techovw/features.html*). I am indebted to Shel Finkelstein for helping me with the features of Matisse. O_2 is described in [Bancilhon 1992]. The Sombrero project [Sombrero-Web] has explored the implications of large address spaces on traditional approaches to persistence and databases.

A preview of some of this chapter's material on schema evolution appeared as [M 1996c]. The questioning of how well O-O and database concepts really match comes from two unpublished keynote lectures, presented in 1995 at TOOLS USA and the European Software Engineering Conference [M 1995d].

EXERCISES

E31.1 Dynamic schema evolution

Study how to extend the schema evolution techniques developed in this chapter to account for the case in which classes of a software system may change during the system's execution.

E31.2 Object-oriented queries

Discuss the form that queries may take in an object-oriented database management system.