Heinz ZÜLLIGHOVEN

Object-Oriented Construction Handbook

DEVELOPING APPLICATION-ORIENTED SOFTWARE WITH THE TOOLS AND MATERIALS APPROACH

Object-Oriented Construction Handbook

This page intentionally left *blank*

Object-Oriented Construction Handbook

Developing Application-Oriented Software with the Tools & Materials Approach

Heinz Züllighoven

IT-Workplace Solutions, Inc., and University of Hamburg, Germany

With contributions by

as lead authors
Robert F. Beeger
Wolf-Gideon Bleek
Guido Gryczan
Carola Lilienthal
Martin Lippert
Stefan Roock
Wolf Siberski
Thomas Slotos
Dirk Weske
Ingrid Wetzel

as co-authors
Dirk Bäumer
Petra Becker-Pechau
Holger Breitling
Ute Bürkle
Rolf Knoll
Anita Krabbel
Daniel Megert
Dirk Riehle
Axel Schmolitzky
Wolfgang Strunk
Henning Wolf

ELSEVIER

MORGAN KAUFMANN PUBLISHERS

dpunkt.verlag

AMSTERDAM ● BOSTON ● HEIDELBERG ● LONDON ● NEW YORK ● OXFORD
PARIS ● SAN DIEGO ● SAN FRANCISCO ● SINGAPORE ● SYDNEY ● TOKYO

HEIDELBERG

MORGAN KAUFMANN PUBLISHERS IS AN IMPRINT OF ELSEVIER

Copublished by Morgan Kaufmann Publishers and dpunkt.verlag

Morgan Kaufmann Publishers
Senior Editor: Tim Cox
Publishing Services Manager: Simon Crump
Project Manager: Brandy Palacios
Editorial Assistant: Richard Camp
Cover Design: Frances Baca Design
Cover Image: Montage of tools in boxes, courtesy of Digital Vision and Getty Images
Text Design: Julio Esperas
Composition: Newgen Imaging Systems
Technical Illustration: RDC Tech
Translators: Angelika Shafir, Bob Bach
Copyeditor: Harbour Fraser Hodder
Proofreader: Graphic World Inc.
Indexer: Graphic World Inc.
Interior printer: The Maple-Vail Book Manufacturing Group
Cover printer: Phoenix Color Corp.

dpunkt.verlag
Senior Editor: Christa Preisendanz

Morgan Kaufmann Publishers is an imprint of Elsevier.
500 Sansome Street, Suite 400, San Francisco, CA 94111

Available in Germany, Austria, and Switzerland from
dpunkt.verlag
Ringstraße 19B
69115 Heidelberg, Germany
http://www.dpunkt.de

Library of Congress Cataloging-in-Publication Data
Application submitted

MK ISBN: 1-55860-687-4
dpunkt ISBN: 3-89864-254-2

This book is printed on acid-free paper.

For information on all Morgan Kaufmann publications,
visit our website at *www.mkp.com.*

Printed in the United States of America
04 05 06 07 08 5 4 3 2 1

CONTENTS

Preface *xvii*

1 Introduction

1.1 APPLICATION ORIENTATION — THE SUBJECT OF THIS BOOK 1
1.1.1 Motivation 1
1.1.2 Structure of This Book 3
1.2 THE TOOLS & MATERIALS APPROACH (T&M) 4
1.2.1 The T&M Approach in a Nutshell 4
1.2.2 T&M As a Method 8
1.3 PROJECTS BEHIND THIS BOOK 9
1.3.1 The Scope of the T&M Projects 10
1.4 THE EQUIPMENT MANAGEMENT SYSTEM EXAMPLE 14
1.5 REFERENCES 15

2 The T&M Object Metamodel

2.1 THE OBJECT METAMODEL 17
2.1.1 Introduction 17
2.1.2 Definition: The Object Metamodel 18
2.1.3 Context: What's the Purpose of an Object Metamodel? 18
2.1.4 Context: A Classification of Programming Languages 19
2.1.5 The Object Metamodel and the Software Model 20
2.1.6 Definition: Objects 20
2.1.7 Discussion: Object Identity 24
2.1.8 T&M Design: Structuring an Interface 25
2.1.9 Definition: Classes 25
2.1.10 Discussion: Generic Operations 27
2.1.11 T&M Design: Generic Operations 27
2.1.12 Discussion: The Object Life Cycle 28
2.1.13 T&M Design: The Object Life Cycle 29

2.1.14 Definition: Inheritance 30
2.1.15 Discussion: Inheritance 30
2.1.16 T&M Design: Inheritance 31
2.1.17 Discussion: Role Relationships as an Alternative to Inheritance 32
2.1.18 Definition: Use Relationships 32
2.1.19 Discussion: Use Relationships 33
2.1.20 Definition: Polymorphism 33
2.1.21 Definition: Abstract Classes 34
2.1.22 Discussion: Specification and Implementation 35
2.1.23 Definition: Loose Coupling 35
2.1.24 Discussion: Loose Coupling 36

2.2 MODULARIZATION 36
2.2.1 Introduction 36
2.2.2 Context: Modules and Object Orientation 36
2.2.3 Definition: Principles of Object-Oriented Modularization 37

2.3 THE CONTRACT MODEL 38
2.3.1 Introduction 38
2.3.2 Definition: Contract Model 38
2.3.3 Discussion: The Contract Model 39
2.3.4 Context: The Contract Model and Abstract Data Types 39
2.3.5 T&M Design: The Contract Model 40

2.4 TYPES 40
2.4.1 Introduction 40
2.4.2 Definition: Types 41
2.4.3 Context: The Theoretical Concept of Types 41
2.4.4 Discussion: Types 42

2.5 CLASSES AND TYPES 43
2.5.1 Introduction 43
2.5.2 Classes in Your Design 43
2.5.3 Differences between Types and Classes 43
2.5.4 Discussion: Classes versus Types 44
2.5.5 Background: Programming Languages and Types 44
2.5.6 T&M Design: Classes and Types 45

2.6 VALUES AND OBJECTS 45
2.6.1 Introduction 45
2.6.2 Characteristics of Values and Objects 46
2.6.3 Using Values 47
2.6.4 Context: Values and Objects in Programming Languages 48
2.6.5 Definition: Domain Values 49

2.6.6 T&M Design: Domain Values 49
2.6.7 Implementing Domain Values 49

2.7 METAOBJECT PROTOCOLS 50
2.7.1 Introduction 50
2.7.2 Motivation for a Metaobject Protocol 51
2.7.3 Definition: Metaobject Protocol (MOP) 51
2.7.4 Representing Your Application Model 51
2.7.5 Representing Your Runtime System 53

2.8 REFERENCES 55

3 Guiding Metaphors and Design Metaphors

3.1 INTRODUCTION 57

3.2 DESIGNING APPLICATION SOFTWARE 57
3.2.1 Definition: The Usage Model 58
3.2.2 Background: Methodology Books versus Design Guidelines 58

3.3 GUIDING METAPHORS FOR APPLICATION SOFTWARE 59
3.3.1 Background: Guiding Metaphors in Software Development 60
3.3.2 The "Object Worlds" Guiding Metaphor 60
3.3.3 The Direct Manipulation Guiding Metaphor 61
3.3.4 Discussion: Guiding Metaphor of Direct Manipulation 61
3.3.5 The Guiding Metaphor of the Factory 62
3.3.6 Discussion: The Factory Guiding Metaphor 62

3.4 DESIGN METAPHORS 63
3.4.1 Definition: Metaphors 63

3.5 T&M GUIDING METAPHORS AND DESIGN METAPHORS 64
3.5.1 A T&M Guiding Metaphor: The Expert Workplace 64
3.5.2 Background: The Supportive View 65
3.5.3 Discussion: Metaphors and Patterns 66
3.5.4 Definition: A Tool 67
3.5.5 The Tool as a Design Metaphor 67
3.5.6 T&M Design: Software Tools 67
3.5.7 Definition: Material 68
3.5.8 Material as a Design Metaphor 68
3.5.9 T&M Design: Software Materials 68
3.5.10 Definition: The Work Environment 69

3.5.11 The Work Environment as a Design Metaphor 69
3.5.12 T&M Design: The Work Environment 70
3.5.13 Definition: Automatons 70
3.5.14 The Automaton as a Design Metaphor 70
3.5.15 T&M Design: Software Automatons 71
3.5.16 Definition: The Container 71
3.5.17 The Container as a Design Metaphor 72
3.5.18 T&M Design: Containers 72
3.5.19 Discussion: Design Metaphors 73

3.6 WORKPLACE TYPES 73
3.6.1 Definition: Workplace Types 74
3.6.2 T&M Design: The Expert Workplace Type 74
3.6.3 T&M Design: The Functional Workplace Type 74
3.6.4 T&M Design: The Back-Office Workplace Type 75
3.6.5 T&M Design: The Electronic Commerce Frontend Workplace Type 76

3.7 REFERENCES 77

4 Patterns, Frameworks, and Components

4.1 INTRODUCTION 79

4.2 BACKGROUND: PATTERNS, FRAMEWORKS, AND COMPONENTS 80

4.3 PATTERNS 80
4.3.1 Definition: Patterns 80
4.3.2 The Characteristics of a Pattern 81
4.3.3 A Taxonomy of T&M Patterns 83
4.3.4 Conceptual Patterns 83
4.3.5 Design Patterns 84
4.3.6 Programming Patterns 85
4.3.7 T&M Design: Design Patterns 86
4.3.8 T&M Design: Models, Metaphors, and Patterns 86
4.3.9 Background: Pattern Form 88
4.3.10 T&M Design: Pattern Form 88
4.3.11 Pattern Collections 89

4.4 FRAMEWORKS 90
4.4.1 Background: Class Libraries 90
4.4.2 Definition: Frameworks 91
4.4.3 Application Frameworks 91
4.4.4 Black-Box and White-Box Frameworks 92
4.4.5 Connectors between Frameworks 94
4.4.6 JWAM Framework: Layered Framework Architecture 95

4.5 COMPONENTS 96
4.5.1 Background: Software Components 96
4.5.2 Definition: Components 97
4.5.3 Current Component Products 98
4.5.4 Components and Frameworks 98
4.6 REFERENCES 99

5 Application-Oriented Software Development

5.1 INTRODUCTION 101
5.2 APPLICATION-ORIENTED SOFTWARE 101
5.2.1 Application Software 102
5.2.2 Definition: Application Orientation 102
5.2.3 Background: Application Orientation 103
5.2.4 Usage Quality 104
5.2.5 T&M Design: Structural Similarity 104
5.3 THE DEVELOPMENT PROCESS 105
5.3.1 Definition: Software Development 105
5.3.2 The Application-Oriented Development Process 105
5.3.3 Discussion: The Development Process 106
5.3.4 The Author-Critic Cycle 107
5.3.5 Discussion: The Author-Critic Cycle 107
5.3.6 Evolutionary System Development 108
5.3.7 Documentation in Software Development 108
5.3.8 Discussion: Documentation 109
5.3.9 Application-Oriented Development Documents 109
5.3.10 Discussion: Application-Oriented Document Types 110
5.3.11 T&M Design: Application-Oriented Document Types 111
5.3.12 Discussion: T&M Document Types 113
5.3.13 Project Documents 113
5.3.14 Documentation Guidelines 114
5.4 REFERENCES 115

6 Software Development as a Modeling Process

6.1 INTRODUCTION 117
6.2 A SIMPLIFIED SOFTWARE DEVELOPMENT MODEL 117
6.2.1 Discussion: A Descriptive Software Development Model 118
6.3 THE APPLICATION DOMAIN 119
6.3.1 Definition: Application Domain 119
6.3.2 Discussion: Analyzing the Application Domain 119

6.4 THE DOMAIN MODEL 120
6.4.1 Modeling Your Application Domain 120
6.4.2 Discussion: Modeling Your Application Domain 122
6.5 THE APPLICATION SYSTEM MODEL 122
6.5.1 Context: The Application System Model 122
6.5.2 Definition: Application System Model 123
6.5.3 Discussion: Structural Similarity and Macrostructures 125
6.6 THE APPLICATION SYSTEM 125
6.6.1 Definition: The Application System 126
6.6.2 Discussion: The Application System 127
6.7 SOFTWARE DEVELOPMENT CONTEXTS 127
6.7.1 Discussion: Software Development Contexts 127
6.7.2 The Application Domain Context 127
6.7.3 Discussion: The Application Domain Context 128
6.7.4 Applied Technique 128
6.7.5 Handling and Presentation 129
6.7.6 Discussion: Handling and Presentation 130
6.8 CONTEXTS INFLUENCING THE SOFTWARE ARCHITECTURE 130
6.8.1 Discussion: How Contexts Influence Your Software Architecture 130
6.9 REFERENCES 132

7 T&M Conceptual Patterns

7.1 CONCEPTUAL PATTERNS 135
7.1.1 Conceptual Patterns in the Development Process 136
7.1.2 The T&M Conceptual Patterns 136
7.2 A GUIDED TOUR OF THE T&M CONCEPTUAL PATTERNS 137
7.3 THE INTERRELATION OF TOOLS AND MATERIALS PATTERN 139
7.4 THE MATERIAL DESIGN PATTERN 142
7.5 THE TOOL DESIGN PATTERN 146
7.6 THE WORK ENVIRONMENT PATTERN 152
7.7 THE CONTAINER PATTERN 159
7.8 THE FORM PATTERN 164
7.9 THE AUTOMATON PATTERN 167
7.10 THE DOMAIN SERVICE PROVIDER PATTERN 171

7.11 THE TECHNICAL AUTOMATON PATTERN 176

7.12 THE PROBE PATTERN 179

7.13 THE ADJUSTING TOOL PATTERN 181

8 T&M Design Patterns

8.1 INTRODUCTION 185

8.2 A GUIDED TOUR OF THE T&M DESIGN PATTERNS 187

8.3 THE ASPECT PATTERN 190

8.3.1 Construction Part: Using Inheritance or Interfaces to Implement Aspects 195

8.3.2 Construction Part: Using Object Adapters to Implement Aspects 198

8.3.3 Construction Part: Using Development Tools to Realize Aspects 200

8.3.4 Construction Part: Alternatives to Using Aspects 201

8.4 THE SEPARATING FUNCTION AND INTERACTION PATTERN 202

8.5 THE TOOLS COMPOSITION PATTERN 204

8.5.1 Construction Part: Using Components to Build Tools 208

8.5.2 Construction Part: Using Components to Build Combination Tools 209

8.5.3 Construction Part: Identifying Tool Boundaries 210

8.6 THE FEEDBACK BETWEEN TOOL PARTS PATTERN 212

8.6.1 Construction Part: Event Pattern 214

8.6.2 Construction Part: Event Objects 215

8.6.3 Construction Part: Chain of Responsibility 217

8.6.4 Construction Part: Tool Component with Reaction Mechanisms 219

8.7 THE SEPARATING FP AND IP PATTERN 221

8.7.1 Construction Part: Interactive Part (IP) 225

8.7.2 Construction Part: FP 226

8.8 THE SEPARATING HANDLING AND PRESENTATION PATTERN 227

8.9 THE FEEDBACK BETWEEN INTERACTION FORMS AND IP PATTERN 232

8.10 THE DOMAIN VALUES PATTERN 236

8.10.1 Construction Part: Domain Value Classes 238

8.10.2 Construction Part: Immutable Domain Value Objects 240

8.10.3 Construction Part: Mutable Domain Value Objects 242

8.10.4 Construction Part: Implementing Domain Values as Streams 243
8.10.5 Construction Part: Domain Value Types by Configuration 243

8.11 THE DOMAIN CONTAINER PATTERN 244
8.11.1 Construction Part: Using Technical Containers to Implement Domain Containers 246
8.11.2 Construction Part: Loading Materials 247
8.11.3 Construction Part: Tables of Contents for Containers 248
8.11.4 Construction Part: Implementing Tables of Contents as Materials 249
8.11.5 Construction Part: Implementing Tables of Contents as Domain Values 249
8.11.6 Construction Part: Coping with Changes to Containers 250

8.12 THE FORM SYSTEM PATTERN 252

8.13 THE AUTOMATONS IN TECHNICALLY EMBEDDED SYSTEMS PATTERN 258

8.14 THE DOMAIN SERVICES PATTERN 264

8.15 THE ENVIRONMENT PATTERN 268

8.16 USING THE T&M DESIGN PATTERNS FOR THE JWAM FRAMEWORK 271
8.16.1 Materials Construction 271
8.16.2 Tools Construction 272
8.16.3 Domain Values 274
8.16.4 Presentation and Interaction Forms 278
8.16.5 Forms 278
8.16.6 Domain Services 279
8.16.7 Work Environment 279

8.17 REFERENCES 279

9 T&M Model Architecture

9.1 THE T&M MODEL ARCHITECTURE 281

9.2 THE DOMAIN CORE OF A SOFTWARE ARCHITECTURE 283
9.2.1 The Use Context 285
9.2.2 The Product Domain 287
9.2.3 The Business Domain 289
9.2.4 How Different Domains Relate 293

9.3 CONCEPTS AND ELEMENTS OF A T&M MODEL ARCHITECTURE 295

9.3.1 Components of an Object-Oriented Software Architecture 296
9.3.2 Elementary Rules for Combining Elements of a Software Architecture 296
9.3.3 Protocol-Based Layer Architectures 297
9.3.4 Object-Oriented Layer Architectures 299
9.3.5 The Layer Concept of the T&M Model Architecture 303
9.3.6 The Three-Tier Architecture 305
9.3.7 The T&M Model Architecture 307

9.4 DESIGN PATTERNS FOR THE T&M MODEL ARCHITECTURE 314
9.4.1 The Role Pattern 315
9.4.2 The Product Trader Pattern 327

9.5 REFERENCES 339

10 Supporting Cooperative Work

10.1 BACKGROUND: COMPUTER-SUPPORTED COOPERATIVE WORK 341
10.1.1 CSCW 341
10.1.2 Technical and Domain Transparency 344

10.2 IMPLICIT COOPERATION 345

10.3 EXPLICIT COOPERATION BY EXCHANGING MATERIALS 346
10.3.1 Cooperation Medium: Mailboxes 346
10.3.2 Cooperation Medium: Mailing System 349

10.4 EXPLICIT COOPERATION MODEL: TRANSACTION PROCESSING SUPPORT 350
10.4.1 The Concept of Process Patterns 353
10.4.2 Cooperation Medium: Routing Slips 354

10.5 REFERENCES 356

11 Interactive Application Systems and Persistence

11.1 BACKGROUND: INTERACTIVE APPLICATION SYSTEMS 357

11.2 PERSISTENCE SERVICES 358
11.2.1 Software Registry 358
11.2.2 The Basic Concept of a Software Registry 360
11.2.3 Cooperation Model for a Registry 361

11.2.4 Usage Model for a Registry 364
11.2.5 JWAM: Architecture for a Software Registry 365
11.2.6 The Generic Persistence Service 369

11.3 DESIGN CRITERIA TO IMPLEMENT PERSISTENCE 373
11.3.1 Client-Server Architecture 374
11.3.2 Identifiers 375
11.3.3 Technical Data Modeling and Structural Mapping 376
11.3.4 Querying and Data Warehousing 378
11.3.5 Load-on-Demand 379
11.3.6 Transactions and Locking 381
11.3.7 Class Evolution 382
11.3.8 Legacy Databases 382

11.4 REAL-WORLD EXAMPLES 383
11.4.1 JWAM Projects 383
11.4.2 MedIS and SyLab 386

12 The Development Process

12.1 BACKGROUND: EVOLUTIONARY AND TRADITIONAL PROCESS MODELS 393
12.1.1 The Context of Our Process Model 394
12.1.2 The Process Model's Application Orientation 394
12.1.3 The Classic Waterfall Model 395
12.1.4 The Spiral Model 397
12.1.5 An Idealized Evolutionary Process Model 398

12.2 TOPICS FOR A DEVELOPMENT STRATEGY 400
12.2.1 Sequence of Development Activities 400
12.2.2 Objectifying the Development Process 401
12.2.3 Lehman's Software Classification 402
12.2.4 The Cooperative Development Process 405
12.2.5 Organizational and Domain Integration 406
12.2.6 Developing an IT Organization 409

12.3 QUALITY ASSURANCE IN THE DEVELOPMENT PROCESS 413
12.3.1 Direct User Integration 413
12.3.2 Prototyping 414
12.3.3 Reviews 414
12.3.4 Pair Programming 416
12.3.5 Refactoring 418

12.4 QUALITY ASSURANCE IN CONSTRUCTION 419
12.4.1 Characteristics of OO Testing 419
12.4.2 Testing OO Programs 421

12.5 PROJECT MANAGEMENT 425
12.5.1 Fundamental Activities of the Management Process 425
12.5.2 The Contexts of the Management Process 427
12.6 PROJECT PLANNING CONCEPTS AND TECHNIQUES 428
12.6.1 Project Calibration 428
12.6.2 Project Goals 430
12.6.3 Decision Principles 431
12.6.4 Project Establishment 432
12.7 STRUCTURING A PROJECT BY SYSTEM DECOMPOSITION 432
12.7.1 Core System and Special-Purpose Systems 432
12.7.2 Core System and Extension Levels 434
12.8 SCHEDULING AND TASK PLANNING 437
12.8.1 General Rules for Time Estimates 437
12.8.2 Planning the Project Stages 438
12.8.3 Using Base Lines for Detailed Planning 441
12.8.4 The UP and T&M Project Planning 443
12.9 DISCUSSING T&M, UNIFIED PROCESS, AND XP 451
12.9.1 Structure of the UP and T&M Development Processes 451
12.10 REFERENCES 456

13 T&M Document Types

13.1 SCENARIOS 459
13.1.1 Using Scenarios in the Development Process 462
13.1.2 Subtypes of Scenarios 464
13.1.3 Scenarios and UML 466
13.2 INTERVIEWS 467
13.2.1 The Interview Process 468
13.3 THE CONCEPT MODEL 472
13.3.1 Using a Concept Model in the Development Process 475
13.3.2 Concept Models and UML 475
13.4 GLOSSARIES 475
13.4.1 Using a Glossary in the Development Process 477
13.4.2 Glossaries, UML, and UP 479
13.5 SYSTEM VISIONS 479
13.5.1 Using System Visions in the Development Process 480
13.5.2 System Visions and XP 482
13.5.3 System Visions and UML 482

13.6 PROTOTYPES 483
13.6.1 Using Prototypes in the Development Process 485
13.6.2 Prototypes and UP 487
13.7 COOPERATION PICTURES 488
13.7.1 Cooperation Pictures in the Development Process 490
13.7.2 Cooperation Pictures and UML 494
13.8 PURPOSE TABLES 494
13.8.1 Using Purpose Tables in the Development Process 495
13.8.2 Purpose Tables and UML 496
13.9 TECHNICAL DOCUMENT TYPES IN UML 496
13.9.1 Class Diagrams 497
13.9.2 Object Diagrams 497
13.9.3 Use Case Diagrams 498
13.9.4 Interaction Diagrams 498
13.9.5 Statechart Diagrams 498
13.9.6 Activity Diagrams 498
13.9.7 Component Diagrams 498
13.9.8 Deployment Diagrams 498
13.9.9 Application-Oriented and Technical Documents 499
13.10 REFERENCES 499
Index *501*

PREFACE

We have written this book because developing good application software is so hard to do. We have been trying to develop good application software for more than a decade, and we wrote this book because we hope to give some help to those who realize that technology and tools are not enough to develop good software.

This is a book about application orientation, which means structuring large interactive software systems along the concepts, interactions, and relations of an application domain. But this also means organizing software projects so that domain experts—the potential users—can actively participate and shape the future system. And, last but not least, it means using metaphors to analyze, design, use, and talk about systems.

We have worked on the idea of application orientation for many years and have used it in a large number of commercial and scientific projects. Over the years we have collected a repertoire of metaphors, concepts, techniques, strategies, patterns, and best practices for designing and constructing software and for managing and conducting projects. This collection, together with a specific way of looking at software development, we call the *Tools & Materials approach (T&M)*.

0.1 THE READER

This book is intended for the following target groups:

- Software developers
- Project managers
- Computer science students
- Method developers

It is neither a programming textbook nor a book about GUI design.

0.1.1 Target Groups

Now, in more detail, who are the readers who will benefit from this book? Admittingly, we were quite ambitious, as we defined several target groups:

Practitioners

- *Experienced practitioners* who develop object-oriented application software. By our definition, they are familiar with at least one popular *object-oriented language*, disposing of basic project experience. They feel challenged to develop new *components* or *frameworks* in future projects, use existing frameworks, and integrate ready-made *components*. They understand the fundamental meaning of *design patterns*. With this background, experienced practioners are looking for an approach that allows them to combine and embed different—both old and new—concepts and technologies with their own experience to form a suitable approach. This book is intended to provide answers for experienced developers and "software architects" who have different questions relating to the design and construction process. To this end, we have tried to put our concepts and experiences together in the form of patterns and have arranged these patterns as collections in the relevant chapters of this book. The example used throughout this book, and its implementation with a Java framework, "the application example" will show how large-scale application software can be developed. Finally, we will explain how these concepts and construction approaches can be represented in an iterative and document-driven development process.

Project managers

- *Experienced project managers* running object-oriented projects who are keen to learn the ideas of an *evolutionary approach*, but who are looking for a feasible and reproducible project organization. They have found that rigid *waterfall models* merely create the *illusion* of a controllable project. They know that technical software know-how and tools are useless unless you also have some application-specific knowledge. They also understand that you cannot achieve persistent software development and high usage quality of your products unless you have technical concepts and excellent tools. They probably have explored the Unified Modeling Language (UML) and the Unified Process and have heard contradictory rumors about agile processes like eXtreme Programming. They may wonder how these technologies can be used in a specific project.

  We have elaborated concepts, guidelines, and document types for these project managers to help them prepare their own application-oriented approach for their teams and the entire organization. Our approach attempts to combine the proven strategies of evolutionary software development with useful aspects of agile processes. We thereby hope to achieve precise planning, high flexibility for changes, and constructive quality assurance.

Students

- *Committed students* who have a solid *education in computer science* and are looking for an integrated view of all aspects involved in object-oriented software development. They have studied relevant *technical tools* and know how to write code in an *object-oriented language*. Though they know the *UML notation*, they may have used it only in small examples. In this book we try to show such students our approach from practical experience, where our patterns will be useful. In teaching software engineers, we have often found that much depends on direct cooperation and joint experience in real-world projects. However, this is not sufficient. Our systematic and annotated collection of patterns in this book will help the student to grasp an overall view of design and construction problems. Again, we will use an example showing how to implement methodological and technical ideas in UML notation in order to build real-world projects.

Method developers

- *Critical method developers* who are challenged to select and tailor methods for use in real-world projects. They are probably interested in getting a detailed insight into the extent of the T&M approach and in order to estimate whether or not this approach can be transported to their use contexts. We think that purely conceptual representations are not sufficient, so we attempt in this book to collect current knowledge and opinions from various groups and experts working actively in promoting the T&M approach. The contents of this book will give experienced method developers an overview of the lessons learned in this approach.

Who should **not** read this book?

Not a programming textbook

- *Beginning programmers* who are interested in learning an object-oriented language like Java. This book is not a programming textbook. All programming examples included in the text itself are only sketches to illustrate construction ideas. For this reason, they are not even good examples of particularly "clever" constructions in a respective programming language. They are designed for experienced programmers who will know how to transport these examples into their own contexts. Since we want to focus on the concept behind constructions, this book does not include extensive code examples. Instead, we often use only class or interaction diagrams. The full code example can be downloaded at http://books.elsevier.com. Readers with no programming experience will find most of these constructions hard to understand.

Not a user interface design guideline

- *User interface designers* who want to learn more about designing sophisticated graphical user interfaces for object-oriented software. We discuss the overall design of interactive software within the frame of a comprehensive usage model. We illustrate this aspect/approach by showing the basic ideas of combining functionality, interaction schemes, and GUI elements. As the actual GUI design is not important for this discussion, our screenshots of user interfaces show very simple examples. From the viewpoint of software ergonomics, they may look rough or clumsy. Elaborate user interfaces are an advanced topic in their own right and are beyond the scope and purpose of this book.

Not a general introduction to object orientation

- *IT managers* interested in gaining an overview of object orientation. This book does not answer questions like "What is object orientation?" or "Is object orientation suitable for real-world projects?" This book assumes that such questions have been answered and understood. Even the sections in the first part of this book that discuss these concepts are intended for an advanced exploration of object orientation.

How to profit from this book

What are the *prerequisites* needed to profit from reading this book?

- Fundamental knowledge of software engineering.
- Knowledge of programming in an object-oriented language.
- "Access" to *Design Patterns* by Gamma et al.;[1] this seminal work on design patterns has coined the lingo of object-oriented software developers all over the world. We also deal extensively with the patterns described in Gamma's work and reference them in our own design patterns.
- Access to a book about UML notation.[2] Though we will explain our own interpretation of UML, we will introduce the notation only briefly.

1. E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*. Reading, Mass.: Addison-Wesley, 1995.
2. For example see, G. Booch, J. Rumbaugh, I. Jacobson: *The Unified Modeling Language*. Reading, Mass.: Addison-Wesley, 1999.

0.2 GUIDELINES FOR USING THIS BOOK

This section explains how to use this book and what its various parts mean. We recommend different ways to read this book for each of our target groups.

0.2.1 Different Ways to Read This Book

Reference guide

This book is conceived as a reference guide, ordered by topic. The first German edition demonstrated that formulating each chapter independently, that is, each with its own self-contained topic, was a good idea. References to other chapters are intended to represent further reading on a specific issue. Chapters 7 and 8 are built as pattern collections. Of course, each chapter makes a different presumption of the reader's knowledge of the topic dealt. But in general, interested readers will find answers to their questions in each chapter without having to read the entire book.

On the other hand, nothing in this book prevents you from reading it consecutively from beginning to end. We arranged the chapters so that you should be able to identify some direction of development. We also hope that there are not too many irritating redundancies. However, since this book was mainly designed as a reference guide, there may be some redundancies, which is another reason why we opted for a the pattern form of organization rather than a strictly linear exposition.

Pattern-type organization

Chapters 7 and 8 of this book are organized as pattern collections in the sense of the familiar design patterns. At the beginning of Chapter 7, we will introduce the pattern form to be used. You will also find a pattern-type structure in the rest of this book, which, we thought would facilitate reading. In general, we used the following structure for this book.

- **Introduction**
  This section is a short summary of what the chapter discusses and what the special lessons to be learned are. It often includes a note about the target group that would profit most from that section.
- **Definition**
  This section defines important terms introduced in the chapter.
- **Background**
  This section contains background information, that is, it reports about the state of discussion on the issue concerned. It also includes historical comments or cross-references to other topics. This section provides food for thought and is aimed at readers interested in getting more detail on a topic.
- **Discussion**
  This section discusses concepts introduced in the chapter as well as the pros and cons of the solutions.
- **T&M Design**
  This section gives instructions for the specific development process. It normally includes recommendations as well as positive and negative experiences gained from real-world projects.

0.2.2 Recommended Reading by Target Group

Software developers

Experienced *practitioners* should read the following sections:

- The first part of this book is conceptual and forms the basis for a common language between the authors and their readers. Experienced developers will find the terms and concepts used throughout this book. The basic terms and definitions pertaining to object orientation are clarified in Chapter 1. On this foundation, we go on to explain and complete the concepts of the T&M approach in Chapters 2 to 5 in order to establish a common language for development teams.
- Chapters 7 and 8 include conceptual and design patterns that have been completed by construction approaches for interactive software, thus it represents the most important part of the book for experienced practitioners. Depending on the issues and problems of concern, they then will focus on particular sections of Chapters 9 to 11 and deal with the implemented example extensively.
- Chapters 12 and 13 look at the process side of our approach. They explain how to get the T&M approach working. They also include our proposals for dealing with application-oriented document types and their correspondence to the UML notation. Thus this is also a particularly interesting part of the book for developers.

Project managers

Project managers should read the following sections:

- This target group will find sections on models and design metaphors, application-oriented software development, and software development modeling in Chapters 2 to 5 of this book. They will learn what the T&M approach is meant to be.
- This target group can skip Chapters 7 to 11. However, project managers will find useful definitions of terms.
- The last two chapters of this book are conceived for project managers interested in implementing an evolutionary document-driven approach in a planned and controlled way. In Chapter 11 they will find the principles as well as various techniques and strategies of our iterative evolutionary process. They will see how this process fits in with relevant and proven aspects of agile processes like eXtreme Programming. Chapter 12 introduces the appropriate document types. Managers will learn how UML can be taken to work within an evolutionary strategy and where additional application-oriented document types make sense.

Students

Students should read the following sections:

- The first 6 chapters provide the defining basis. Students can check whether or not their knowledge will allow them to understand our concepts. It is necessary to read this part of the book to better understand the rest of our discussion.
- This target group should work through Chapters 7 and 8 systematically, because the pattern collection represents a step-by-step introduction to building interactive software. In addition, they should analyze the corresponding examples. The other chapters discuss advanced issues that this target group may read as needed or according to their interests.

- This target group can read Chapters 12 and 13 selectively; they deal mainly with aspects of using notations for application-oriented modeling and applying this approach in evolutionary and cyclic project models.

Method developers should read the following sections:

Method developers

- The first 6 chapters provide an important overview of the basic concepts of the T&M approach, but method developers can skip Chapter 1, which deals with the object metamodel.
- Of particular interest for this target group is Chapter 7, because it explains our design approach.
- We recommend that developers read Chapters 12 and 13.

The EMS example

The *Equipment Management System* (*EMS*) example described below is used throughout this book. We intentionally kept this example small and easily manageable so that readers would be able to understand its professional and technical aspects. This example, which appears in various places in the text of this book, serves to illustrate the basic idea of the T&M approach. Note that we have not even tried to turn this example into a "realistic" application system. This holds true both for the Java code and for the user interface examples we use. They primarily serve as simple and clear illustrations of the issue we are introducing. Thus they are neither meant to be efficient code nor good interactive GUI designs.

A complete implementation in Java, using the JWAM Java framework, can be downloaded from the website of this book.[3] Whenever we discuss the EMS example we denote our discussion by a note in the margin.

Practical project examples

We tried to create a "real-world look and feel" to our book by using examples from our project practice. For example, we repeatedly describe designs produced for banks, insurance companies, or hospitals. Again, we will highlight each example in the margin.

To build the examples included in this book, we used mainly Java and some C++ and Smalltalk as programming languages. As mentioned above, we have no intention to teach programming in these languages. We used these popular programming languages so that the reader will get an impression of how a construction might look in the respective language.

Programming languages and notations used in this book

The notations used in this book are based on the de facto standard defined in UML (Version 1.5). We assume that most of our readers are familiar with these notations. Chapter 12 includes a description of the notations and the special variants used in this book.

0.3 THE WRITING PROCESS AND SOURCES

This section explains how we wrote this book and the sources involved. It makes clear that the writing of this book was motivated mainly by two goals: first, to bring together the practical work and experience of many different people from many professional contexts, and second, to produce a uniformly designed and written work.

3. http://www.mkp.companions/1558606874/.

0.3.1 Motivation for This Book

We developed our particular way of writing and publishing a book during our work on the first German edition. Since we first published a comprehensive presentation of the T&M approach in 1994, much has happened, which naturally pleases us. Many design discussions in various cooperation projects, new design ideas, and a changing technical foundation all had their influence on the T&M approach. We held discussions in different contexts and assessed our experiences. In 1998, the time was right for a design and construction book on the T&M approach. This book reflects our view of the state of the art.

The editorial concept

However, several years have passed since the publication of the German edition, and many new results have accumulated. Therefore, this English edition has been completely revised, but in it we still use an editorial concept similar to that in the first German edition:

Lead authors: We agreed on a list of core topics. Each topic was accepted by one or two persons in the role of lead authors. They fully revised this text and coordinated the work of the other authors.

Coauthors: Each coauthor contributed essentially to this book. This can mean that he or she was an author (or lead author) of the German edition, or that he or she made a major contribution to the English edition.

Primary author: Beyond authoring major parts of this book, Heinz Züllighoven edited all sections of the book, ensured uniform style, and established cross-references.

Translator: American native speaker Angelika Shafir translated and adapted the German text to English. All modifications made in the course of the translation were discussed with the lead authors and the primary author, in order to minimize semantic shifts from translation.

Editor: Tim Cox served as the editor for this edition of the book for the publisher. We discussed with Tim the concept of the revision for this edition as well as the readability and coherence of every chapter after its translation. Stacie Pierce, Richard Camp, Mamata Reddy, Brandy Palacios, and Suzanne Kastner then greatly helped to bring the manuscript into production.

Reviewers: After translation, every chapter was reviewed by specialists in the field. These reviewers contributed significantly to the second round of revisions.

With this rather complicated and labor-intensive editing and translation process, we hope to have done justice to the various contributors and their contributions to the development of the T&M approach without denying the reader a book with uniform style.

0.3.2 Sources

The coauthoring arrangements do not by a long shot cover all sources of the T&M approach for this book. Therefore, we list here those persons whose publications and ideas also contributed significantly to this book. Since many of these sources were published in German only, we do not provide a detailed list of references.

First, we want to mention the spiritual fathers of the ideas behind the T&M approach: Reinhard Budde and Karl-Heinz Sylla.

The following people, with their dissertations and theses contributed greatly to the further development of the T&M approach:

Christian Beis, Holger Bohlmann, Marlies Eschner, Malter Finsterwalder, Boris Fittkau, Frank Fröse, Michael Gatzhammer, Achim Gehrke, Thorsten Görtz, Andreas Hartmann, Andreas Havenstein, Andreas Kornstädt, Holger Koschek, Timm Krauß, Sven Lammers, Klaus Müller, Lara Niemeyer, Fabian Nilius, Björn Ostermann, Michael Otto, Jörg Penning, Thomas Pfohe, Sabine Ratuski, Joachim Sauer, Norbert Schuler, Michael Skutta, Eike Steffen, Olaf Thiel, Horst-Peter Traub, Matthias Witt, and Ulfert Weiss.

1 Introduction

1.1 APPLICATION ORIENTATION — THE SUBJECT OF THIS BOOK

Recent major changes in global markets have encouraged, if not forced, many corporations to review their corporate strategies, with "customer orientation" being the ubiquitous catchword. This section discusses the motivation behind customer orientation, what it means, and why it requires a new approach toward software development—the application orientation. This discussion demonstrates that application orientation relates both to software products and how projects should be managed.

1.1.1 Motivation

The global economic environment has motivated many companies to orient themselves more closely to their customers. Customer wishes, requirements, and expectations are central factors in your corporate strategy, and products and services have to be tuned to this environment. The more individual you want your products and services to be, the more specialized the application systems you use in your organization have to be. This flexibility translates into high demands on any software development project.

Does this sound far-fetched? Not really. In fact, these could be the introductory words of a speech about the *Tools & Materials Approach (T&M)*. They could provide the economic background to motivate potential listeners or readers in an intuitive way to deal with this approach. Note, however, that these words mean more than just a marketing strategy. Customer orientation not only has become one of the modern catchwords in the business but is also a term expressing a visible change in many organizations. Customer orientation means understanding that for an organization to be successful in the current harsh economic climate, it has to distinguish itself from its competitors, and customer orientation is what can help it achieve this goal. As Peters and Waterman have observed, customer orientation may have been the trigger that released the current reorganization processes for the following reasons:

Customer orientation

- Competition is becoming increasingly fierce, so that corporate managers are forced to rethink their assumptions.
- Many products and services are introduced to the markets, so that a large number of these become interchangeable from the customer's perspective.

- Poor service drives customers away, while many organizations overlook the fact that the acquisition of new customers is generally more expensive than keeping their existing customer base.
- Companies increasingly look for ways to distinguish themselves with above-average service or additional service offers to win more satisfied customers and a more loyal customer base.

In this tense situation, customer orientation translates into a continuous effort to improve *customer satisfaction* in order to achieve long-term customer loyalty and ensure the *company's success*. As a consequence, success-oriented organizations have been critically reviewing their work processes, organizational structures, and corporate strategies. One of the most important factors within this reorientation effort is information technology. It has quickly become clear that information technology must only be a means to an end and never an end in itself.

In the field of *software development*, this scenario challenges us to ask how traditional approaches, methods, and principles can contribute to customer orientation. The practitioner does not normally have a clear answer to this question, encountering serious doubts when taking a closer look at the technologies and tools used in the real world. Can new products and approaches give rise to hope for a decisive improvement of the current situation? When observing the current euphoria about Java and J2EE, application servers, and Internet services, you may easily think so.

Application orientation

The authors of this book are more critical. Methods, technologies, and tools are just the means to an end. We think that the idea of customer orientation in an organization should lead to *application-oriented* information technology. This means that an organization's staff can act in a customer-oriented way provided that they have the corresponding technical support at the workplace. Let's look at application software as a relevant and integral part of information technology in the sense of our discussion. We can then ask how this application software should be designed to actually contribute to customer orientation. Our short answer is that only application-oriented software development can supply the prerequisites required to achieve this goal.

The central purpose of this book is to explain what application-oriented software development is and how it can be conceptually and constructively designed by use of object-oriented means.

Large-scale application software

When we talk about *application software* in this book we mean *large, interactive, and long-lived software systems*. By our definition, "large" means object-oriented software systems with more than 1,000 classes developed by a team composed of at least five to ten people over a period of more than a year. "Interactive" means application software that can help do the job via different technical and marketing channels within a specific application as a means to its end, where the way the program runs is influenced by both user intervention and system feedback. And "long-lived" means that this application software can be used over a period of several years and has to be adapted continually to business and usage conditions as they change. Finally, we consider a piece of application software to be "large" when more than one project within the same application domain works with components or versions of that system.

We also think that *frameworks, platforms, and components* are an indispensable part of these dimensions. Frameworks (nowadays often called platforms) form the architectural backbone of the system, and large software systems cannot be built at an acceptable cost and with satisfactory technical quality without this backbone. But frameworks are no panacea. Both the development and the use of large frameworks are extremely complex, so that many software projects are simply overtaxed. For this reason, we will explain how you can develop and use frameworks. The attentive reader will note that application orientation plays a decisive role once again. We think that one of the major problems with existing frameworks is their technical orientation. In the course of this book we will show how application-specific concepts and structures can influence the architecture of software systems on both micro and macro levels.

Since the beginning of the nineties we have observed vehement discussions about the use of components as an alternative to frameworks. As Clemens Szypersky put it, "A software component is a unit of composition . . . [It] can be deployed independently and is subject to composition by third parties." At this point, we should ask how and in what context this composition is supposed to take place. We will explain that components and frameworks can complement one another well.

Another thing that plays a major role in the description of frameworks and components are *patterns*. Relying extensively on the seminal work of Gamma et al. and our own work, the main part of this book will describe *conceptual and design patterns* that are used to develop interactive application software. We will use the presentation means of the Unified Modeling Language (UML) for technical figures.

However, we don't want to stop at the architecture of a software system and its construction. We think that application orientation requires an *altogether different* approach to software development. There is currently much discussion about the role and proper use of the Unified Process (UP). We will show that an application-oriented approach is compatible with the principles of UP. In fact, this approach can be seen as a user-oriented interpretation of UP. Among other things, this new approach will lead to different software products and a different development process. This book is about both of these issues.

1.1.2 Structure of This Book

This book is divided into three main parts:

- The T&M approach did not emerge ad-hoc or without any concept. The first part of this book, which includes Chapters 1 to 6, describes the T&M idea, what motivated it, and which important object-oriented concepts and modeling rules it uses to achieve an evolutionary approach.
- Part 2, encompassing Chapters 7 to 11, discusses in detail how to build interactive application software. We will first explain how you can use metaphors to design an interactive system. Then we show how to systematically convert this conceptual design into the implementation of tools and all the other elements of the T&M approach. Chapter 9—a central chapter of Part 2—describes a framework-based architecture for large application systems. The remaining chapters of this part discuss other approaches and constructions; important issues include the support of cooperation and coordination, and persistency.
- Part 3 begins with Chapter 12, which discusses the development process; Chapter 13 describes document types. We will explain how we see industrial software projects from the management perspective, and how these management aspects can be planned within an evolutionary approach. One of the central issues discussed in this context are the document types proposed in UML and the

basic approach defined in the Unified Process. In this context, application orientation requires a set of document types that extend UML. Agile processes like eXtreme programming are often seen as an alternative to "heavy-weight" strategies like UP. Having participated in the early discussions about agile processes, we disagree. We will introduce a balanced integration of agile and evolutionary approaches, pointing at their underlying unifying concepts—the need for software developers and users to understand each other and to cooperate. Therefore, it appears meaningful to interpret the general statements of the Unified Process from an application-oriented view. This issue forms the focus of this part.

1.2 THE TOOLS & MATERIALS APPROACH (T&M)

This section provides a brief overview of the most important ideas behind the Tools & Materials (T&M) approach. It describes models and central design metaphors, such as tools, materials, automaton, and environment, as well as the general principles of and useful document types for our approach. It discusses why the T&M approach differs from many object-oriented methodologies.

1.2.1 The T&M Approach in a Nutshell

The T&M approach attempts to achieve application orientation in software development. *Application orientation* focuses on software development, with respect to the future users, their tasks, and the business processes in which they are involved.

Application-oriented software is thus characterized by high usage quality, in other words, on the characteristics that a system should show when actually used. We can identify some important characteristics of this type of software:

- The system functionality is oriented to the tasks to be solved within the application domain.
- The system is easy to use and manipulate by its designated users.
- The processes and steps defined in the system can be easily adapted to the actual requirements, depending on the individual application domain.

Based on object-oriented design and techniques, this approach combines many different elements, such as a model with design metaphors, application-oriented documents, and an evolutionary development strategy with prototyping. But the T&M approach is not only a collection of best practices. The underlying idea links its different elements in a way that makes designing good application software easier. You start by choosing a guiding metaphor and fleshing it out by appropriate design metaphors. So, for example, you decide to build a software system as an expert workplace for a customer advisor (guiding metaphor) in a bank. You add different form sheets, folders and trays as well as tools like an interest rate calculator or an account finder as equipment to an electronic desktop (design metaphors). Constructing the system you look at the patterns like "Interrelation of Tools and Materials" related to the design metaphors. They fit into the proposed model architecture. As you set up your project you will find guidelines and document types that put the application-oriented idea to work.

Building large object-oriented systems is a demanding task. Today, many developers have realized the need for a clear architectural concept for these systems. The central architectural idea behind the T&M approach is that the structures of the application domain and the software system should be similar. This *structural similarity* means that objects and concepts from the application domain are taken as a basis for the technical software model. The result should be a close correspondence between the application-specific terminology and the software architecture. This is not a new idea as it was the initial spark for Simula, the first object-oriented language. But we took the idea from designing systems from the small ones (e.g., on the GUI level) up to the overall structure of large distributed systems.

Having similar structures for the two worlds, that is, the application domain and the software model, offers two major benefits. First, the application system represents the real-world objects and work-specific terms for the users. This means that these users can organize their work in a familiar way. Second, the developers have a development plan for their system. They get an idea of how to structure the "landscape" of the domain. At a more fine-tuned level they can put software components and application concepts in specific relationships, so that they can be easily identified when the software needs to be adapted to real-world changes.

To deserve the term "interactive," an application software has to combine a *task-oriented functionality* and *suitable handling and presentation means*. From this perspective, software development is a design challenge in the original sense: both the form and the contents have to match. This brings up the question of what "shape" a software product should take and how it should be handled to help complete the task. Software developers should be supported in this design challenge. Within the scope of the T&M approach, this support is provided in the form of a "guiding metaphor."

Model and design metaphors

A *guiding metaphor* for our purposes is a pictorial term that characterizes a unifying view in the software development process. As a familiar term it helps both users and developers to understand and design the application software. This means that it is not used for GUI design but to shape the functional and behavioral characteristics of the entire application.

Currently, our most successful guiding metaphor is the *expert workplace*. This metaphor has proven valuable in the large number of software projects we have conducted in the banking and service industries. These industries primarily support office work done by qualified staff, in other words, activities that, in addition to industry-specific knowledge and experience, require some amount of independent work and initiative.

When looking at the expert workplace as a guiding metaphor, the reader should realize some fundamental ideas related to it:

- A workplace is a place where people accomplish their work tasks.
- It is furnished or equipped with the necessary things for doing the job.
- An expert workplace is meant for a person who knows how to cope with the different work tasks of the job.
- The expert is the actor; the workplace simply offers the means to an end.

Other guiding metaphors, such as the functional or back-office workplaces, have also proven to be valuable in different working contexts or domains.

A guiding metaphor should be "granular" to ensure that it can really be understood. To achieve this goal, we use a set of matching design metaphors that describe that "leading" metaphor in more detail.

A *design metaphor* uses an object from the real world to describe a concept or a component of the application system. Such a design metaphor creates a common basis of understanding for developers and users, as it clearly refers to a common background of experience and represents an understandable term.

To match the guiding metaphor of a workplace, we use the following design metaphors: *tool*, *material*, *automaton*, and *work environment*. Obviously, these metaphors originate from the context of human work. When working, people take intuitive decisions between the things that represent their work objects, and things they use to get their jobs done. Accordingly, our definition of the fundamental metaphors, or tools and materials, is as follows:

What is a tool?

A *tool* supports recurring work processes and actions. It is normally useful for a different set of tasks and objectives. A tool is handled by its users according to the requirements of a situation and put aside when it is not needed. It does not dictate fixed work steps or processes. If it is a software tool, it allows users to manipulate work objects interactively.

What is a material?

A *material* is a work object that is manipulated so that it will eventually become the work result. Materials are manipulated by use of tools. Software materials embody "pure" domain-specific functionality. They are never used directly and are never active on their own. A software material is normally characterized by its behavior rather than its structure.

In an office environment, there are more than tools and materials. Today, many tedious work routines are done automatically. We thus introduced the automaton as an additional design metaphor:

What is an automaton?

An *automaton* completes a task that has been fully specified in advance and produces a defined result in defined variants. Once parameterized, an automaton can run without intervention over a lengthy period of time. A software automaton normally runs in the background. It is initially configured by the user, fed with additional information upon demand, or stopped in case of emergency. An automaton may have a simple interactive interface.

Tools, automatons, and materials need a place where they can be provided, arranged, and stored. This is the purpose of the work environment metaphor.

What is a work environment?

A *work environment* is the place where work is completed and the required work objects and tools are provided. Work environments embody a spatial concept and ordering principle. We distinguish between personal workplaces, allowing privacy and an individual arrangement of things, and generally accessible rooms in a work environment. In an organization, we can often define meaningful workplace types with a matching set of characteristic equipment.

What is cooperative work?

In the context of office work, and in many other domains as well, it is no longer sufficient to support individual work. Application systems for *cooperative work* represent new challenges to software developers. Our experiences in this field led to the conclusion that cooperative work can be supported by work tools and objects similar to those for the individual workplace. However, cooperative work involves new resources that address the special character of cooperation. This includes design metaphors like *archives*, *circulation folders*, and *dockets*.

This expansion of our design metaphors to include cooperative work needs to fit into the context of the selected guiding metaphor. In cooperative work, as in individual work, there are users who have to decide within their work situation whether they handle a predefined standard process, or whether specific changes to the workflow and the work distribution are meaningful. Other workplace types and forms of cooperation represent other requirements for cooperative work support. Based on the *cooperation types*, we have put together a basic choice of *cooperation tools and media*.

Evolutionary system development

For developers, application orientation means that they have to understand the tasks they are supposed to support with appropriate software systems. To achieve this goal, they have to gain access to the specialized knowledge and experience of the experts in the application domain. The keyword here is *evolutionary system development*, because this type of development process is oriented to close cooperation between the developers and the users. However, cooperation needs a foundation. Therefore, we use application-oriented document types and the prototyping concept.

Application-oriented document types and UML

Using application-oriented document types as a necessary basis to understand the concepts and tasks of an application domain is nothing new. What's important to understand is that these document types describe the application under development in the specialized user language. These document types are considered an expansion and specialization of the UML standard documents. We use the "classic" UML document types for the technical design. In addition, we use the following application-oriented document types:

Scenarios

A *scenario* is an interpretation of an organization's business use cases. It describes the current work situation, focusing on the tasks within the application domain and how these tasks will be completed by the use of tools and work objects. Scenarios are formulated by the developers based on interviews with users and other groups involved in the application domain.

Glossary

A *glossary* defines and reconstructs the set of terms of the specialized language in the work environment. Glossary entries are written by the developers parallel to the scenarios. They provide initial ideas about the materials needed in designing the system.

System visions

System visions correspond to use cases. They are at the transition point between the analysis of the application domain and the construction of the future system. Based on the scenarios, system visions anticipate the future work situation. They are oriented primarily to the developers and help build a common system vision. They describe the ideas about the domain functionality and how tools, materials, automatons, and other system components should be handled.

Prototypes

A *prototype* is an executable model of the application domain, representing an important and constructive enhancement of the system visions. A prototype makes the ideas about the system "tangible." In particular, so-called "functional prototypes" are of central significance not only in the developers' communication with the users, but also in giving developers the necessary experience and experimental basis for the technical construction of the system. This means that prototyping plays a central role in our approach.

To analyze cooperative work and design that is appropriate to application software, we need to extend the set of document types described above. Scenarios, glossaries, and system visions can be used to describe the different set of tasks and activities from the workplace perspective. What's missing is an overall view. To solve this problem, we have successfully used so-called cooperation pictures.

Cooperation pictures

A *cooperation picture* is an extension of use case diagrams, which are used for group discussion and workshops. Cooperation pictures are essentially based on well-understood pictograms that represent the work objects and information to be exchanged between all participating parties or "actors."

The development process

Once we have defined our guiding and design metaphors, together with document types and prototypes, we have a set of interrelated concepts and techniques on hand to support software development that is focused on the application domain. This means that the developers have all the prerequisites to start working towards the goal set: high usage quality. However, whether or not this goal can really be achieved depends largely on the right implementation of the concepts and the use of tools during the development process. Therefore, we think that it is mandatory also to prepare a set of instructions for the design of this process for the developers. The basic principle is then to look at software development not as a primarily technical or formal task but as a communication and learning process. Learning and communicating are evolutionary processes driven by constant feedback between the participants. This means that we agree with the basic principles of the Unified Process.

This evolutionary process between development and feedback includes basically all documents pertaining to that project. It is not a matter of completing milestone documents sequentially in a defined order, but of linking the analytic, modeling, and evaluation activities as needed. This obviously conflicts with the principles of the classic waterfall or phase models, but it can be harmonized with the core workflows of the Unified Process.

The evolutionary process, in our definition, is realized in the form of so-called author-critic cycles in all our projects.

Author-critic cycles

Our *author-critic cycle* means switching between analyzing, modeling, and evaluating activities, where the developers are normally the authors. Their work objects are documents and prototypes. The critics are normally all participating parties who have the required specialized or domain knowledge. In the sense of application orientation, these are normally the users. Ideally, the author-critic cycle is iterated quickly and often.

To better understand our approach, it is important to bear in mind that the problems identified in the feedback process determine what activities should follow next and which documents should be prepared. In this sense, there is no predefined sequence of documents and work steps. You could basically prepare any document at any given time. Of course, there is no doubt that the entire set of activities have to be planned and controlled in every project and cannot be subject to individual arbitrariness. What we should take home from these definitions is that the basic process itself is determined by application-oriented issues rather than by a technical mechanism.

1.2.2 T&M As a Method

This book discusses the T&M approach. Now, what is it? Shouldn't we simply speak of a T&M method?

What is a method?

A large amount of object-oriented literature contains a collection of graphical notations and associated instructions for use, including UML literature. Unfortunately, notations, diagrams, and graphical CASE (Computer Aided Software Engineering) tools are not sufficient to fully support a specialized software development process, because this notation does not tell you what should be modeled.

At the other end of the spectrum, you find books on methods, organized similarly to cookbooks or do-it-yourself handbooks for model construction kits, with instructions like, "Take this, do that, be careful about a third thing, and . . . your object-oriented application system is ready." We are somewhat skeptical about such books, because almost thirty years of software engineering history and experience have taught us that this type of instruction does not work in the real world. That's why we say: the methodical approach should provide a set of techniques and means of representation. In addition, there should be a set of instructions telling how to use these techniques and means of representation based on concepts.

In her seminal work, Christiane Floyd draws the following conclusion: despite all similarities, each software project is definitely different from its predecessors. So, for a method to be successful, you cannot simply apply it, you have to work it out in your project.

The T&M approach is a methodical framework

Consequently, the T&M approach is not a ready-to-use method but

- a view of object-oriented application development;
- a collection of proven construction, analysis, and documentation techniques in the form of patterns;
- a description of matching concepts and architectural models;
- an evaluation of different and extensive project experience; and
- a set of guidelines to develop a concrete construction technique and a matching approach.

In short, the T&M approach is designed to help elaborate a method tailored to a specific project and a specific development organization.

Views

To achieve this goal, it is important for our readers to be able to reconstruct our views to a sufficient extent. Views evolve on the basis of experiences and value concepts. This background cannot be represented in a handbook. Nevertheless, this basic understanding between the author and reader has to be established. For this reason, we comment on all concepts and constructions, show examples, and report on the experience we have gained from our real-world projects. We think that this will create the right context for our readers to understand the T&M approach and use it in their own work.

Originality of T&M Relationship to UML and UP

Finally, a word about originality. We never intended to create something totally new, or to knowingly distance ourselves from others with the T&M approach. Those readers who find known constructions, representation means, or ideas in this book should know that this was our intention exactly. After all, as software engineers we don't want to be "originality geeks," but instead rely on work that has proven reliable and meaningful. In this sense, we also do not distance ourselves from UML or the Unified Process. We rather interpret both techniques and concepts in the application-oriented sense, giving developers a set of instructions about how to handle these diagrams and principles in their project organization. The goal of this set of instructions is to combine high usage quality in our software with state-of-the-art software technologies.

1.3 PROJECTS BEHIND THIS BOOK

In the following we give an overview of the scope of the projects where the authors of this book were actively involved and in which T&M played a major role. The reader will thereby understand more of the background of this book and get a better feeling about the relevance of our approach to his or her everyday work.

1.3.1 The Scope of the T&M Projects

For almost fifteen years the T&M approach has evolved. A lot of people from different fields and organizations have contributed their ideas about software development and application orientation. But the essential driving force of our approach has always been project work. The experiences of many projects in many areas have both served as a usability test and as an inspiring source for new ideas or revisions of concepts and techniques.

The projects where the T&M approach has been used covers a wide area, from workplace solutions to technical embedded systems. Examples include:

- An IDE (Interactive Development Environment) for logic programming;
- Workplace systems for banking;
- Fleet management for a car rental company;
- Workflow editor for e-commerce;
- Certified aircraft engine software;
- Embedded medical lab system; and
- Maintenance management for municipal water works.

In the following, we will look at some projects in more detail.

An Object-Oriented Programming Environment for Logic Programming

In the mid-1980s in a software technology department of a national research institute, we developed a programming environment for logic programming in Prolog. The architecture of this environment featured the nucleus of many technical T&M concepts—we used tools and materials for the first time. Through discussions with users, we realized the importance of usage quality and a clear domain-oriented usage model. The programming environment was used for several years at universities and research institutes for research and training.

A Workplace System for Retail Banking

A software and service center for a major German banking group was faced with an urgent customer demand for a new workplace system that would comprehensively support their customer advisors. An initial software project along the traditional lines of a waterfall model and procedural programming had failed. A rather desperate management ventured to relaunch the project with object-oriented techniques and an application-oriented strategy. We trained and coached the team and consulted management. The T&M approach was used both for all basic constructions and for designing the different workplace types with their usage models. Within an incremental process of almost three years the workplace system was developed in several extension levels and is working at almost 2,000 workplaces in more than 300 banks. It received an outstanding reception by its users. Several external evaluations and reports showed the high level of usage quality.

Reorganizing the IT Department of a Banking Software and Service Center

Motivated by the unexpected success of the workplace system project, the software and service center just mentioned decided to reorganize their IT department. Object-orientation was chosen as the main technology for application development. All other IT activities were to be grouped around this strategic concept. We consulted IT management, trained and coached teams, and cooperated in major conceptual projects. Over a period of six years, more than 150 developers and technical staff were trained in object- and application-oriented concepts and techniques. We coauthored the company's project strategy and handbook. Together with the company's architecture group, we designed the architecture of their banking framework, which was a major source for the T&M model architecture. The consistent conceptual view of the T&M approach and the concept of structural similarity proved to be the fertile basis for the growth of a common development culture in the company. The company played an outstanding technological and strategic role among the software and service centers of the banking group.

A Fleet Management System for a Car Rental Company

The German branch of a European car rental company needed a system for the strategic management of their vehicle fleet. They used to work with their company-wide central information system. This system, however, gave no proper support for ordering, buying, or selling vehicles, or for the bookkeeping related to them. First, we analyzed the business processes involved and cooperated with the technical departments in restructuring these processes on the basis of cooperation pictures and presentation prototypes. In subsequent project stages we developed an integrated fleet management system (plus a specialized bookkeeping module) with interfaces to the central information system. Both management and users were convinced by the combination of usability and complex domain logic of the system.

A Graphic Workflow Editor for E-Commerce

An Internet company had the idea of a new e-commerce application by which the business organization staff could design and model e-commerce business processes. The application consisted of a graphic editor for designing workflow processes and a generator component that transformed the graphic models into Internet applications. We contracted for the design and implementation of the graphic editor. As a promotion show at an international IT trade fair was already scheduled, we had roughly three months to analyze, design, and implement a fully functional prototype. We set up a programming team of eight people who used an agile process incorporating the principles of eXtreme programming. The project was a complete success under all technical and domain aspects. We delivered the prototype with all requested features precisely on time and on budget. Unfortunately, the product never met a market as the Internet company became insolvent after the trade fair.

A Certified Aircraft Engine Software

A German race car engine-tuning company had the idea of using an augmented standard vehicle engine as the basis for an aircraft engine for sporting airplanes. A new electronic motor control unit had to be developed. This embedded software system had to be certified by national and international avionic authorities. We contracted for elaborating a development and quality assurance process that could be certified. In order to meet the very high demands for software quality and for a clearly documented and reproducable development process, we combined the main features of our application-oriented development strategy, that is, *Design by Contract*, our interpretation of eXtreme programming (especially engineering cards and test first, of Sections 12.4.2 and 13.5.2) and the UML document types. Within six months, we designed a meta-model and a software development process according to the FAA's (Federal Aviation Administration) RTCA (RTCA Inc is an association of aeronautical organizations of the U.S.A. from both government and industry) requirements, which was then certified by all relevant avionic authorities. The IT department was happy about the development strategy and the quality assurance process, which on the one hand fulfilled all the formal requirements, and on the other hand fitted well into the actual teamwork and programming practices. The new aircraft engine went into production and was a major market success.

Redesigning a Reverse Auctions Application

An Internet company successfully provides an application for reverse auctions, where buyers publish their tender offers and potential suppliers can place their bids. These suppliers then have the opportunity to undercut their competitors. The existing software for this application, however, had grown rather complex and was hard to understand and maintain. We analyzed the system, wrote a report, and proposed a complete redesign. The T&M conceptual patterns and the model architecture helped to clarify the pros and cons of the existing system and to highlight the improvements of the new design. Within several architecture workshops, the development team discussed our proposals and was able to work out a scheme for completely refactoring the reverse auction system.

An Embedded Medical Lab System

A market leader for medical labs wanted to develop a new generation of medical lab systems with a higher throughput of samples and a more efficient control system. The task was demanding, as a smooth integration of low-level automation, interactive analysis workplaces, and high-level system calibration was required. Major parts of the lab's hardware and basic software were custom-made. We redesigned the entire second-level automation software and the control system and added tailored integrated workplace systems. Starting with the core design metaphors of the T&M approach, we refined our tool construction and the workplace types. We were able to design a system with a clear technical and usage model that overcame a few severe shortcomings of the previous generation. While the prototypes of the system were very satisfactory, the first live tests unveiled major structural and performance problems in the database connection. The database mapping and the access interface had to be redesigned. Then the system ran smoothly and went into production.

Product Design System and Contract Management for an Insurance Company

A German insurance company, one of the middle-sized enterprises in this domain, wanted to redesign its host-based legacy system for contract management and at the same time implement a new concept for designing insurance products. We worked as consultants and software architects with their software team. Introducing eXtreme programming, we used techniques like pair programming, story writing, and tracking to improve programming skills, product quality, and project management. The T&M design metaphors helped to create a common vision of the future workplace system. The model service architecture was used as the basis for designing and implementing the new components and services for product design and contract management. The layer architecture provided the interface concepts for the connection to the host backend and its stepwise displacement. The first desktop tools with related services are operative, as well as an Internet portal for insurance agents.

Cooperative Workplace System for Corporate Banking

A major Swiss bank had the need to redesign its workplace system in the corporate customer department due to poor acceptance and performance. The existing system was implemented with a 4 GL (i.e., a 4th generation database manipulation language) system on top of a complex host network. As the system had been installed just a year before with the goal of improving the rather poor user support, the banking staff was not amused by the idea of yet another software project. In addition, a new development team of twelve with many graduates was hired, who had few skills in object-oriented design and little project experience. We were contracted as consultants and software architects. In a pilot project we combined object-oriented training with team formation and becoming acquainted with the banking domain. The T&M approach served both as a guideline to object-oriented development and as the overall view on application development. The actual project started after almost one year and ran for one and a half years. By the end the team size had doubled. An entirely new workplace system with different workplace types and several rather complex software tools was designed, implemented, and evaluated at some pilot banks. The different workplaces were linked with a clear cooperation model for the underlying flexible workflow management system. A few weeks before the actual roll-out the bank merged with another company of equal size. The top management decided that for the new company's corporate banking the business strategy of the other bank was to be used. Thanks to the layers of the T&M model architecture we had used, the team was able to substitute the entire business logic within three months, reusing most components for handling, interaction, and cooperation. The system successfully went operative.

Redesign of a Management System for Municipal Day-Care Centers

The *public* administration of the municipal day-care centers of a major German city was developing a management system. The small development team had little background for developing a complex system of this size and had chosen to use Visual Basic. We worked as trainers and software architects. The T&M model architecture and the conceptual patterns were a substantial help in getting across the idea of designing and implementing a large and complex system. As the programming skills within the team varied considerably, the distinction between tools, materials, and services were essential to establish a fitting division of work. The T&M design patterns proved to be on the right level of abstraction, as we could easily specialize them for a system implemented in Visual Basic. Most redesigned modules have seen integrated successfully and the project is in good progress.

A WORKPLACE FRONTEND FOR A TELEPHONY SYSTEM

A manufacturer was supplying server software for distributed telephony systems. They had a call-center application with a rather poor interface. We worked as software architects and developers. The task was both to design a highly usable frontend for call-center applications and an architecture for this software components that would fit into the overall architecture of the telephony system. We introduced the concept of different workplace types and design metaphors. Although the general characteristics of tools, materials, and automata were applicable to this domain, we found it useful to refine some of these design metaphors. So we proposed the conceptual patterns of adjustment tool, probe, and technical automaton. This led to an improved usage model, as the users could understand potential delays and missing reactions when they tried to adjust the actual telephone switches encapsulated in a technical automaton with a tool. The new call-center software was implemented using these patterns and was a success from the very beginning, as a network of distributed call-centers managed the nationwide marketing campaign of one of the biggest German IPOs in years without major problems.

1.4 THE EQUIPMENT MANAGEMENT SYSTEM EXAMPLE

In this section we introduce the *Equipment Management System (EMS)*. It is the main example we will be using throughout the book. This example is of small size and moderate complexity. It should meet the background of most software developers and shows a lot of characteristics of a domain, where interactive workplace solutions are useful.

Imagine that we are a small software company developing the JWAM framework as a platform for its project work. The JWAM team comprises approximately twenty people, with the usual fluctuation. Each developer has access to a networked workstation. In addition, there are the following devices: two servers, two printers, one integration computer, and one fax machine.

All developers have to rely intensively on their computers, so that the underlying infrastructure is a critical task. Our ficticious software company calls this task “Equipment Management,” and an *equipment manager* is responsible for it. The *equipment* manager finds himself in a constant target conflict: members of the staff keep asking for better-performing computers, but the budget for device procurement is limited.

The central tasks of our *equipment* manager include:

- Procurement of new devices: A device is normally procured upon the request of a member of the staff (see the scenario “Buying a new device” in Section 13.1). In such a case, the *equipment* manager has to carefully weigh the requirement against the budget. The employee’s old device is either disposed of or passed on to another employee (see the action study “Sorting out a device” in Section 13.1).
- Minimizing costs in device procurements: To procure a device the *equipment* manager has to get several cost estimates from different vendors to ensure economic management (see the scenario “Buying a new device” in Section 13.1).
- Upgrading existing devices: Existing devices are checked regularly as to their performance and upgraded, if necessary, for example, by adding main memory or hard disk capacity (see the system vision “Updating devices” in Section 13.5).

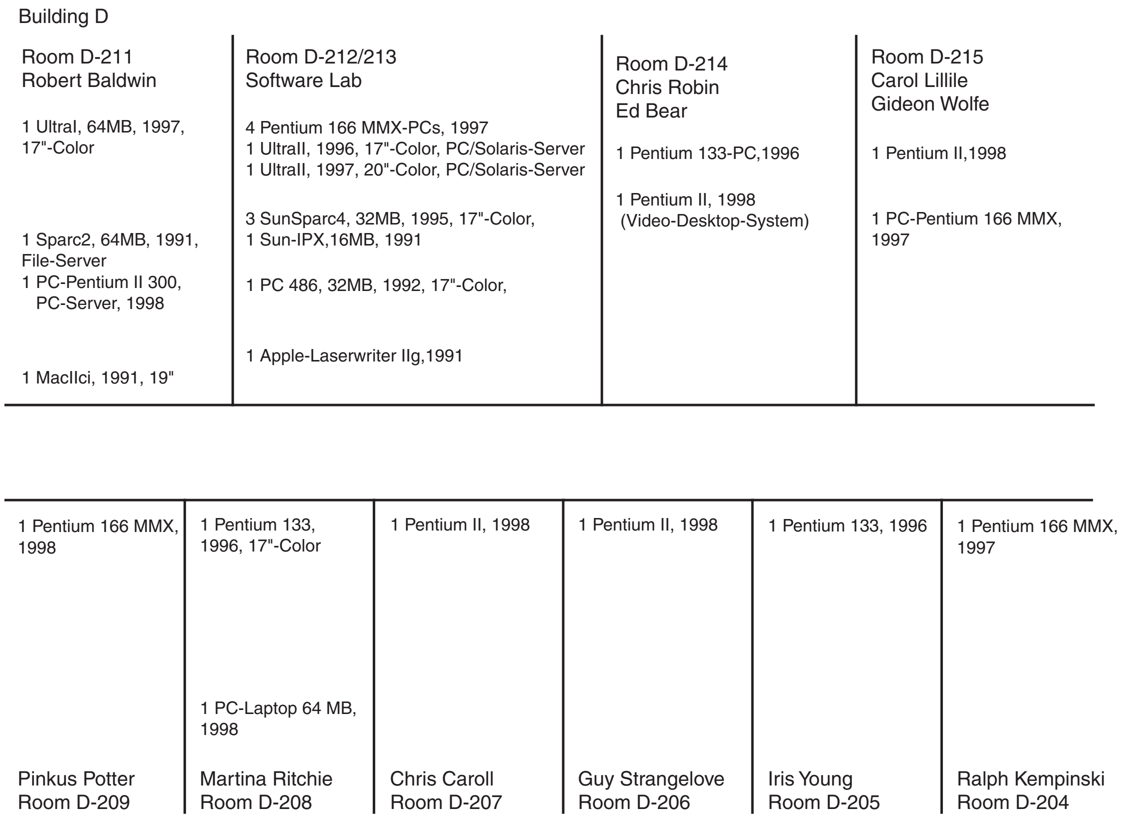


FIGURE 1.1 The EMS example's room plan.

- Updating the office plan: The team members are allocated to projects, and projects are located in adjacent rooms. Therefore, equipment and team members frequently have to change offices. An office plan showing a plan of all rooms with the team members and their equipment has to be updated with every change. A manual version of this office plan, that was used before the new system was developed, is shown in Figure 1.1.

The company estimates that the number of computers to be managed will double within the near future, because new members of the staff are going to join the JWAM team. This will, of course, increase the *equipment* manager's workload.

1.5 REFERENCES

D. Bäumer, G. Gryczan, R. Knoll, C. Lilienthal, D. Riehle, H. Züllighoven: "Framework Development for Large Systems". *Communications of the* ACM, October 1997, Vol. 40, No. 10, pp. 52–59.

This article describes the model architecture for large application frameworks based on the T&M approach.

G. Booch, J. Rumbaugh, I. Jacobson: *The Unified Modeling Language*. Reading, Mass.: Addison-Wesley, 1999.

The current standard work on UML.

R. Budde, H. Züllighoven: "Software Tools in a Programming Workshop". In C. Floyd, H. Züllighoven, R. Budde, R. Keil-Slawik (eds.): *Software Development and Reality Construction*. Berlin, Heidelberg: Springer-Verlag, 1992.

This publication is a short version of the dissertation of both authors, discussing the conceptual background of the T&M approach.

U. Bürkle, G. Gryczan, H. Züllighoven: "Object-Oriented System Development in a Banking Project: Methodology, Experience, and Conclusions". In *Human-Computer Interaction*, Special Issue: *Empirical Studies of Object-Oriented Design*, Vol. 10, Nos. 2 & 3. Hillsdale, New Jersey, England: Lawrence Erlbaum Associates Publishers, 1995, pp. 293–336.

This is a detailed report about the approach and experiences gained in our first and largest application project based on the T&M approach.

C. Floyd: "Outline of a Paradigm Change in Software Engineering". In G. Bjerknes, P. Ehn, M. Kyng (eds.): *Computers and Democracy: A Scandinavian Challenge*. Aldershot, Hampshire: Dower Publishing Company, 1987; pp. 192–210.

The seminal work, where the author also describes our understanding of software engineering.

E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*. Reading, Mass.: Addison-Wesley, 1995.

The seminal work on patterns, which formed the basis of our construction handbook.

T. Peters, R. H. Waterman: *In Search of Excellence: Lessons from America's Best-Run Companies*. New York: Harper & Row, 1982.

One of the best books (albeit criticized) that initiated the discussion about customer relations.

C. Szyperski: Component Software. Reading, Mass.: Addison-Wesley, 1997.

A popular and substantial book on software components.

2

The T&M Object Metamodel

2.1 THE OBJECT METAMODEL

2.1.1 Introduction

Work on object-oriented development projects requires all developers to share a common understanding of the basic terms relating to object orientation. In this respect, the authors of this book have a different conception from those of method books.[1] In addition, relying on an object-oriented programming language won't help either since various languages implement the concepts more or less cleanly. For this reason, we begin this section with a definition of our so-called object metamodel.

A *metamodel* essentially provides the elements and relations that can be used to build models. In the context of our work it means that an object metamodel helps us to understand the concepts behind, for example, UML diagrams or the Java language constructs.

Our object metamodel reflects the principle of application orientation. We will show how the elements of technical object-oriented models correspond to the concepts and notions used for domain-specific analysis and modeling.

By the end of this chapter readers will know what the authors mean by, for example, inheritance. They will also understand the importance of relating the terms of an application domain to a hierarchy of classes.

When this chapter is used by a software team, it helps developers to arrive at a common understanding of object-oriented modeling and programming from the perspective of the T&M approach.

1. The authors of various method books diverge considerably on the basic concepts that will be discussed here.

2.1.2 Definition: The Object Metamodel

What is an object metamodel?

Object orientation offers a set of concepts to describe domain-specific and technical models:[2]

An *object metamodel* describes

- **elements,**
- **relationships,**
- **element properties, and**
- **formation rules,**

which are all available for object-oriented modeling.

The most important elements are objects and classes, linked by a use relationship and inheritance. In addition, there are (mostly pragmatic) rules specifying how to link elements by relationships.

Among the most important elements in our object metamodel are classes. A *class* has specific properties, for example, an interface with admissible operations and a hidden description of attributes and implementations of operations. An *inheritance relationship* is one of the most important relationships used to link classes. An inheritance relationship has certain properties. For example, a subclass inherits all attributes of its superclass. Inheritance as a pure software concept may be used in many different ways. One important formation rule of our object metamodel says that you should use inheritance mainly to model domain-specific generic terms and semantic members of these terms. Therefore, our Equipment Managment System (EMS) example, `BusinessCard`, is a specialization of the generic concept, `Form`.

The rules of composition are thus of great importance. They give us guidelines for utilizing various elements and relations in modeling. They thereby support a common understanding of models and a clean use of the elements from a software engineering viewpoint.

2.1.3 Context: What's the Purpose of an Object Metamodel?

In the context of this book, we look at object-oriented application development from a software engineer's perspective, that is as a way of designing models (see Chapter 6). In fact, the core of the T&M approach is to create a domain-specific and technical object-oriented model, based on the objects and terms of that application domain, and to derive an object-oriented program from it. To do this, we need an exact understanding of what all the basic terms of object-oriented modeling mean and how they should be used.

At this point, you may be asking yourselves why we have gone to all this trouble, since each known programming language in practice implements such an object metamodel. This is true. Each program written in one of these languages shows what the elements of the language mean. But the problem with object-oriented languages is similar to those with method books. In recent years, we have used different object-oriented

2. When speaking of technical models in the course of this book, we always mean technical software models. When discussing the modeling of technical embedded systems, we will make that clear.

(and conventional) programming languages in our projects based on the T&M approach. What we have found is that each language implements a more or less different object metamodel. This leads to situations that we do not find useful, neither from the software development nor from the user's perspective. It has proven useful instead to turn a uniform object metamodel into the basis for conventions and program macros in our projects in order to achieve a common understanding of our modeling and programming efforts.

These issues led to the following requirements for our object metamodel: We have to define our object metamodel so that it can be used to describe the domain concepts of our future application. It has to be defined as closely as possible by the concepts of an object-oriented programming language so that we really can implement a domain model. And finally, our object metamodel should encourage software quality characteristics such as easy understanding and easy modification.

2.1.4 Context: A Classification of Programming Languages

When we talk about the basic elements of an object metamodel, we use the accepted classification of programming languages proposed by Peter Wegner. He defines a three-level model, where each level comprises the programming language characteristics that a language has on that level. The classification leads from object-based via class-based to the last level of object-oriented programming languages (see Figure 2.1).

According to Wegner's classification, a language is *object-based* if it creates and lets you manage objects as a primary language construct. In contrast, *class-based* languages have the class concept in addition to the object concept. This means that similar objects can be defined in appropriate classes, and that these classes are then responsible for creating objects as instances. Finally, an *object-oriented* language adds the inheritance

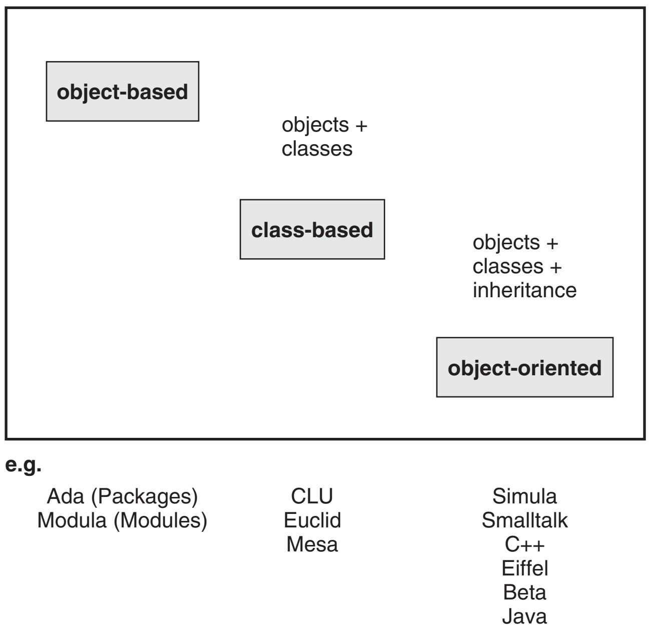


FIGURE 2.1
Wegner's classification of programming languages.

concept to the features of a class-based language. This inheritance concept allows you to create a hierarchy of subclasses and superclasses. Such class hierarchies have to be supported by a polymorphic-type system to be able to call an object of a subclass indirectly in the context of a superclass, that is, through a typed identifier. In the course of this book, we will only deal with object-oriented languages. Based on this classification, we will describe the important elements of our object metamodel in detail.

2.1.5 The Object Metamodel and the Software Model

One of the fundamental ideas of the T&M approach is that the terms of the software model can be put in relation to the terms of the application domain. Figure 2.2 shows how the elements of the domain and the software models interrelate. The processes called generalization, specialization, use, and composition lead to a concept model based on domain-specific objects and concepts of the application domain. This concept model can be mapped onto a software model, formed by the elements and relationships specified by the object metamodel. Note that mapping the concept model onto the software model does not break the model, because the structure and meaning of the elements used in both models are similar.

However, precisely this similarity represents a central rule of our object metamodel. For example, the model elements called term, generalization, specialization, composition, use, and hierarchy of terms correspond to the elements of the object metamodel called class, inheritance, aggregation, association, and class hierarchy. This means that terms are modeled by objects and that their behavior is modeled by operations, as shown in Figure 2.2.

This interrelation will become clear in this section as we continue interpreting the elements of the object metamodel from both the domain and the software perspectives. Note that we elaborated the software side of the object metamodel in more detail, because the main part of this book focuses on views relevant to the creation of the software model. Nevertheless, we will continue dealing with the domain modeling and the interrelation between the domain and the software models in later chapters (see particularly Chapters 6 and 9).

2.1.6 Definition: Objects

This section defines the term *object* from our two perspectives, that is, for the domain model and the software model.

FIGURE 2.2
Interrelations between the domain and the software models defined in the object metamodel.

Relations of the object metamodel:

Domain model	**Technical model**
Thing	Object
Interaction	Operation
Concepts	Class
Generalization, Specialization	Inheritance
Composition	Aggregation, Association
Sub- and Superconcepts	Class hierarchy

In the *domain model*, an *object* is a concrete thing or an ideal concept used in daily work and forms the starting point to model an application as software objects. These objects are characterized by their *behavior*, that is, how they can be handled and manipulated.

We distinguish *probing* and *altering* behavior:

- **What information can be drawn from an object? Which domain states are relevant?**
- **What changes can be affected in an object without destroying or transforming it into another object? What actions can I invoke on an object?**

In the *software model*, objects encapsulate data and operations to form a program component. They are the units of the *operative software system*. An object can be uniquely *identified* across the entire system, and it has a *state* represented in a private memory location of the object.

The operations of an object that can be seen from the outside form the *interface*. Each of these operations is defined by its *signature*. The *operations* assigned to an object provide information about that object and can change the object's state. These operations are the only means to read or change the state of an object. To activate an operation, you have to send a *message* to that object. In addition, a *contract* (see Section 2.3) can be used to make assertions about an object's behavior at the interface.

Figure 2.3 shows how we describe a device within our EMS example, described in Chapter 1.

Visible and encapsulated object properties

Both the structure and the behavior of similar objects are defined in the common class of these objects. This class also defines the visible and hidden properties of an object. From the software model, we need to ask ourselves to what extent the language we use supports encapsulation of an object's properties. For example, with the appropriate declaration, we can access the internal structure of an object from the outside in both Java and C++; Smalltalk, on the other hand, does not let you protect operations from being called by clients at all.

Class and type

Furthermore, a type is assigned to each object. In most statically typed object-oriented programming languages (e.g., C++), the type is defined by the class that an object belongs to.

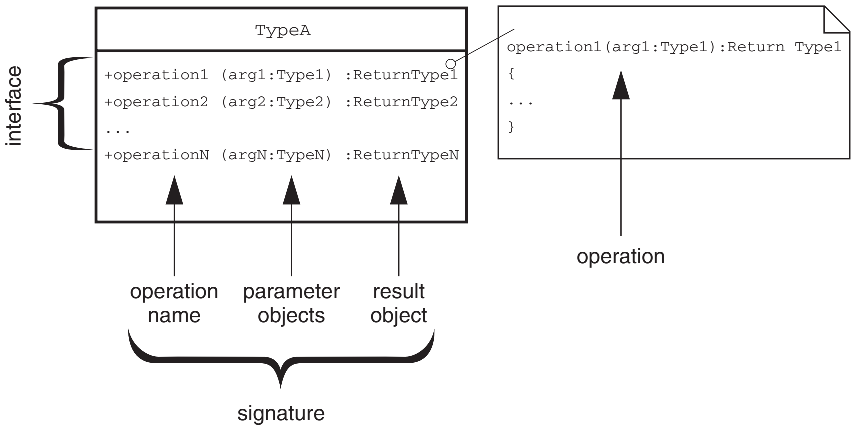
Creating an instance

Each object has to be created explicitly at *runtime*. To create an object at runtime, you call a creation operation of the relevant class. While they are created, objects are allocated in memory, and then perhaps initialized and bound to an *identifier*. Once created, this object is an *instance* of that class as long as it lives. Note that the type of an object also remains the same for an entire lifetime. On this basis it is important to

```
Device
    Name
    Classify
    Order at dealer
    Purchase at date
    Describe with text
    Due for upgrade?
```

FIGURE 2.3

Object-oriented, domain-specific description of a device from the EMS.



TypeA
+operation1 (arg1:Type1) :ReturnType1
+operation2 (arg2:Type2) :ReturnType2
...
+operationN (argN:TypeN) :ReturnTypeN
interface
operation1(arg1:Type1):Return Type1
{
...
}
operation
operation name
parameter objects
result object
signature

FIGURE 2.4

An object's interface.

understand the difference between static and dynamic types of object identifiers:

Static or dynamic identifer types

Static identifier type: **To be able to use your object in a program, you have to bind it to an identifier, using a reference. This identifier obtains a static type (in statically typed languages) by declaration.**

Dynamic identifier type: **At runtime, due to polymorphism, objects of a subtype can be bound to an identifier by either assignment or parameter passing. This defines the dynamic type.**

Objects are direct instances of their creating class. This means that, at runtime, these objects have the structure of attributes (also called fields) defined in the class. Such attributes can contain values (see Domain Values in Section 2.6.5) or references to other objects.

State

As mentioned above, each object has a *state*.

A *state* is specified by the particular[3] values of the object's attributes and references. The state of an object is protected against access from the outside; it can be probed or altered only by operations defined at the interface.

Interface, visibility

Each operation you declare for an object defines the name of this operation and its argument types and return type. All these elements together form the *signature* of an operation. The set of all signatures of the public operations of an object form its interface (see Figure 2.4), where *public* means that these operations are visible to the outside. In contrast, an *internal* interface can be accessed only by the object itself.

Services

The interface of an object defines its *services*. We expect from an object to offer a set of domain services. The services an object offers at its interface are often realized so that services provided by other associated, that is, referenced, objects are used. In this connection, we often speak of a *call-in* or *incoming interface*, as opposed to a *call-out* or *outgoing interface*. For a programming language, the call-in interface is always explicit

3. *State* refers to the values of all attributes at a given point in time.

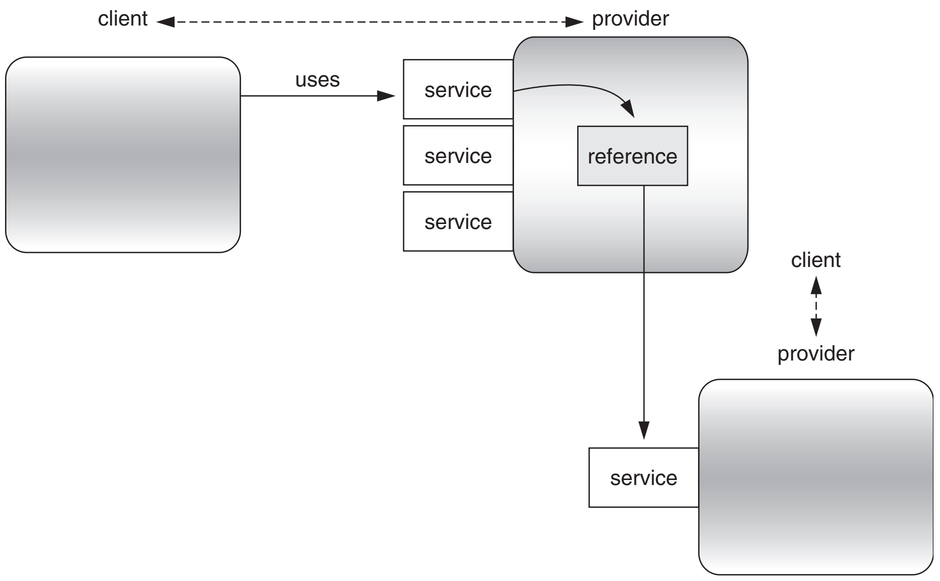


FIGURE 2.5
Objects acting as providers and clients of services.

(because it is the public interface of an object), while the call-out interface can normally be determined only by analyzing the program code.

The idea of *services* supplies a basic interpretation of the use relationship between objects: one object offers services, acting as a provider, and another object, acting as a client, uses these services. An object can be both a provider and a client for other objects (see Figure 2.5).

We have to distinguish the *interface* of an object and the *call* of services over this interface, both as terms and as concepts.

Operations, messages, routines, procedures, and functions

Conceptually, the interface's elements are called *operations*, that is, executable steps or activities running on a computer system. In terms of programming languages, these operations are implemented by the constructs of each language and are then called *procedures, methods*, or *routines*.

When we *call* an operation, we often speak of sending a *message* to the called object. Such a message includes the identifier of the object (as the addressee), the name of the operation, and the call parameters. When polymorphism is used, then the respective operation in its specific implementation is executed at runtime. This is also called *late* or *dynamic binding*.

When handling objects, we often find that operations cannot always be called in any state of the object. The reason is that an object obeys a certain use or state model, which can be explicit (e.g., as a finite state automaton) or implicit. We speak of the protocol of an object:

Protocol

- **An object's *protocol* is a set of rules, defined processes, or operation calls that each client has to observe to enable the object to render certain services. This depends on the state of the object. To be able to observe a protocol, the client must at least know the appropriate interface.**

Occasionally, the literature dealing with object orientation uses the terms *protocol* and *interface* synonymously. Our definition of *protocol* originates from the field of distributed systems, where the term describes the interplay beween two partner instances, which, in our sense, provide a bundle of services. However, popular object-oriented programming languages offer no way of representing a protocol at the interface, except for assertions in Eiffel.

Although the interface of an object defines the operations and perhaps attributes of the object that are visible from the outside, *encapsulation* means that you cannot access any property not explicity exported from the outside. This is normally called *information hiding*.

Encapsulation, information hiding

Information hiding **in object-oriented languages means that encapsulation is used to hide the specific representation of an object's state, that is, its structure. Dave Parnas recommends localizing design decisions in an object. The clients of an object depend only on the elements of the interface. You can change the implementation without losing the object's consistency from the client's perspective.**

The encapsulation concept also means that, though the client knows the signature of an operation and can draw conclusions about that operation's behavior from the contract model (see Section 2.3), the implementation is totally hidden. In this respect, any change to the implementation should be made in such a way that no change in the specified *behavior* is visible from the outside.

2.1.7 Discussion: Object Identity

It is important to understand that an object's identity has nothing to do with it being *addressable*. The fact that an object is addressable means that you can access an object from within one or more contexts (which might then lead to the so-called dynamic alias problem; see Section 2.6.2). Addressability is an external object quality and depends on the context.

In contrast, the *identity* of an object allows you to uniquely identify this object, regardless of the path you have to use to reach this object. From a *software-specific* stance, identity means first and foremost that whether or not two identifiers point to the same object or to two objects in the same state can be determined on the level of programming language. This means that identity is an internal object quality, that is, it belongs to the object, regardless of its context and the way its structure and operations are defined.

In addition, we need a concept to obtain a *domain-specific* identity. This domain-specific identity has to be independent of the software-specific identity of an object. Two objects with a different software identity can have one single domain identity. For example, a bank customer can be a debtor in one context and a portfolio owner in another. The identity of that person must be unique, regardless of the behavior in the respective context. This means that the developers have to be able to express what they mean by identity and how this identity relates to equality: Do they mean equal values, equal behavior of objects, or equal domain identity of an object? Note that the domain identity can never be a global property of an object. It can be modeled in a meaningful way only in dependence from the application domain. We will propose a possible solution to this problem in Section 9.4.1.

2.1.8 T&M Design: Structuring an Interface

As mentioned earlier, objects have a state that can be probed or altered. Depending on this state, sometimes operations cannot be called for software and domain reasons. To better understand this relationship in the domain and software models, the domain description of how objects are handled and the description of their interfaces should be arranged similarly (see Figure 2.6).

- From the domain-specific view, an object is organized as follows:
  - *Instructions* (in the sense of commands, actions) change the state of an object, resulting in procedures.
  - *Requests* supply information about the object in the form of domain-specific result objects, resulting in functions.
  - *Tests* are a special form of requests that probe the domain-specific state of an object and return a Yes/No answer, resulting in predicates, which are used in assertions.
- From the software-specific view, the interface of an object is organized as follows:
  - *Procedures* alter the (externally visible) state of an object. This change of state is not necessarily admissible at any given point in time. It normally depends on the actual internal state of an object.
  - *Functions* return result objects, without changing the externally visible state of the object, that is, a function always returns the same result, while the state and arguments of the object remain the same. A function can be *partial*, if it returns a result only for specific arguments and object states.
  - *Predicates* probe the state of an object and return a boolean result. They do not change the state and can be called at any given time.

We distinguish the concept of procedures, functions, and predicates rather than using a syntactic differentiation of a signature in input, input/output, and result parameters. Nevertheless, it is important to see how this differentiation is supported by a programming language to better understand our concepts. For example, in C++ each operation has the syntactic form of a function, and there is (currently) no primitive boolean type; in Smalltalk, on the other hand, each operation returns a result object.

2.1.9 Definition: Classes

We define a class as follows:

In the *domain* model, *classes* model the concepts and terms behind the objects we use in our daily work. This means that a class is an abstraction of similar *objects*,

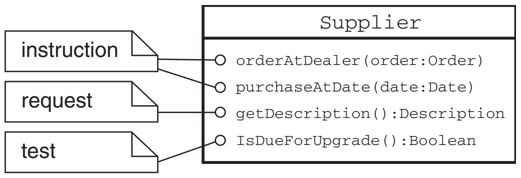

Supplier
instruction
request
test
orderAtDealer(order:Order)
purchaseAtDate(date:Date)
getDescription():Description
IsDueForUpgrade():Boolean

FIGURE 2.6 Organizing an interface in statements, requests, and tests.

based on the common behavior of different domain objects. The similarity of terms can be thought of as a generic term. Such *generalizations* or hierarchies of terms are part of the respective domain language, which forms the basis for cooperation and furthers a better understanding of the application domain. Hierarchies of terms are modeled by *subclasses* and *superclasses*.

In the *software* model, a class is a piece of program text describing the fundamental properties of the objects it can create. A class is defined by its name, its inheritance relationship to its superclasses, and a set of object properties (see Figure 2.7). These properties include the interface of objects and their internal implementation by algorithms and data structures. This means that a class defines the creation and behavior model of its instances. An object is always an instance of exactly one class.

Thus, in addition to the behavior at runtime, a class also defines how its instances are created and initialized. Each object you create differs from all other instances of a class in that each one has its own memory location. In the example shown in Figure 2.7, this would be for each `Room` object the reference to the respective lists of devices.

Encapsulation is used to protect the internal representation of an object against inadvertent use. For example, the reference to the object with the `devices` identifier in Figure 2.7 is not visible in objects belonging to the `RoomMap` class; it belongs to the internal representation.

Generic operations

Classes define *internal interfaces*, in addition to the public interface. The internal interfaces of a class are first the so-called *inheritance interface*, which is used only for

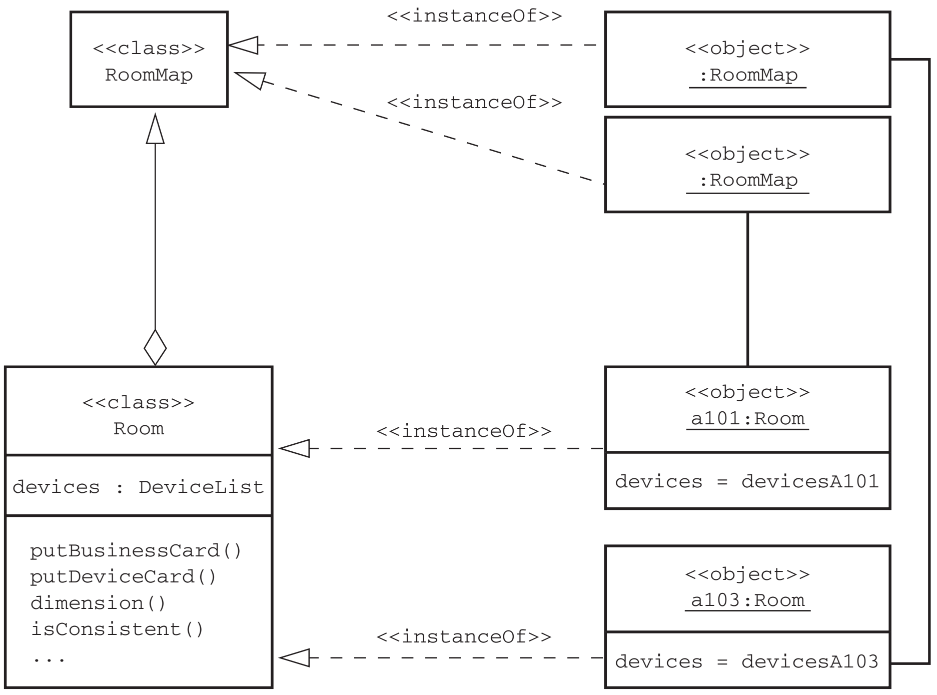


FIGURE 2.7
Classes describe the interface and representation of their instances.

subclasses of a superclass and not by clients, and the *private interface*, which denotes the operations for use within one and only one class.

Many programming languages can protect, that is, hide, their internal interfaces by use of language constructs (e.g., `protected` and `private` in Java or C++).

You often find in the literature and programming environments that a reading and a writing operation are supplied for each attribute of a class. These are *generic operations*, not to be confused with "generic components" or "genericity":

> ***Generic operations* are normally a pair of read and write operations (`set` and `get`) for each attribute of a class.**

2.1.10 Discussion: Generic Operations

Generic operations are used in graphical interactive programming environments (e.g., VisualAge) to make application objects "accessible" for graphical interface elements. The same approach is often found in database application software.

These generic operations can violate the information-hiding principle at the public class interface, because they publish the internal structure of an object, at least in part. In addition, they can easily destroy the domain consistency of an object. In general, we recommend not using such generic operations.

The Bank example

Let's look at a class, `Account`, where the generic operations could be `setBalance` and `getBalance`. When designing the class, the developer will have considerable troubles to implement the bank-specific handling of accounts using these operations. A better way would be to use `deposit`, `withdraw`, and `calculateFees` to define the conditions when a balance may be modified.

2.1.11 T&M Design: Generic Operations

Note that the T&M approach does use generic operations, provided that they are used at the internal interface.

It has proven to be a good idea from the software viewpoint to use generic operations to implement all probing and state-altering operations. Generic operations should thus be used in their defining class and all subclasses. More specifically, the generic operations are defined in the class that introduces the corresponding attribute. At this point, the attentive reader will probably ask whether these generic operations can be protected as internal interfaces by the mechanisms provided by the programming languages. The answer is that, C++ or a similar programming language have a special language construct (`protected`). But you can only implement this by appropriate categories and related programming conventions in Smalltalk.

In the example shown in Figure 2.8, the class `Note` offers an operation, `enumerate`, with a parameter object of the class `Number`. Internally, this operation uses the protected generic operation, `setNo`, to set the encapsulated private attribute `_No` of type `integer`. This introduces a form of data abstraction to the internal relationship between classes. This approach can be used to implement abstract things. For example, generic operations can be abstracted to serve as template operations in superclasses, where their attributes are then added to subclasses.

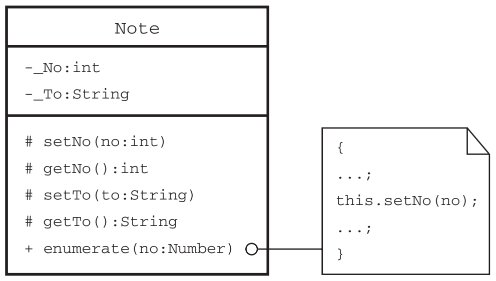


FIGURE 2.8
Generic operations at the internal interface of a class.

2.1.12 Discussion: The Object Life Cycle

Life cycle of objects

In the interplay between objects and classes, there are a few operations that limit the life cycle of an object; they are considered “critical situations” in software modeling and construction:

- Create,
- delete, and
- transform into another object.

Conceptually, none of these operations can be assigned to the object itself, because an object cannot create, delete, or transform itself. These operations belong to the metalevel (see Section 2.7).

Creating objects

In programming languages where classes themselves are available as objects in the system, *creating* is normally implemented as a class operation, that is, for each class there is at least one operation (constructor) that can create instances of that class. In compiler languages, which do not represent classes as objects, the compiler generates code to create objects, and the actual instantiation is normally handled by the runtime system. The creating procedure is then normally defined as a specialized operation within the class text. In many cases, implementing another operation that puts the object in an initial state, in addition to the creating operation, has proven to be a good idea. This initialization operation can be called when creating an object and whenever needed during the object’s lifecycle.

Deleting objects

Deleting technically specifies how an object is removed from the runtime memory. In object-oriented systems, this process comprises two steps:

- There is no more reference to the object in any location of the application system, and the object can no longer be reached in the application system.[4]
- An object is removed from the heap. This can be handled by the garbage collector, if available, when the object is no longer referenced. Otherwise, the delete operation has to be specifically written, and problems relating to lost objects, dangling pointers, and the correct deletion order of related objects have to be dealt with.

Transforming objects

Transforming means transfering an object into another one. This is normally the case when an object is processed so that it changes its “character” due to a specific action, resulting in a “different” object.

4. We will not be discussing problems with objects stored in a database, where they can be accessed as a result of database queries, at this point.

Changing class

Such a transformation of an object into another one is normally necessary for domain-specific reasons and rather difficult to reproduce in an object-oriented environment. From the technical stance, this often means that the object has to change its class and type membership, which is not supported in strongly typed object-oriented languages. Normally, if an objects needs to change class, you first create a new instance of the "target" class, then you transfer the required values from the old object to the new one, and finally, you delete the old object. Though this solution is simple, it can be problematic, because it can happen that both the software and the domain identities of the original objects are lost.

2.1.13 T&M Design: The Object Life Cycle

When we design the usage model (see Section 3.2.1) of an interactive system, we have to clarify how objects can be created. The users of our system have to clearly understand how and from where they can obtain new domain instances of their task-related objects. In practice, it has proven a good idea to use tools or domain containers (e.g., form files) to create new materials. In addition, the tools themselves can be created by use of toolboxes or tool managers.

The same rules apply to deleting objects. The question here is the *domain* view that a user has about the "deletion" of an object. We can easily see that there is a big difference between the software and domain views. To allow users to easily express that they no longer need an object, we recommend defining an explicit "location" for the domain-specific deletion of objects, for example, a tool or a domain container (e.g., a desktop bin).

The EMS example

Let's look at these things in our EMS example. Assume that a developer changes offices. The device manager updates the room plan, using the Device Organizer tool, then puts the updated room plan into a device file and closes the device Organizer. The usage model assumes that both the room plan and the tool are still available for work. However, from the software view, once the room plan and the tool settings were saved, both objects could be deleted from the runtime system. If, however, the device manager moved an old work copy of the room plan to the bin and then empties the bin, this action tells us that he no longer has any use for this plan. Still, the actual software-specific deletion of the object could be postponed until the system is shut down.

We already said that transforming an object into another one is normally due to domain reasons. Let's look at a bank example:

The Bank example

A customer consultant prepares for a meeting with a customer, filling in personal information in an application form for a small loan. Some information will be added in the course of his meeting with the customer. Next, the form is printed and signed by the customer, the customer consultant, and a second consultant. This means that the application form is now a contract for all participating parties. The customer consultant selects the "Contract signed" option in the form editor. From that point, no changes should be made to this contract object.

For the software implementation of this scenario, we could use different wrappers around an "object core" instead of changing the object's class. We will introduce a more complex technique for role modeling in Section 9.4.1. For example, the roles of a core object could be "contract" and "form."

2.1.14 Definition: Inheritance

An inheritance relationship is an essential feature that distinguishes object-oriented languages from conventional ones.

From the *domain* view, an *inheritance relationship* organizes classes in hierarchical structures. This allows us to model a hierarchy of terms using generalizations and specializations.

From the *software* view, inheritance facilitates the reuse of classes. For inheritance, *all descriptions* of a superclass are initially descriptions of its subclasses. Then a subclass can modify the properties it inherited from its superclass:

- ***Inherited operations* can be *implemented* (or *defined*) in a subclass, while they are merely specified in its superclass.**
- **Operations can be *redefined* when a new implementation in the subclass overrides (hides) the implementation existing in the superclass.**
- **New *attributes* (descriptions of the storage structure) and *operations* can be added.**
- **Access rights can be *modified*.**

We speak of *single inheritance* when the inheritance hierarchy is arranged in a tree, that is, if a class has only one direct superclass and an arbitrary number of subclasses.

In contrast, *multiple inheritance* is given when a class has more than one direct superclass.

The inheritance unit is a class, and the inheritance relationship remains unchanged during runtime, at least in statically typed languages. Note that many object-oriented languages allow you to *restrict the access* to inherited properties (particularly for operations):

- First, you can generally restrict the interface for clients, that is, an inherited property that is visible in the superclass no longer belongs to the public interface in the subclass. Examples include the private inheritance in C++, where the subclass relation is lost, or the non-export in Eiffel, where type errors can occur due to the subclass relation.
- Second, you can restrict the interface, with the exception of specific classes, that is, only specific client classes can access the properties. For example, this can be implemented by selective export in Eiffel, or by breaking the data encapsulation between classes, for example, by using the `friend` construct in C++.

2.1.15 Discussion: Inheritance

You can think of inheritance as a mechanism that allows you to reuse specifications and code. In Simula 67, inheritance is called *program concatenation*. In addition, inheritance forms the basis for polymorphism in connection with the appropriate type system.

Using inheritance

The way you use inheritance determines its meaning for the domain model and the software model. There has been vigorous discussion about single and multiple inheritance among language developers. Languages like Smalltalk or Beta support only single inheritance, while multiple inheritance is supported in Eiffel and C++. Java supports single inheritance on the class level, and the multiple inheritance concept can often be replaced by named interfaces.

Note that you should be careful when restricting access to inherited properties. This technique can be useful when building frameworks or implementing complex patterns. The downside, however, is that this often reduces the readability of your code or design. Major problems can occur with type or runtime errors, particularly with the non-inheritability of `friend` constructs in C++.

2.1.16 T&M Design: Inheritance

Single and multiple inheritance

In our T&M design, we pragmatically assume consistent use of *single inheritance* when modeling concepts and terms in the *domain model*, because it ensures that our concept model is clear. After all, a concept model built in the form of a tree is easier to understand than a mesh of complex relationships, although our everyday language has many examples of meshed terms (like the relations of bird—flying animal—fast runner—ostrich). On the other hand, we do not simply map our everyday language to our application systems but *reconstruct* a domain language instead. This reconstruction should be easy to understand in our targeted application domain. And it should be as clear and minimal as possible.

In contrast, multiple inheritance is often useful in the *software construction*, provided that it is used in a disciplined way and supported by an appropriate low-cost language implementation. Section 8.3.1 shows an example that uses multiple inheritance.

The EMS example

Let's look at these things in our EMS example. It is important for our device manager to be able to copy both the current business cards of all staff and a device identification card. For this reason, the class design for our EMS includes a `copyable` interface and the `copy()` operation. This interface implements two classes, `BusinessCard` and `DeviceCard`. However, the operation is implemented in a different way, because the data to be copied from business cards and device cards differ.

Using inheritance to build a hierarchy of terms

Concept modeling is the primary domain motivation for the use of inheritance. From the software view, we are additionally motivated by the *separation of specification and implementation* into a superclass and a number of subclasses (see Section 2.1.22).

Using inheritance for incremental modifications

Another software motivation is the use of inheritance for the *incremental transfer and modification* of properties from fixed class descriptions that cannot be changed in the source text. This is often the case in commercial class libraries. With a commercial library, if you want to change a property you often have no other way but subclassing to implement a change. In this way, you can implement changes even when the impact of such a change on client objects cannot be fully anticipated. In this case, you can encapsulate the change in a subclass, because dynamic binding lets you use both the instances of the new and the old class (polymorphic).

Be careful when reusing existing code and extracting attributes

Finally, the last of the uses of inheritance we consider acceptable and useful serves to *abstract common properties* from existing classes. You should always carefully make sure that these common properties are motivated from the domain view, rather than reusing existing code and extracting attributes for the sole reason that you have a piece of code available. In fact, by our definition, inheritance is not about the reuse of existing code but about the abstraction of a common behavior. This view also justifies our critique of "extracting" common attributes into superclasses. These superclasses are oriented to the internal structure, having not much to do with behavior-specific abstractions. Structure-oriented inheritance hierarchies are often found in projects based on "traditional" data modeling.

In summary, our practical experience has shown that "artificial" superclasses oriented to code reuse and the internal structure tend to become hard to understand, thereby hindering further development of such a system.

2.1.17 Discussion: Role Relationships as an Alternative to Inheritance

Roles as additional relationships between classes

As we have said, we use inheritance primarily for concept modeling. We interpret superclasses and subclasses in "Is-a" or "Seen-as" relationships. In this sense, a folder is a special form of a container. However, when developing large-scale application systems, we often incur a problem that cannot be elegantly solved in this way. In an organization with several departments or groups, each having a different view of the same work object in different contexts, you obviously have to deal with many different views that could be better modeled by different roles rather than by generalizing and specializing a term.

A generalization relationship connects objects solely through their common abstraction. A folder is a container, and a file is a container; that's the only thing these two objects have in common. A folder *is not* a file only because both objects are containers. Things are different with changing roles. For example, a customer of our fictitious bank can be both a debtor and a portfolio owner, depending on the situation. We say that the customer plays different roles. Depending on the role, the customer has a different meaning for the bank. We could say that the customer has a different behavior. Nevertheless, when mapping that customer and his or her roles onto the application system, we have to ensure the customer's domain identity in each of his or her roles. We have developed our own role pattern (see Section 9.4.1), because inheritance or use relationships won't let us build these roles. This means that we add a role relationship to our application development process, in addition to specialization and generalization.

2.1.18 Definition: Use Relationships

A use relationship corresponds to the traditional call relationship or module import. In fact, it is the classic form of connecting two program components.

A *use relationship* realizes the relation between clients and service providers from the *domain* perspective. This relationship can be regulated by a contract (see Section 2.3).

In the *software* model, a use relationship interconnects both objects and classes. On the class level, the use relationship is expressed by a (static) type declaration. At runtime, operations of different objects of the same type or subtype can be called using one identifier, based on polymorphism.

The EMS example

Let's see what this looks like in our EMS example. Our device manager will occasionally want to copy the business card of an employee or a device identification card, for example, to use it as a template. He uses a photocopier for this purpose. To copy a business or identification card, we need a `copy()` operation. The photocopier is not interested in any other operation of the `BusinessCard` and `DeviceCard` classes. For this reason, we declared an identifier of the type `copyable` in the `Copier` class. At runtime, objects from both the type-confoming classes `BusinessCard` and `DeviceCard` are bound to this identifier.

2.1.19 Discussion: Use Relationships

Modularization and the use relationship

A lot of practical experience and knowledge about the meaningful use of this form of component relationship has been collected in the modularization camp. This appeared to have been forgotten in the initial euphoric phase in the advent of object-oriented programming. For example, many modern method and construction textbooks still discuss the pros and cons of inheritance relationships, totally ignoring the use relationship. In our approach, the use relationship between classes ranks high, and its methodological use will be discussed in Section 2.3.

Reference semantics

In the object-oriented model, the use relationship is generally based on pointers or references, and a static type is declared for an identifier. At runtime, this identifier can be bound to pointers or references to instances of classes of a conforming type. The idea behind this concept is reference semantics, which is covered in the classic Smalltalk literature, for example, by Adele Goldberg and David Robson. However, you will note quickly that this concept cannot be maintained for (arithmetic) calculations, where we need values in the mathematical sense. Section 2.7 discusses what all of this means for our object metamodel.

2.1.20 Definition: Polymorphism

On a class level, the use relationship means that the class text includes a static definition of the relationships that are basically allowed between the objects of that class and other classes. The actual relationship between objects is then determined by the principle of polymorphism at runtime.

The general meaning of *polymorphism* is the quality or state of being able to assume different forms. In object-oriented programs, polymorphism is the ability to bind objects of different types to an identifier at runtime.

If the types of objects have a subtype relationship to the statically declared identifier type, then we speak of *constrained* or controlled polymorphism, while *unrestrained* polymorphism means that arbitrary objects can be bound to an identifier.

Most statically typed languages use the inheritance relationship to control polymorphic binding. This means that objects of the declared class and all subclasses can be bound to an identifier. This polymorphism is typesafe, that is, all operations of the declared class can also be called in its subclasses as long as the relationship between the superclass and its subclasses is a type-subtype relationship.

Unrestrained polymorphism

Dynamically typed languages like Smalltalk allow unrestrained polymorphism. This means that whether or not an operation is supported by an object can be decided not at compile time but only when that operation is called. If that object doesn't support the called operation, then the system reports a runtime error. This means that programming conventions should be used to control unrestrained polymorphism. The reason is mainly that, unless you can tell from the identifier names, the class text tells you nothing about the objects that are actually bound at runtime.

Dynamic binding

Dynamic binding is the prerequisite for polymorphism in connection with redefining operations in subclasses. *Dynamic binding* means that the runtime environment decides what implemented operation will be executed when you call it.

When you can bind different objects to one identifier at runtime, then these could have different implementations by the same operation names. It means that the static program text defines only the messages sent to an object.

The EMS example

In our EMS example, we defined an aspect class, `copyable`, for the photocopier. At runtime, instances of the `BusinessCard` and `DeviceCard` classes can be bound to identifiers of this type. The message `copy()` is bound to the operation implemented in either the `BusinessCard` or the `DeviceCard` class, depending on the specific case.

2.1.21 Definition: Abstract Classes

In object-oriented languages we often use their ability to separate classes by specification and implementation, for example:

- A superclass contains the specification, that is, it specifies behavior on an abstract level.
- The subclasses of this superclass contain implementations, that is, the behavior specified in the superclass is implemented on a concrete level.

From the software view, we implement specifications either by abstract classes or by named interfaces, as in Java.

- An *abstract class* is a class from which no instances can be created. They ususally contain at least one nonimplemented (i.e., abstract) operation.
- Abstract classes are used to specify a common interface and an abstract behavior for all subclasses that implement the concrete behavior.
- The abstract class itself can implement some operations, which will then be inherited.

We say that a class is abstract (or *deferred*) when at least one of its operations is abstract. Some object-oriented languages (e.g., Java and Eiffel) let you mark a class as an abstract class, even if it contains no abstract operations. In other languages, abstract classes have to be denoted as such by convention, or implicitly by a deferred method.

Abstract classes are often more than pure interface specifications. The operations declared in an abstract class can be used right there in the abstract class. Using these abstract operations you can implement several operations. They can be called to serve as template operations or standard implementations by client classes. This technique is important for building frameworks.

Ralph Johnson and Vince Russo distinguish three types of operations in abstract classes:

- ***Abstract operations* are not implemented, that is, their implementation is left to subclasses. But abstract operations specify an interface mandatory for all subclasses. An operation specifying only a default implementation but intended for redefinition in subclasses is often called a *hook*.**
- ***Template methods* implement an algorithm based on abstract operations. This algorithm is fully specified, but it lacks the implementations of the abstract operations to be actually executable.**

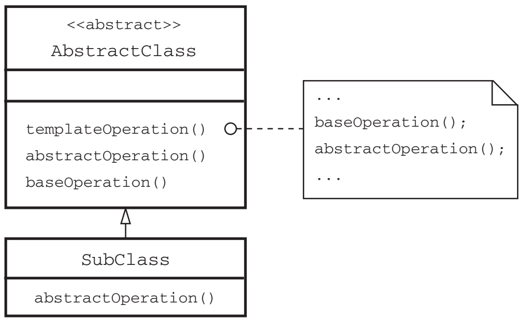


FIGURE 2.9
Specializing an abstract class.

- ***Base operations* are the operations already implemented in the abstract class in full and executable.**

Example

Figure 2.9 shows an abstract class whose interface includes the following operations: `templateOperation`, `abstractOperation`, and `baseOperation`. Both `baseOperation` and `abstractOperation` were used to implement `templateOperation`, while `abstractOperation` itself will be implemented later in the inheriting class.

The subclasses of abstract superclasses are normally used on the basis of polymorphism. However, we often have to expand or specialize an interface in subclasses for domain or software reasons. If we want to use the instances of subclasses both polymorphic and by their dynamic type, we will soon find that we need a metaobject protocol (see Section 2.1.15).

2.1.22 Discussion: Specification and Implementation

Separating specification from implementation

An important way to use inheritance is to separate the specification from the implementation. A common superclass specifies the interface and the abstract behavior of all subclasses. The fact that no objects can be created from an abstract class tells us that we cannot use it directly. The subclasses are responsible for implementing the concrete behavior. On the other hand, it is normally sufficient for a client to know the abstract class with its abstract behavior. For example, you can hide an entire class tree "behind" the abstract superclass.

With their interface and behavior specifications, abstract classes define how their subclasses can be handled, thus defining in certain limits the entire system design.

Patterns with abstract classes

Abstract classes with a hidden class tree form the basic idea for a fundamental design pattern, the so-called Family pattern of Erich Gamma,[5] and expanded constructions like the Bridge pattern of Gamma et al. work similarly. The T&M design lets you implement aspects in this way (see Section 8.3.1) by use of aspect classes.

2.1.23 Definition: Loose Coupling

The use of abstract classes is one example of how you can develop systems with loosely coupled components.

5. E. Gamma: *Objektorientierte Software-Entwicklung am Beispiel von ET++*. Berlin, Heidelberg: Springer-Verlag, 1992.

A *component* is *loosely coupled* to another component when the client does not know the entire interface of the provider but only a section of that interface (and its behavior). This part of the entire interface should be defined by a type and contain only operations actually needed by the client for this coupling.

As a result of loose coupling, you can replace both individual classes in the program code and objects at runtime by appropriate components with the same interface and similar behavior but a different implementation.

2.1.24 Discussion: Loose Coupling

Loose coupling was motivated by the discussion of modularization concepts (see Section 2.2). These concepts say that components should have little dependencies to the outside while having a strong contextual coherence inside. One technique to achieve loosely coupled components addresses only the type or interface of other components when using these components, without knowing the concrete classes and their implementation (Gamma et al.: "Program to an interface, not an implementation" p. 18). We take this concept a step further, reducing loose coupling to the section of the interface, which is the minimum required for use. Section 8.3 will discuss aspects for implementing loosely coupled tools and materials.

Loose coupling and object creation

When using loose coupling between a client class and the abstract superclass it uses, we have to solve a construction problem: The concrete subclass has to be known at the location where we actually create an object. Initially, this seems to conflict with our decoupling idea. A number of mechanisms have been proposed to be able to create concrete objects without losing the decoupling concept. Gamma et al. describe these mechanisms in the form of creational patterns. Section 9.4.2 will introduce a creational pattern, the Product Trader, which is particularly useful for our T&M approach.

2.2 MODULARIZATION

2.2.1 Introduction

Classes are traditional modularization units for object-oriented design. More recently, larger design and construction units, like design patterns or packages, have been proposed as modularization units. This section discusses the basic principles of modularization in view of using them in our object-oriented design.

2.2.2 Context: Modules and Object Orientation

In the software model, you can think of classes as a further development of the modular concept. As Bertrand Meyer says, "Classes provide the basic form of module". This means that basically the same principles of maximum cohesion and minimum coupling apply to both modularization and the use relationship.

Modules and classes

Unfortunately, these traditional modularization principles cannot simply be transposed to object orientation. First, classes or objects have a different granularity than common modules. Although it is customary to build an application system from far less than a hundred modules, the number of classes or objects can easily reach ten times that number in a moderately complex application. Each class includes a manageable number of operations.

If we think of objects and classes as the "atoms" of our object-oriented design, we obtain a totally different view. Obviously, we design and build our application in units or components, formed of more than one class or one object. Such a unit could be a container with a table of contents and markers or iterators (see Section 3.2.7). To implement such a container, we need mutual-use relationships. Note that some successful patterns, such as those described by Gamma et al., for example, Visitor, Observer, or Mediator, are also based on mutual use.

Object-oriented design and construction units

Recent discussion has led to the understanding that the conceptual design and construction units used in object orientation are often beyond a single class, and it has also led to the development of design patterns, clusters, subsystems, and frameworks. This means that the modularization principles in our object metamodel have to be reformulated in view of these design and construction units. We will attempt to reformulate them for the basic design units in Section 2.2.3 and for frameworks in Section 9.3.

2.2.3 Definition: Principles of Object-Oriented Modularization

Maximum cohesion and minimum coupling are the fundamental factors for traditional modular design.

Cohesion **is the "inner connection" between the properties of a design or construction unit. The principle of maximum cohesion requires a strong bond within a design or construction unit.**

Coupling **is the relationship between different design or construction units. Minimum coupling attempts to reduce the bond between units, particularly to avoid cyclic use.**

Modularization criteria

Bertrand Meyer proposed a list of criteria and rules to facilitate the transfer of the modular concept to object orientation. We will briefly introduce some of these criteria and rules below.

One of the most important features of object orientation in this context is the open-close principle.

Open-Close Principle**: A design or construction unit is *open* if it can be extended. In contrast, a design or construction unit is *closed* if it can be used by clients in a stable way.**

A design and construction unit should meet the following important criteria:

- *Decomposability*: A design problem should be decomposed in smaller and less complex partial problems, which are mapped to design and construction units. These units should form a simple and independent structure.
- *Composability*: Design and construction units should be composed by simply recombining them into new software systems in different application domains.
- *Understandability*: Each design and construction unit of a software system should be understood independently of the other units.
- *Continuity*: A change to the design problem resulting from the domain or software system context should lead to changes only in one or a few design and construction units.

Modularization rules

To meet these criteria, we have to observe the following rules:

- *Direct mapping*: The structure of the software system should closely relate to the structures identified in the application domain.
- *Few interfaces*: Each design and construction unit should interact with as small a number of other units as possible.
- *Small interfaces*: When two design and construction units interact, then they should exchange as small an amount of information as possible.
- *Explicit interfaces*: When two design and construction units interact, then this exchange should be explicit.
- *Information hiding*: Clients should see and access only relevant properties of a design and construction unit.
- *Open-closed principle*: Design and construction units should be both open and closed.

2.3 THE CONTRACT MODEL

2.3.1 Introduction

We want to use an object-oriented approach to develop not only expandable, reusable, and modifiable but also correct and robust classes. We try to achieve robustness by modeling behavioral objects, that is, modeling objects characterized by their behavior and not by their inner structure—their attributes. We use the contract model to sketch a concept allowing us to specify the behavior of operations in a "semiformal" way. This contract model represents an important contribution to better software quality and should form the basis of each development project. It provides what Bertrand Meyer calls "contracting for software reliability."

2.3.2 Definition: Contract Model

We can define the essence of the contract model as follows:

A *contract model* specifies the use relationship between classes as a relationship between service providers and clients, based on a formal contract.

A contract specifies the pre-condition that a client must meet before the service provider supplies its service. The service supplied is specified in a post-condition.

Contracts are described on the provider side; they have the form of assertions, that is, they are pre- and post-conditions as well as invariants.

The use relationship between classes is interpreted as a service relationship between a client and a provider (see Figure 2.10). In this context, a class offers a service that another class, acting as the client, wants to use. These two classes enter a contract relationship, where the contract specifies all underlying assertions as well as the rights and obligations of both parties relating to that service.

The contract model is implemented by pre- and post-conditions for operations as well as for class invariants, having the form of boolean expressions and residing in the provider class.

Contracts are operation calls

Contracts established between classes always refer to operation calls, that is, requests for services. The pre-condition specified for a contract defines the rules that

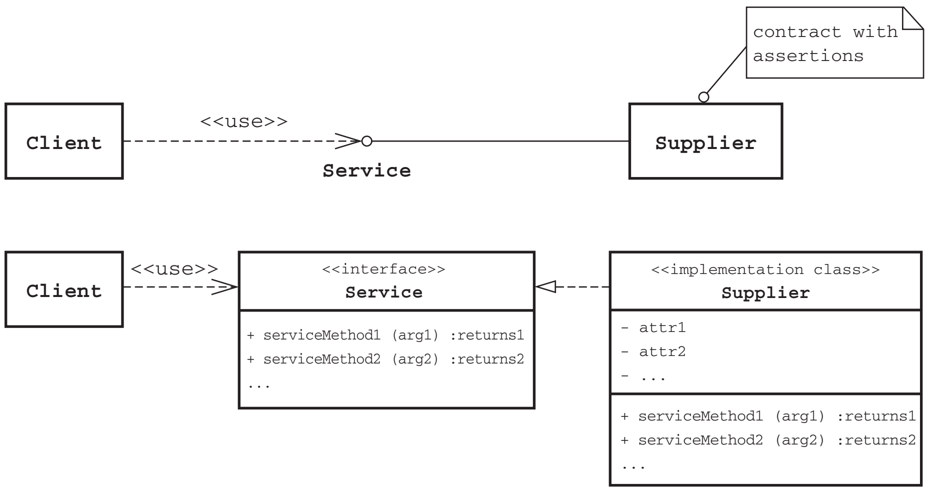


FIGURE 2.10 Service provider and client in the contract model.

have to be observed to be able to call an operation. The client is responsible for observing this pre-condition. More specifically, the client has to ensure that the specified conditions are met. In order for the client to meet this obligation, the predicates used as test questions in the pre-condition have to be accessible at the provider's interface. Before calling an operation, the client can then test for observation of the pre-condition and establish the required state, if that pre-condition is not yet met.

Once the pre-condition is in place, the requested operation is executed, and the provider declares that they guarantee the post-condition. This condition is also composed of boolean expressions and does not have to be tested by the client. The client can assume that the condition holds when the operation has terminated.

Invariants

An invariant is formulated for a class as an overall property of that class, which preserves its consistency. This holds true for each service provided, that is, its observance is tested upon each call.

2.3.3 Discussion: The Contract Model

By stating assertions within the interplay between clients and service providers, we can make a significant step towards the correctness and reliability of our software system. In this context, correctness refers to the conformity between specification and implementation. We want to use invariants and pre- and post-conditions to describe parts of a behavior specification contained in the class text and make them checkable. This means that the contract model helps us to

- reduce design and construction errors;
- improve the understanding of our classes; and
- use an appropriate exception mechanism to catch runtime errors early.

2.3.4 Context: The Contract Model and Abstract Data Types

Using classes as abstract data types

From the software view, classes should be implementations of abstract data types. Using known language features available in most object-oriented programming languages we can only define operations and attributes. However, if we want to use types to model "a set of programming language objects with similar behavior," we need an adequate means of expression. The reason is that the specification of abstract data types involves axiomatic semantics, which is normally expressed by axioms and equations. Object-oriented programming languages do not normally support such axiomatic semantics. We can use the contract model, which Bertrand Meyer describes as “design by contract,” to overcome the gap between specification and implementation of abstract data types.

2.3.5 T&M Design: The Contract Model

The contract model is currently supported only by Eiffel among the object-oriented languages, where assertions can be formulated as an independent sublanguage (see Figure 2.11). However, it has proven useful in practical projects to try to transport the assertion concept to other programming languages. For example, our JWAM framework implements the assertion concept in Java.

Contract violations

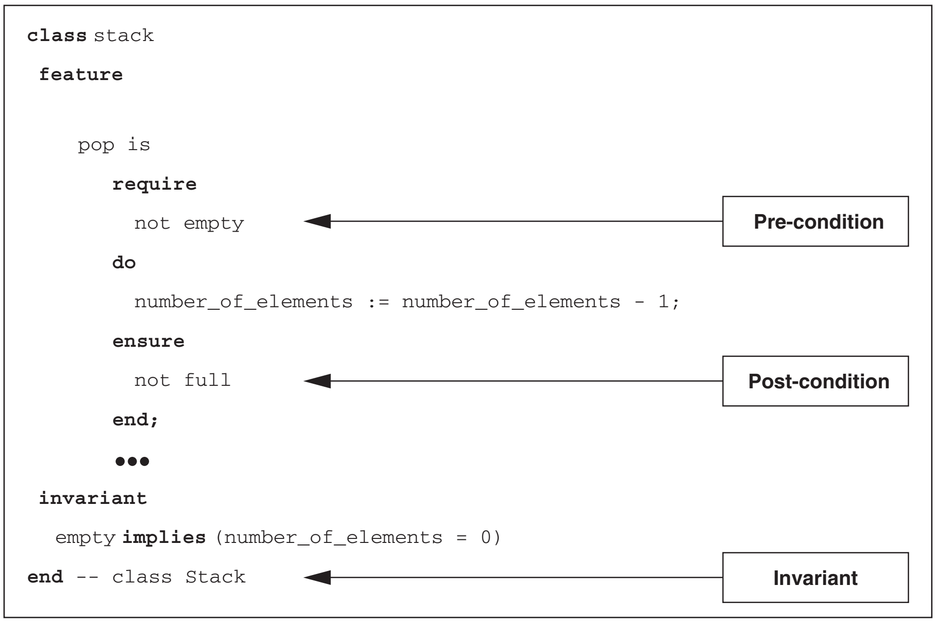
As in the case of a commercial contract, we have to check for the observance of a contract, and the violation of the contract has to have corresponding consequences. The contract model is more than a pure piece of documentation only if contract violations are sanctioned. For this reason, we should link assertions to an appropriate exception mechanism. An exception will then be thrown at the client’s end as soon as a pre-condition is violated, while a violation of post-conditions and invariants will lead to exceptions at the provider’s end.

2.4 TYPES

2.4.1 Introduction

Current object-oriented programming languages use classes and types interchangeably. Though this is practical, it blurs the conceptual differences we want to observe to

FIGURE 2.11 Example showing assertions in Eiffel.



```
class stack
 feature

   pop is
      require
        not empty
      do
        number_of_elements := number_of_elements - 1;
      ensure
        not full
      end;
      •••
 invariant
   empty implies (number_of_elements = 0)
end -- class Stack
```

ensure a clean software system. This holds true particularly when we use a dynamically typed programming language like Smalltalk, or when we have additional type definition options for interfaces, as in Java.

2.4.2 Definition: Types

A *type* traditionally denotes a set of values and the admissible operations on these values. From the software view, a type is used to declare identifiers and parameters for a program to test its typesafe use.

In the object-oriented world, a type specifies the behavior of objects in the sense of an abstract data type.

A type specifies the syntax of an interface, that is, it names the operations you can use to call an instance of that type. In addition, a type can describe the behavior of objects in the sense of a protocol.

In contrast to a class, a type does not include information as to how the state and implementation of operations should be represented.

With this definition of *type*, we have determined at least the checkable interface used by instances of that type. This means that an identifier we declared to have that type can be bound only to objects that meet this interface. In other words: a type should be checked. For this reason, we like typed languages, because declared identifiers and parameters of our program can then be checked for typesafe use either statically (at compile time) or dynamically (at runtime).

Type and behavior

In addition, we want to specify potential limits for the behavior of these instances. As we saw in Section 2.3 on the contract model, it is not easy to transfer the axiomatic semantics of abstract data types to an object-oriented construction. Nevertheless, we expect that instances of a specific type behave in a "semantically compliant" manner. Accordingly, assertions should also be included in the definition of a type.

A type in the sense of an abstract data type does not tell us anything about the concrete implementation of the interface and its behavior. This means that implementations and information about the internal structure are not part of the definition of a type.

2.4.3 Context: The Theoretical Concept of Types

The concept of types is primarily motivated by software-specific factors. We use types to specify our construction units. Following the general definition developed by Luca Cardelli and Peter Wegner, a type is a set of objects with similar behavior. In their formulation, Cardelli and Wegner didn't focus on the object-oriented elements, that is, classes and objects, but generally refer to programming language objects. When we try to transfer this to object orientation, we could say that, for example, the two rooms, `a101` and `a103`, in Figure 2.12 have the same interface and a similar behavior, so that we can use the type `Room` to describe them.

Traditionally (e.g., by the definition of Hoare), an object of a programming language may have one and only one type. The object-oriented inheritance principle requires us to expand this concept of types. By this principle, an object is exactly the instance of a (creating) class, but it can have more than one type.

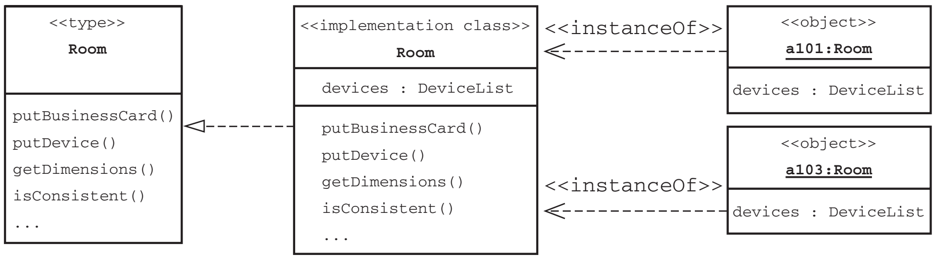


FIGURE 2.12
From objects to the type.

A type is characterized by the set of its *properties*. For example, an object `x` is of type `T` exactly when object `x` meets the properties characterized by type `T`. The following properties can be described by a type:

- the class of an object;
- the names of operations or the entire signatures of an object; and
- the behavior of operations (e.g., pre- and post-conditions and invariants).

Real-world type systems normally offer only a fraction of these properties.

Types can be organized in type hierarchies by forming subtypes.

A *type hierarchy* according to Barbara Liskov consists of subtypes and supertypes. A *subtype* supports at least the operations specified in its supertype, with the following variants:

- **the operations have identical signatures; and**
- **the types of parameter and result objects can themselves be of a subtype or supertype of the original type (covariance and contravariance).**

At runtime, an instance of a subtype must be able always to assume the place of an instance of the supertype, where either identical or similar behavior is required, that is:

- **an instance of a subtype does not cause any observable change to the program; and**
- **an instance of the subtype behaves similarly in that it produces the expected behavior and never a runtime error.**

Conceptual methods used to specify the behavior of subtypes are discussed in the seminal work of Barbara Liskov and Jeannette Wing. The problem relating to the covariant redefinition of parameter types and a proposed solution are discussed by Bertrand Meyer.

2.4.4 Discussion: Types

Building subtypes is primarily a specification concept that we can use to express that objects behave similarly. This concept reaches beyond the usual check for identical interface names. This means that it is not sufficient to simply be able to interpret the called operation by the object. The reason for this is that it only means that the call does not immediately cause a runtime error. What we normally need is a way to interpret all operations specified by a type interpret, which leads us to the concept of similar behavior. Similar behavior is a much "softer" formulation than the identical behavior recommended by many computer scientists. Unfortunately, it is difficult to check it formally. In contrast, when we implement an object or redefine its properties, we expect it to behave both "in the sense of" the specification and formally compliant with its type.

Types and behavior specifications become increasingly important in the use of components. When components are shipped merely as binary units, then all that remains to evaluate their functionality is basically the interfaces and thus the type information. In that case, it is important to know what other characteristics are assigned to types, in addition to the signatures, for example, assertions of a contract model.

2.5 CLASSES AND TYPES

2.5.1 Introduction

In the advent of object-oriented programming, many developers used classes and types synonymously. The reason for this was that the construction of classes was the only way to introduce user-defined types. Over the course of time, object-oriented programming languages have been improved and expanded so that the typing concept is now more pronounced. For this reason, developers should understand the conceptual differences between classes and types.

2.5.2 Classes in Your Design

From the domain-specific perspective, classes are our elementary design and construction units. In the design, we model the common features of similar objects in classes. These common features apply to objects in their behavior and state. In this context, class hierarchies should correspond to the domain-specific concept model. This means that they model the basic concepts or abstractions of your application domain.

On the construction level, a class defines how its instances behave and how they are built. In this context, class hierarchies express the generalization or specialization of this behavior and construction. The compiler and the runtime system ensure that the correct instances are created based on a set of "complete" class descriptions.

2.5.3 Differences between Types and Classes

Compared to a class, a type is primarily a specification concept that defines an "external" view of declared identifiers and program objects. Note that the emphasis is more on the software implementation rather than on the domain modeling and construction side. A type defines the syntactic and (to a limited extent) the semantic characteristics that can be guaranteed for the structure of the program.

Let's take a closer look at these differences.

How classes and types differ

- Besides the interface, a class also defines the behavior and structure of its instances, that is, their internal state and the implementation of their operations.
- The type of an object refers intially only to its interface, that is, a set of messages to which the object can respond. The type of an identifier specifies what objects can be basically bound to it.

- To be able to define the behavior of the instances of a type, in addition to the interface, we can implement types by classes. As long as the type hierarchy and the class hierarchy match, you can use assertions and controlled redefinitions to achieve a similar behavior of objects.
- Some languages (e.g., Java) can use named interfaces instead of classes to introduce user-defined types. Most object-oriented programming languages provide primitive data types (e.g., integer) that are not defined as classes.
- An object can have different types, and objects from different classes can have the same type.
- One part of the interface of an object can be characterized by a type, and other parts by other types.
- For two objects to be of the same type, it is sufficient that only part of their interfaces are equal.

2.5.4 Discussion: Classes versus Types

Classes and types can be totally separate concepts if the type refers strictly to the interface definition. If a type represents only a named interface definition, then it merely guarantees that instances of that type recognize a certain set of messages. Each type names operations that have to be defined in some place of your program. A class can then declare the types it meets in the sense of named interfaces. The actual class that defines these operations is irrelevant and independent of the class hierarchy.

Of course, if the programming language we use couples the class and type concepts, then there is a relation between class and type. In that case, the class defines a type. When we say that "an object is an instance of a class," we imply that the object supports the interface defined by the class. A type-compliant call of a message requires the object to understand the message, in other words, that the object's class implements the called operation.

We can go one step further in coupling classes and types when we use the type definition to specify the behavior of instances. In this case, it appears to be meaningful to use an appropriate class to define the behavior of the instances of a type. If the type hierarchy and the class hierarchy match, then we can define and limit the concrete or abstract behavior of objects of a type by the class hierarchy. Appropriate means for this approach would be the contract model and the mapping of our concept model onto domain-specific class hierarchies, which are designed by the principles of generalization and specialization.

2.5.5 Background: Programming Languages and Types

Popular object-oriented languages use different type and class concepts, for example:

Java, Objective C: named interfaces

- Java separates classes and types by introducing *interfaces*, as does Objective C with its *protocols*. In these languages, a class can declare, besides its superclass, which named interfaces it meets. Classes can be grouped by class hierarchies and by their interfaces. This means that classes can fall under a common named interface, even when there is no inheritance relationship between them.

- C++ and Eiffel use the class construct to define both the type of an object and its implementation. In addition, these languages let you write pure specification classes, from which no instances can be created. Similar to C, C++ also supports types that are not classes.

C++, Eiffel: classes act as types

- In Smalltalk, there are no type declarations for identifiers and program units, that is, there is no type checking during compilation. Consequently, whether or not an object is called by the appropriate message through an identifier is not checked, so that a runtime error occurs if the object does not offer the called operation. The Smalltalk world works with message conformity rather than with type conformity, in other words, not a specific class or type relationship is in the foreground, but the question of whether or not an object understands a message.

Smalltalk: no type declarations

2.5.6 T&M Design: Classes and Types

In our T&M design, we need a clear understanding of the possibilities offered by classes and types as constructs, and which concepts we want to use in our applications. Do we want to separate a named interface from the definition of its behavior? Should there be a hierarchy of named interfaces, in addition to the class hierarchy, and what does this mean in our design? What kind of reliability would we like to have when calling messages, and when (at compile time or runtime) do we want to check this reliability? The answers to these questions determine whether or not we can easily deal with the constructs of a language, how we use that language, and whether or not we should add our own mechanisms, such as runtime type checks in Smalltalk.

Some answers originate from practical experience with the T&M design. For example, type systems represent a major support for the *save* construction of software systems. Dynamically typed languages like Smalltalk offer more flexibility during the actual construction phase, compared to statically typed languages. However, it is often meaningful to design in Smalltalk as if that language were typed. To be on the safe side, we will add dynamic type checks at important points in our software system.

Inheritance hierarchies should correspond to domain hierarchies, thus defining a similar behavior. This means that we normally work towards class hierarchies in the form of type hierarchies.

In cases where we cannot or do not want to fall back on multiple inheritance to implement software architectures technically, named interfaces are a meaningful alternative. We then have to stick to design conventions to ensure similar use behavior.

2.6 VALUES AND OBJECTS

2.6.1 Introduction

Values and objects are two fundamental concepts for the development of interactive software. This section explains that the differentiation of values and objects is not a terminological trick, but important for the design and construction of software systems. We use the term *domain values* to introduce an important concept of the T&M approach. Domain values can be used as the “atomic” design and construction units for application-oriented software development.

2.6.2 Characteristics of Values and Objects

To better understand the following discussion of the terms *value* and *object*, it appears meaningful to understand first where these two terms differ. The taxonomy in Figure 2.13 is schematic and in no particular order, but shows the essential characteristics of values and objects. It is based on the seminal article by Bruce J. MacLennan. We will explain the differences listed in Figure 2.13 in a moment.

- A *value* has no *temporal* or *spatial dimensions*. This means that concepts like time, duration, and location are not applicable. Values have no beginning and no end, and they exist in no particular place. No values are created in expressions, and they are not consumed. For example, it wouldn't make sense to talk about time in the equation $40 + 2 = 42$, or about the fact that the addition "creates" a new value. And it would be equally useless to talk about "the 42 on top of this page" when referring to the value and not to its concrete representation in digits.
- In contrast, *objects* exist in *time and space*, and they have a beginning and an end. Any two objects can differ from one another merely because they reside in different locations. For example, you can create a folder. Two otherwise identical folders can be in different locations. It would make sense to talk about "the folder I used yesterday."

Another difference between values and objects is that a *value* is *abstract* and has no identity.

- A value is an abstraction from all concrete contexts. It is not bound to the existence of concrete things. And because it has no identity—only equality—it wouldn't make sense to talk about several instances of a value (although it would make sense to talk about its different representations). For example, $50 is a value. Though there are many bank notes representing this value, there is not more than one instance of the value itself. Also, it would only make sense to talk about whether or not a bank note is worth $50.
- In contrast, *objects* are *concrete instances* of a generic concept (a class) that have an identity. There can be many equivalent but different instances of a class. For example, you can create several instances of an application form on your electronic desktop that differ in that they reside in different folders.

FIGURE 2.13

Characteristics of values and objects.

A value is:	An object:
• timeless	• exists in time and space
• abstract	• has identity
• immutable	• is concrete
• stateless	• can be instantiated
• unique	• is changing
• spaceless	• is stateful
	• is created and destroyed
	• can be used cooperatively

Yet another difference is that a *value* is immutable or *invariable*.

- Though you can calculate and relate values to other values, they won't change. Values can be named. The value of a name or an identifier (of an "unknown quantity") cannot yet be calculated or can be undefined. The same name can be bound to another value, depending on the context. For example, when looking at the equation `40 + x = 42;  pi = 3.14`, the idea that adding `x` would change the value of the number `40` to `42` is wrong. Also, `x` is not a variable, but has a calculable value. We can call the number `3.14` by the name `pi` and use the same name to bind it to the value `3.1415` in another context.
- In contrast, an *object* can *change over time*, that is, its state can change without losing its identity. And this identity does not have to be linked to a specific name. For example, you can edit an application form that you created yesterday and sign it tomorrow. You could change the form's name from "New application" to "Edited application" without risking that it might lose its identity.

Yet another difference relates to how you can *use* values and objects.

- Considering that a value has no identify and no location, and that it is invariable, it would be hard to build communication and cooperation on the basis of values alone, because values cannot be exchanged or edited. In situations where you have to communicate and cooperate on the basis of values, you often use a specific value to "build" an identity. For example, an abstract value of $500 won't be of much use in the banking business. It will become useful only if it is connected with an account, and if this account is seen in its temporal change. Only then can you use that value for cooperative work. For instance, to build an account in a value-based database, you can use the account number for unique identification of all other values related to this account.
- In contrast, *objects* can be used *jointly*, if you can access them by references. Then they can be known by different names in different locations, and you can use them as a common work object. Unfortunately, there is an alias problem to be solved: An object can be changed in one context without another context taking notice of that change. For example, a form can be accessed by identifiers on two different electronic desktops. This means that two employees can use it for cooperative work and coordinate their work through this form. Problems will arise when one employee does not see that the other employee changed the form without prior agreement to do so.

2.6.3 Using Values

Using values

When we think of using values, we probably first think of mathematics or engineering, where numerical analysis is of prime interest. For example, mathematical problems are solved by operations on numbers, a form of values. However, in our everyday lives, values and numbers have a much broader meaning. We use them to identify, characterize, count, or order things and eventually to represent them as measurable entities.

We basically always use values when we model abstract entities and do not want to be distracted by concrete and objective characteristics. In doing this, we also abstract from the context of the thing represented by a value. In this sense, the authors of a popular textbook, Richard Bird and Philip Wadler, discussed functional programming as follows: “Somewhere, in outer space perhaps, one can imagine a universe of abstract values, but on earth they can only be recognized and manipulated by their representations. There are many representations for one and the same value” (p. 5).

What does this mean? It probably means that we always need values and numbers when we have to calculate or order things. There is no doubt that we then need integers or real numbers, as well as currency values or periods of time. Even when we try to represent measurable parameters, ignoring the concrete circumstances of an object, values like bank codes or the current Dow Jones index are certainly useful values.

2.6.4 Context: Values and Objects in Programming Languages

Many object-oriented languages support both value and object concepts. The difference between these concepts is normally implicit and hardly discussed explicitly. The conceptual separation is least clear in a language like Smalltalk. When talking about values, it is important to understand that a truly smooth transfer of the abstract concept of values to programming languages was successful only in purely functional languages, such as Miranda or Hope. These languages let us write value-oriented program code; for example:

- They use expressions that are almost algebraic, consisting of functions. *Functions* have no side effects, so that they are similar to the concept of algebraic expressions.
- They use *mathematical variables*. A *variable* is a name for a known or yet unknown value (as in “an equation in two unknowns”), but this value cannot change.

The idea behind functional languages is referential transparency.

- *Referential transparency* means that an expression can be fully understood on the basis of its partial expressions.
- Each *partial expression* (subexpression) is independent of its context.
- A *variable* is a name for a (known or unknown) immutable value.

Object-oriented programming languages suggest a totally different programming model.

- They use objects that encapsulate an internal state, which can only be changed through permissible operations.
- They use an *imperative variable concept*. A variable is an identifier for a memory location or container, the state (or value) of which can be changed through assignment.
- Each expression depends on its context, that is, on the state of all participating objects. An object can be known by different names in different contexts, so that changing the parts of an expression will normally lead to side-effects.

Values in the object metamodel

A closer look at values in object-oriented programming languages shows that many languages use values as so-called primitive types, such as integers, floating point numbers, or booleans. These value types behave as we would expect from values or numbers (apart from precision errors caused by representing them on a finite computer).

Some “puristic” languages—like Java, Eiffel, or Smalltalk—embed these value types in the normal class concept. However, such embedding can be problematic, because certain properties of objects cannot simply be transferred onto "value objects." For example, arithmetic expressions would lose their mathematical semantics if an operation would change object `3` to an internal value `4`. This is why we find it confusing when textbooks, such as the Smalltalk handbook of Adele Goldberg and David Robson, use number arithmetics as one of their primary examples for objects and their operations. We want to have numbers that behave like values, and objects that have all object properties.

2.6.5 Definition: Domain Values

The primitive built-in value types of an object-oriented programming language are easy to use. A problem can normally occur when you try to introduce user-defined value types. We refer to these values as domain values, because they are motivated by the application domain of a system.

A *domain value* is a user-defined value. It represents values from the application domain.

A domain value type is a data type with a defined set of values and defined operations. Its internal representation of values is hidden.

Object-oriented languages define domain value types as classes. It is important to understand that the instances of a domain value type always have value semantics, that is, once you set a value, you can't change it.

The motivation for domain values is obvious. Language developers have tried to supply such data types, specific to an application domain, since the development of early programming languages. The only difference is that it was not as obvious as it is today, because most early applications were oriented to number-crunching tasks.

2.6.6 T&M Design: Domain Values

Domain values are of prime importance in the T&M approach. When we say we want to use classes and objects to model the concepts and things of our application domain, then this also applies to the values that play a role in that application domain.

The Bank example

Let's look at the idea of domain values versus conventional programming in our bank example. Assume that we have accounts with account numbers. In an object-oriented application system, a specific account is an instance of the class `Account`. The account number in this object should be a domain value of the type `AccountNumber`. The appropriate attribute would have an identifier, `AccountNumber`.

In conventional programming, we would also have an attribute called `AccountNumber`, but declare it as an attribute of the type `integer`. The fact that it represents an account number can be seen only in the identifier `AccountNumber`.

2.6.7 Implementing Domain Values

In object-oriented languages, we have to use the class construct to build user-defined domain values. This is because classes are the only way to add new types and instances of these types to the system. This appears to have several benefits:

In a user-defined class, you can specify the set of values that can be created as instances of a type. To do this, a class can take an external representation of the desired value and only create a domain value object when the representation can be transferred into a valid (well-formed) value.

In this connection, we could implement further concepts, such as adding a so-called "undefined" (bottom) value and other special values to the set of valid values. The benefit is that you can distinguish explicitly between defined and special values, as suggested by Ward Cunningham. For example, many developers would use a workaround like 999 for a yet-unknown account number they have to represent as an integer. This means that, by convention, they turned a defined value of type integer into a special undefined value. Problems occur in programs that do not observe this convention; the above method would cause serious domain errors that such a program would not catch. To avoid this problem, you must not handle domain values as a simple set of values.

Operations on values subjects

We have known since the seminal work of Tony Hoare that typing should include not only the defined set of values, but also the operations permissible on values of that type. Although this had been hard to implement for user-defined types in classic imperative languages, we actually get this option "for free" in object-oriented languages.

The Bank example

For example, we can think of permissible operations on values of the type `AccountNumber`. This is not as trivial as it may sound, because our account numbers should allow addition as well as relational operations. The reason is that the sum of account numbers is normally transmitted as a checksum in batch money transfers between banks. In addition, some banks add additional information to an account number. For example, you can derive the customer number or the type of account from an account number. All of these are permissible operations of the type.

Together with the introduction of special values, you can also define appropriate semantics for handling these values. In our bank example, an attempt to add an undefined account number to a "normal" number could lead to an exception.

Building domain values

On the other hand, implementing domain value types as classes has some drawbacks. To avoid a serious trap we have to take care that values belonging to domain value classes are always handled by value semantics rather than by reference semantics. This means that a domain value object must not be accessible for modification over two identifiers. More specifically, a domain value object should not be modifiable at all. This is the only way to maintain the referential transparency of the values we require. Some of the proposed solutions will be discussed in Section 8.10. In summary, we have compiled the following different techniques for building domain values:

- Domain values are instances of "normal" classes. Domain value objects are always passed as copies. This is an insecure programming convention.
- Domain values are built from classes allowing one single value-setting operation, while otherwise offering only probing operations. This uses a lot of memory.
- Refined techniques to build value objects are known from the literature by the name of body/handle (see James Coplien).
- A pattern that implements value objects with minimum space requirements is the Flyweight pattern of Gamma et al.

2.7 METAOBJECT PROTOCOLS

2.7.1 Introduction

When developing interactive programs, we often observe that it is not sufficient to write program code to implement domain concepts or objects. We additionally have to be able to handle the elements of our program, that is, we have to be able to query and modify properties of objects and classes in our program. How important this meta-information really is becomes clear when we work with components. We have to rely on components giving us information about their interfaces and other properties, because such program code is normally inaccessible. This section discusses such meta-information. It is aimed at the more advanced reader because using or even designing metaobject protocols is not an everyday task. But when dveloping programming tools or components, you should understand the concepts and means of this type of meta-programming.

2.7.2 Motivation for a Metaobject Protocol

Simply speaking, we have used objects in object-oriented application development to model the concepts and things of a real-world application domain. Now let's have a look at software engineering as a potential application domain. We normally do this when building programming tools or other software development tools. If software engineering is our application domain, then there is nothing that would deter us from selecting an object-oriented program as our object of modeling. To work with this object, we need a metaobject protocol, which should sound familiar to Smalltalk programmers. The following section explains what a metaobject protocol is all about. Figure 2.14 shows a schematic view of modeling an application domain versus modeling a software program.

2.7.3 Definition: Metaobject Protocol (MOP)

A *metaobject protocol* (*MOP*) is based on the constructs of the programming language you use to write your program. These constructs themselves can be modeled as objects. Such objects are called *metaobjects*. Like all objects, metaobjects are instances of classes, only we call them *metaclasses*. The set of interfaces of these metaclasses form a metaobject protocol.

Metaobjects can be classified in two categories:

1. Objects representing the (static) application model (e.g., which operations does an object offer).
2. Objects representing the runtime system (e.g., how an operation is executed; how an object accesses its attributes).

The organization of metaclasses and the behavior of metaobjects are described in a metalevel architecture. Metalevel architectures can be used for many different

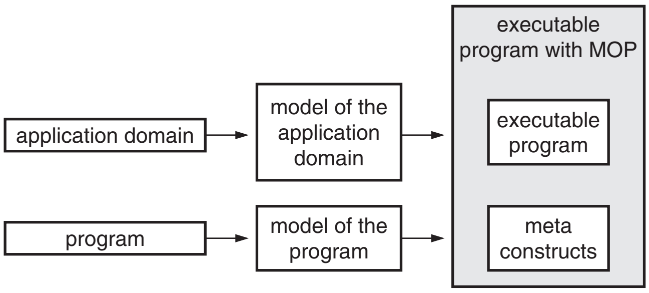


FIGURE 2.14
Modeling an application domain versus modeling a software program.

technical purposes. However, they are normally required only for large-scale application systems. The following section will, therefore, focus only on well-known applications.

2.7.4 Representing Your Application Model

We briefly mentioned in Section 2.1.20 that utilizing polymorphism in your development project means that the reference you obtain on an object may not be of the same type as the object itself but of a supertype of the object's type.

Using Runtime Type Information (RTTI) to implement a downcast

In some situations, this supertype reference is not sufficient. What we need is a so-called downcast, or a conversion of a supertype reference into a subtype reference. Downcasting is useful in the following situations:

- In a specialized team of classes, one partner obtains the reference to another partner as a reference of the type of the abstract superclass, but it requires the full interface.
- A container with objects pertaining to different classes can only return a reference to the common superclass of the objects it contains. If you need to access an object taken from the container over its full interface, then you need a downcast.
- One example for a special case is building an object table for the entire system. This table includes references to all objects existing in the system and is continually updated. Such a table can be used to run queries on objects, similar to querying a database: for example, "Find all devices purchased more than three years ago." For this purpose, we need to be able to query the class membership of each object as well as the references on objects having the same type, as the queried class has to be converted to the queried type.

To recover the original type information, each MOP lets you query the type of an object at runtime. You would normally represent the type of an object by its class object. Downcasting on an object level triggers a test for an inheritance relationship on the class object level (metalevel).

To make our programming lives easier and to hide the complexity of the metalevel, MOPs of typed languages (e.g., Java) offer a special operator to implement a downcast. This operator internally accesses the class metaobjects and checks for conformity of the downcast before it runs it. If there is no such protection mechanism, a downcast can cause critical errors in programs (e.g., as in C++), and it should be used only together with strictly controlled programming conventions.

Information on the state representation to implement generic operations

Direct access to the state representations of objects, and avoiding their interface, is normally one of the most serious programming sins you can possibly think of. You should bear in mind that the class offers a set of domain-oriented operations, hiding access to the internal state representation. However, there are also technical tasks, for which a specific interface should allow direct access to the state representation. The following are examples for tasks that have to have access to the state representation of an object:

- Generic relational and copy operations; for example, copying business cards and identification cards to a clipboard and subsequently inserting the copy into the room plan by use of generic copy operations.
- Object input and output; for example saving objects to files, connecting to relational databases, or transporting objects within a distributed system.
- Garbage collection.

If access to the representation of an object state is allowed, then we can specify an algorithm for these services for all classes. Otherwise, we would have to program the corresponding operations for each class. You can access the state of an object in either of the following two ways:

- The required information is implicitly available, because the required operations are generated automatically. For example, Eiffel uses relational copy and input and output operations for each class, without the developer's intervention. This approach has several benefits. The developer does not have to program anything manually. The encapsulation of objects cannot be violated. And finally, the runtime behavior is improved, especially in compiler languages. On the other hand, a major drawback is poor flexibility: the developer cannot modify access operations or add new operations.
- To represent an object state, the developer can use a so-called attribute dictionary. This list maintains one attribute metaobject for each attribute, including the following information (which may differ, depending on the language):
  - name of the variable;
  - type of the variable;
  - size of the variable;
  - reference to the variable (for actual access); and
  - other attributes (e.g., properties like "persistent" or "not persistent").
- Based on the above information and information about the superclasses, the developer can write operations that iterate over the attribute metaobject list, running a specific action for each variable (e.g., copy, compare, save to database). Major drawbacks of this variant are loss in state encapsulation and poor runtime behavior.

Information about interfaces to implement dynamic operation calls

It may be desirable in many cases to open up an application to the "outside," that is, to allow other applications to call arbitrary public operations. For example, you can use an external interpreter to program frequently recurring processes in a script.

Take our device management system as an example. If, after upgrading all PCs of one type, you want to update the device data in the room plan you could write a script. This script could then iterate over all old devices in a room plan and update the device data automatically.

To allow this flexibility, a MOP should support an operations list, similar to the attribute list described above, to handle queries on existing operations and call such operations. Languages like Java support this technique by the so-called introspection. Such an operations list maintains a metaobject entry for each operation with the following information:

- name of the operation;
- signature of the operation; and
- other attributes.

To then call an operation, you either use the operation's metaobject or a special operation (metacall) of the public object. This operation will then call the actual operation with the name you passed and the passed parameters.

2.7.5 Representing Your Runtime System

Though software developers are normally happy when they don't have to know the details behind the implementation of a language, there are situations where you have to deal with the implementation. For this purpose, an MOP can present a well-defined interface, allowing you to redirect the control flow within the runtime system. This is of interest, particularly in two places:

1. where you store instance variables, and
2. where you call operations.

Manipulating the storage of instance variables

To disclose the storage algorithm of instance variables, you need two additional operations, that is, one to read an instance variable and another to set an instance variable.

For example, Smalltalk uses the method `instVarAt: i` to return the value of the *i*th instance variable. The method `instVarAt: i put: value` assigns `value` to the *i*th instance variable. These two methods are called every time an instance variable is accessed. By overloading these methods, the standard storage algorithm can be replaced by your own algorithm for all classes or part of the class tree. Overloading the standard storage algorithm may be useful for efficiency reasons, or to store objects externally.

For objects with many instance variables that are seldom used, it may prove more efficient, in terms of required storage space, to create entries for each of these instance variables in a hash table instead of storing them in a consecutive memory block.

When the state of an object is to be updated immediately to reflect each change to a database, you also need to manipulate the storage mechanism. You can use the mechanisms to represent your application model (see Section 2.7.4) only to implement explicit operation calls to save entire objects.

Manipulating operation calls (redirecting operations)

You can use a metaoperation, to which each operation call is directed, to access the operation call mechanism. This metaoperation can run arbitrary actions either before or after the actual execution. Redirecting operations is useful for the following application domains:

- *Distributed applications*: In this case, the above mechanism lets you redirect operation calls for an object to another computer. To redirect operation calls, you pack the name and arguments of the operation before it is called (also called *marshalling*) and send the package to the server, where the actual operation runs. The result is then returned to the client, where it is unpacked and forwarded to the caller. The static CORBA (common object request broker architecture) interface is normally implemented in this way.
- *Persistent objects*: In a manner similar to the mechanism used for distributed objects, when an operation is called for an object not residing in the main memory, the call is caught. Based on the object reference, the referenced object is then loaded from a database, before the actual operation call of the freshly instantiated object is executed. In CORBA, this would correspond to the "incarnation" of an object by an object adapter.
- *Additional handling of operation calls*: There is a large number of algorithms you may want to apply on an operation call before or after it is executed. Good examples are the logging or auditing of operation calls. Another more sophisticated algorithm is used to check the authenticity of the caller and to guarantee to a certain extent that the client really is authorized to call that operation. This is relevant for virtually all software systems.

There are different implementation techniques to catch an operation call and have it handled by a metaobject before or after it is executed on the actual target object. The most common technique hides the target object from the client by using an intermediate proxy object. The proxy object catches the operation call and redirects it to the metaobject in charge. In simple cases, it can assume the metahandling itself, as in the example relating to security proxies described in Erich Gamma et al. (Chapter 4) or Frank Buschmann et al. (Section 3.4).

Another implementation technique modifies the virtual machine of a language so that it can work with enhanced object references. The enhancement is the so-called *pointer swizzling*, where a call is forwarded to a corresponding metaobject. For example, the object-oriented database ObjectStore uses this technique; although it does not forward calls to a generically accessible metaobject. Instead, it merely uses a restore mechanism to wake up the called object in the database.

2.8 REFERENCES

This chapter on the T&M object metamodel has been kept quite concise. For readers who want to delve deeper into the material, we have assembled an annotated list of references that contains both practical and theoretical work.

R. J. Bird and P. Wadler: Introduction to Functional Programming. New York: Prentice Hall, 1988.

A seminal textbook on functional programming.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-Oriented Software Architecture—A System of Patterns*. Chichester, New York: Wiley & Sons Ltd, 1996.

A well-received pattern collection book.

L. Cardelli, P. Wegner: "On Understanding Types, Data Abstraction, and Polymorphism". *Computing Surveys*, Vol. 17, No. 4, Dec. 1985.

Seminal work on type systems.

J. Coplien: *Advanced C++: Programming Styles and Idioms*. Reading, Mass.: Addison-Wesley, 1992.

Coplien introduces, among others, the body/handle pattern.

W. Cunningham: "*The CHECKS Pattern Language of Information Integrity*." In J. O. Coplien and D. C. Schmidt (eds.): *Pattern Languages of Program Design*. Reading, Mass.: Addison-Wesley, 1995. Chapter 3, pp. 145–156.

This is an interesting work on (domain) values in object-oriented languages.

E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*. Reading, Mass.: Addison-Wesley, 1995.

The present chapter refers to patterns from this book.

A. Goldberg, D. Robson: *Smalltalk-80: The Language*. Reading, Mass.: Addison-Wesley, 1989.

This is a classic among programming language textbooks.

C. A. R. Hoare: "*Notes on Data Structuring*." In O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare: *Structured Programming*. London: Academic Press, 1972.

This is a theoretical classic on the concept of type and data types.

R. E. Johnson, V. F. Russo: *Reusing Object-Oriented Designs*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1991.

This work offers more on abstract classes.

G. Kiczales, J. des Riverieres, D. G. Bobrow: *The Art of the Metaobject Protocol*. Cambridge, Mass: MIT Press, 1992.

This work offers more on metaobject protocols.

B. Liskov: "*Data Abstraction and Hierarchy*." OOPSLA 1987 Addendum to the Proceedings, *ACM SIGPLAN Notices*, Vol. 23, No. 5, May 1988, pp. 17–34.

B. Liskov, J. W. Wing: *Family Values: A Behavioral Notion of Subtyping*. Technical Report, Carnegie Mellon University, CMU-CS-93–187, 1993.

These fundamental theoretical works treat abstract data types and the concept of type.

D. C. Luckham, J. Vera, S. Meldal: *Three Concepts of System Architecture*. Technical Report CSL-TR-95–674. Stanford, Calif.: Stanford University, 1995.

This work offers more on interfaces.

B. J. MacLennan: "*Values and Objects in Programming Languages*." *ACM SIGPLAN Notices*, Vol. 17, No. 12, Dec. 1982.

This classic article distinguishes values and objects in programming languages.

P. Maes, B. Nardi (eds.): *Meta-Level Architectures and Reflection*. The Netherlands, Amsterdam: North-Holland Publishers, 1988.

This book offers more on the metaobject protocol.

B. Meyer: *Object-Oriented Software Construction*. Second Edition. New York, London: Prentice-Hall, 1997.

This fundamental book covers the concepts and mechanisms of object-oriented programming and programming languages. The explanation of the contract model is particularly important.

D. L. Parnas: "On the Criteria to Be Used in Decomposing Systems into Modules." *Communication of the ACM*, Vol. 5, No. 12, December 1972, pp. 1053–1058.

This classical software engineering paper introduces important principles of modularization and information hiding.

T. Reenskaug, P. Wold, O. A. Lehne: *Working with Objects*. Greenwich: Manning, 1996.

This book presents interesting aspects of object-oriented design, especially regarding role-modeling.

R. W. Sebesta: *Concepts of Programming Languages*. Reading, Mass.: Addison-Wesley, 1998.

This is a good practice-oriented textbook about the concepts and constructs of programming languages; it presents design alternatives as examples.

P. Wegner: "*Concepts and Paradigms of Object-Oriented Programming*." *OOPS Messenger*, Vol. 1, No. 1, August 1990, pp. 8–87.

This conceptual article contains the fundamental definition of the term "object-oriented."

C. Zimmermann (ed.): *Advances in Object-Oriented Metalevel Architectures and Reflection*. Boca Raton, Florida.: CRC Press, 1996.

Another interesting paper on metaobject protocols.

3 Guiding Metaphors and Design Metaphors

3.1 INTRODUCTION

Application-oriented software development requires more than the purely technical elements of an object-oriented model. What we also need is a design view and a language to be able to think and talk about our daily development work, the tasks related to it, and its support. At the same time, our future system should initially develop as a kind of "vision" in the minds of the developers. To support this design process, our approach uses guiding metaphors and design metaphors, helping us to form a view to orient ourselves in the design of an interactive application software. They also give us a common language so that we can discuss our designs and systems.

This chapter outlines the "essence" of the T&M approach, a view of software development involving the guidance provided by concepts like guiding metaphor and design metaphors that make the T&M approach different from other method books. By the end of this chapter the reader will better understand the specific application-oriented perspective through which all the techniques, methods, and strategies described in the rest of this book may be seen as a uniform approach to software development.

3.2 DESIGNING APPLICATION SOFTWARE

An object-oriented model defines the elements we can basically use to model our software. Though this specifies the means to express ourselves, it does not tell us what we should model or how. What and how we should model is initially a domain-related question, because unless we have a suitable model of the application domain and its constructive implementation in a workable system, application software has no right to exist.

Designing an application system

Unfortunately, a pure domain model won't answer the question about how we might design an interactive application system based on our ideas. *Design* refers to the layout of the graphic user interface (GUI) and different options to interact with the computer, that is, what is usually called the look & feel of the software. But more importantly, we think that design also refers to a usage model. The following section explains what a usage model means and what we use it for.

3.2.1 Definition: The Usage Model

A *usage model* by our definition is a domain-specific model telling how an application software can be used to do a job within the specified application domain.

A usage model is based on an idea about how the software should be manipulated and represented, and it includes all domain objects, concepts, and processes supported by this software.

It makes sense to use design metaphors based on a guiding metaphor to realize a usage model.

Following along the line of this definition, a usage model will give us an idea or image of how the developers and the future users see the system in the application context. This means that the usage model should tell us what means we should use and what kind of support we should provide for the tasks on hand. Thus we can say that our usage model goes "below" the mere GUI design and encompasses the mental model of interacting with the system in a task-oriented way.

The idea or image of a usage model that the participating parties have should not be solely oriented to a single purpose. In fact, it should be a harmonious design, so that we will be able to speak of a unity of form and content when referring to our application software. For this reason, we look for guidelines to direct us in formulating our usage model. Obviously, knowledge of purely object-oriented model elements won't do the trick.

3.2.2 Background: Methodology Books versus Design Guidelines

A look at methodology literature has proven to be of little help in designing application software. The reason is that this topic is normally dealt with only briefly, although we consider it a real problem. Falling back on the design of conventional application software also leads to a sobering result: The dominance of the analysis of work processes and their flow-oriented modeling, combined with hierarchical decomposition methods, have led to a huge number of rather one-sided systems that all link menu trees to screen sequences. Even modern workstation computers with their graphical user interfaces can change little in such a sequential system design. Unfortunately, menus and screens borrowed from vintage mainframe applications have found their lucky revival in the form of menu bars and modal dialogs.

Design guidelines

If we really want to utilize the possibilities of object orientation and modern system platforms, then we need *design guidelines* describing how such a system should look. These design guidelines should primarily have a technical character, because their purpose is to help us in our daily jobs. Beyond pure technologies, we find approaches, such as software ergonomics, HCI (human computer interaction) or CSCW (computer supported cooperative work) that deal with the ergonomic aspects of modern software. The reader should understand that this book is not dealing with software ergonomics and HCI, so you should not look for brilliant interface designs in our examples. We try to impart the idea of how to design a task-oriented usage model with its elements and fundamental interactions. Chapter 10 is dedicated to CSCW.

A rather well-known and sophisticated usage model for object-oriented application systems is “People, Places, and Things” proposed by Taligent. This usage model represents an important step towards what we discuss in this book, only we focus on a “human interface.” What we try to achieve is a unity of domain contents and the forms in which they are represented. In that respect, we feel like toolmakers or machine designers, except that we don’t have a century of tradition to look back on. We still know little about what good interactive software should look like and how it should support its specific purpose.

3.3 GUIDING METAPHORS FOR APPLICATION SOFTWARE

Our starting point for the development of a usage model is a *concrete image* of how our future system will be used. Naturally, we cannot give a general definition of this image. What we can anticipate are ideas about how work is conceived and what kind of support is expected. We refer to such a basic image as a guiding metaphor.

> **A *guiding metaphor*—or leading motif—is a theme or other coherent idea, clearly defined and named, whose purpose is to represent or symbolize a person, object, place, idea, or state of mind.**

In German, we originally used the term “Leitmotif” which was coined for a recurring theme in music (by the German Composer Richard Wagner) and then used in psychological or philosophical disciplines. More recently, it has been used in connection with software development. But for the English edition, we decided to introduce the term ‘guiding metaphor’:

> **For our purposes, a *guiding metaphor* is a basic viewpoint that helps us perceive, understand, and design a piece of reality.**
>
> **In software development, a guiding metaphor provides a common orientation for all participating groups throughout the development process. It supports the design, use, and evaluation of software and is based on value concepts and objectives.**
>
> **A guiding metaphor can have a constructive or an analytical function.**

A guiding metaphor has onstructive and analytical functions

When a guiding metaphor is used to help the development team to transfer the model of an application domain to the design of a future system, it has a constructive function. However, we can also use a guiding metaphor in its analytical function, that is, when we want to evaluate existing application software. A guiding metaphor plays a similar role when it is used by users to better understand and use an application program. In this book, as in the T&M approach in general, we use guiding metaphors primarily in their constructive function, for software development.

Guiding Metaphor	Design Goal	Role of Users	Role of Developers
Object Worlds	Virtual worlds populated with active objects (e.g., agents)	Originators of initial orders; triggers	Creators of miniworlds
Direct Manipulation	Manipulating working objects like using our hands	Independent workers, actors	Constructors of artifacts
Factory	Guiding and controlling work in assembly line–like fashion	Machine operators	Machine constructors and machine fitters
Expert Workplace	Supporting experts in their work by creating a work environment with adequate equipment	Experts	Tool designers, work place designers

TABLE 3.1 Using guiding metaphors in software development.

3.3.1 Background: Guiding Metaphors in Software Development

In the software development discipline there are various explicit or implicit guiding metaphors. For example, you can find the guiding metaphor of a factory or a computer as a communicating partner. Exploring all kinds of guiding metaphors would go beyond the scope of this book; we will use a few guiding metaphors of interest in view of object orientation to show what properties are relevant (see Table 3.1). The first property you should note is that a guiding metaphor introduces roles with different values for all participating people, particularly developers and users.

3.3.2 The "Object Worlds" Guiding Metaphor

Let's look at the *object worlds* guiding metaphor that is often found in the Smalltalk community, where object orientation has virtually become a guiding metaphor in its own right. This guiding metaphor is based on virtual worlds, populated with active and communicating system objects. These objects are occasionally triggered from the outside, and then they act in their respective roles as "actors" or "agents."

Actors, agents, and artifacts

This guiding metaphor fits well into the concept of artificial, intelligent artifacts. Users of such a system are often reduced to marginal figures, merely giving impulses or initial orders but having little to do with the actual things that happen in the system. In contrast, the developers are the creators of these miniworlds. They overlook all interactions and plan the entire "game."

The EMS example

If we try to transfer the object worlds guiding metaphor to our EMS example, we are faced with problems. Our objects would most likely be employees, devices, and rooms. What would be missing is a simple idea about how procurements should be handled. How should we inform devices and employees when and how a new device is to be purchased or updated? All rules and special cases for procuring new devices or upgrading old ones would have to be defined either in device objects or in employee objects. If this is not possible, we have to introduce a person acting as device manager to the game, which wouldn't be an easy task. This problem is often solved by introducing a procurement object or a buyer agent. But still, all conceivable events relating to a procurement process would have to be reflected in the actions of these objects.

3.3.3 The Direct Manipulation Guiding Metaphor

Interactive systems primarily use a guiding metaphor based on *direct manipulation of work objects*. This means that individual objects of a complete application system should be designed so that we have the illusion of manipulating them directly with our hands, as in the real world. These objects are often represented as direct (mimetic) images. A once well-known example was the HyperCard system on the Macintosh, the first system of its kind; it was introduced in the 1980s and was probably the most popular hypertext system. HyperCard was based on the idea of using linked user-defined filing cards.

The EMS example

If we consider the concept of a guiding metaphor based on direct manipulation in our EMS example, the device manager could create device identification cards and business cards, directly manipulate all these cards, and file them in electronic files and folders. A room plan could be used, and the device manager could drag an identification card and drop it onto a room in the room plan on his desktop.

3.3.4 Discussion: Guiding Metaphor of Direct Manipulation

The guiding metaphor of direct manipulation introduced in the previous section is very simple and easy to understand. However, its simplicity also means that is rather limited. More complex functions, such as statistics on procurement costs, budget calculations, and similar functions, cannot be created as easily as cards in a cardbox system, based on direct manipulation. In particular, the interaction of several objects in a work situation is difficult to formulate consistently within a system.

The Bank example

In fact, looking again at our bank example, we can see how difficult this would be. Consider this problem: A bank wants to implement complex financial operations, such as granting business credits by analyzing and evaluating corporate balance sheets, taking the last account movements and the current business plans into account. Within the guiding metaphor of direct manipulation, rating a customer would be a problem. Objects like balance sheet, account, and business plan would either have to directly interact for rating or the user would have to drop the relevant objects on a creditworthiness object for this purpose.

Although the guiding metaphor based on direct manipulation seems rather limited for complex application systems, the role distribution appears to be much more useful than that of the active objects in virtual object worlds. The reason for this is that, in the direct manipulation guiding metaphor, users are always the main players, determining independently what should happen with which objects. In this scenario, the developers play the role of suppliers of such useful objects. They become the constructors of artifacts, or mappings of known objects. What's missing in this guiding metaphor is control over the entire system. In fact, the developers are limited to developing single components. Though some may consider this a welcome self-constraint, it often has the drawback that the individual components do not fit well.

In summary, these examples show another important feature of guiding metaphors: They can become more intuitive by use of appropriate metaphors like filing cards or forms.

3.3.5 The Guiding Metaphor of the Factory

One of the classic guiding metaphors is the *factory*, because this guiding metaphor has been extremely important in traditional software development. The factory guiding metaphor relates to a factory worker who works by fixed rules. Human work is modeled in a "Tayloristic" style, that is, detached from its environment and reduced to repetitive movements.

Users become operators

If we transfer this model to the use of an application system, then this means that users are guided through their work process by rigid rules, in a manner similar to an assembly line. The user of such a software system becomes a machine operator. The machine (software system) controls the speed and steps (required input) to control the work process.

It is characteristic for the factory guiding metaphor that the worker is considered a potential source of failure and error. As a consequence, this source of error should be either replaced by automation or restricted by control. In this situation, software engineers see themselves as machine constructors, designing and building these software machines or automatons. On site, less qualified machine fitters would then tune the machine to the specific work situation.

The EMS exmaple

If we apply the factory guiding metaphor to our EMS example, we might have the following scenario: The device manager selects the "Change current room plan" option from the main menu, and then the option "Change device description" from the submenu. The system presents a list with all devices and prompts the device manager to "Select the devices to be edited." The device manager doubleclicks a device name to display options to delete, edit, and create identification cards for devices. Once the device manager selects the "Edit device configuration" option, a screen template is displayed for the device manager to enter new data for that device. Subsequently, the program runs a calculation to check whether the modified data match the specified side conditions (the device class referring to the employee), and whether or not the data is consistent with pending procurement processes. In the positive case, the system outputs an appropriate message. In the negative case, the system reports an inconsistency and takes the user back to the input menu. If the device manager knows nothing about the procurement process for a specific device, then she has to return to the main menu and select the "Edit procurements" option. The entire device management functionality is handled like this, that is, sequentially and based on a rigid user interface.

3.3.6 Discussion: The Factory Guiding Metaphor

The EMS example shows the character and limits of the factory guiding metaphor. The user is guided step by step by the program and has a few options to choose from after each selection. This rigid user interface means that such an application could be "operated" by people with little knowledge of computers and some limited knowledge of the application domain. On the other hand, a rigid flow control like this hinders flexible action. For instance, there is no parallel view of procurement processes and room plans in our example, which means that it is simply unavailable. The device manager has to adapt to the program flow.

In software engineering, the factory guiding metaphor, here called *software factory*, is still a current issue. It is interesting to note that the driving motivation for the introduction of the software factory is said to have been a lack of qualified programmers or costs of programmers.

The factory guiding metaphor is the best-known example of a generic *process-controlled view*. This view is characteristic for conventional application software development.

Process-controlled view

Developing application software based on a process-controlled view means that we implement operations that replace or regulate and control human work. Here, the program has control over the process or sequence of human work steps. This means that the process-controlled view is behind the concept of automation, which pursues the general goal of replacing human work by machines, or at least reducing it to data input. For this purpose, plans and regulations, which are implemented by algorithms on computers, are normally used.

Workflow management systems

This process control idea has been used in areas outside the classic factory based on the ideas of Henry Ford. For example, office automation has been implemented under the catchword "workflow management system." Behind the process-controlled view, there is always the goal of refining individual actions so that they can be executed or controlled by a machine.

It is worth noting at this point that this view is not fundamentally bad. In fact, software automatons have been built for many jobs that were considered tedious, expensive, health-damaging, or simply disturbing. Today, however, many work situations are too complex to allow their formalization and implementation in software based on the process-controlled view. We will explore this issue further in Section 12.2.2.

3.4 DESIGN METAPHORS

This section discusses design metaphors. We first provide a definition of what design metaphors are, and then discuss some background of using metaphors and what metaphors mean in the context of our T&M design.

Design metaphors round out the image evoked by a guiding metaphor. They provide the necessary details and the actual elements of a usage model. On the constructive side, they relate the usage model to conceptual and design patterns. Metaphors are an essential characteristic of application-oriented software systems developed according to the T&M approach.

3.4.1 Definition: Metaphors

Generic metaphors

In the generic context, a *metaphor* is a figure of speech in which a word or phrase literally denoting one object or idea is used in place of another to suggest a likeness or analogy between them. Metaphors are a figurative language that we use daily and to such an extent that we are hardly aware of it. For example, if someone says a sentence like, "that was close to the limit," who would think of a borderline between two countries?

A *metaphor* in its generic form takes a figurative expression (e.g., garbage) from a context (e.g., household) and uses it to replace the actual expression (e.g., useless data in storage). Metaphors (like garbage) emphasize certain properties or views in the original expression (e.g., useless things one wants to get rid of).

Design metaphors

In the T&M our context, we use metaphors purposefully to better convey our guiding metaphors, thereby increasing their effect. For example, when talking about the guiding metaphor of direct manipulation, the metaphor of a filing cardbox and its cards makes it easier to understand what we mean. To emphasize that we are using metaphors to design software systems in our context, we speak of design metaphors.

A *design metaphor* is a figurative and objectified idea that details a guiding metaphor from the domain and software views.

A design metaphor structures the perception and contributes to our concept model. It guides the idea and communication of what should be analyzed, modeled, and implemented from the domain perspective.

A design metaphor serves to design software systems by facilitating their handling and functionality for all participants. It becomes part of the usage model. In the T&M approach, a design metaphor is also related to appropriate construction guidelines and design patterns.

For example, design metaphors for an office environment appear to be very useful for office communication systems. We saw this earlier in the section discussing the direct manipulation guiding metaphor. We very quickly got used to moving folders back and forth on an electronic desktop, putting documents in folders and others in the recycle bin. However, most of us also noticed the limits of these metaphors, when they are the only ones available. In fact, we find that they are helpful only for elementary tasks. In more complex applications, such as a graphics editor, this guiding metaphor is normally insufficient. What's missing are special tools to do routine jobs, like clicking a tools' palette to draw ovals and rectangles quickly and shortcuts for connecting many boxes by arrows.

3.5 T&M GUIDING METAPHORS AND DESIGN METAPHORS

In the T&M approach, we look for suitable guiding metaphors that can direct us in turning the domain model into a design of our future system. Each guiding metaphor should be realized by intuitive and figurative design metaphors, giving us detailed guidelines to refine our design of the system components involved. After all, the different roles and their values given by a guiding metaphor should fit the requirements of our future users. This means that we have certain requirements for a guiding metaphor and its design metaphors.

- The guiding metaphor and its design metaphors have to fit seamlessly.
- The guiding metaphor and its design metaphors should support us in analyzing the application domain, the design, and the use of our future system.
- The design metaphors should allow both a domain-specific and a software-specific interpretation.

3.5.1 A T&M Guiding Metaphor: The Expert Workplace

The T&M approach is based on two historical and conceptual roots, that is, the work of software developers and the office work in the financial and service industries.

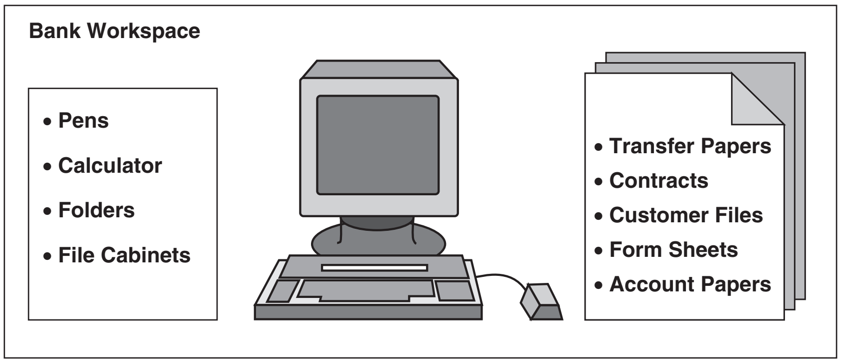


FIGURE 3.1 Example showing tools and materials for a bank workplace.

For these two application domains, we often use the expert workplace for autonomous activities—in other words, as our explicit guiding metaphor (see Table 3.1). Examples include the workplace of an account manager in a bank, a software developer in a software company, and a device manager in a software development team.

This guiding metaphor is based on a supportive view. Experts are supported in their work in that we create an work environment for them. In this work environment, these experts should find the tools and materials they need to complete their tasks.

Tools and materials

Initially, we use tools and materials as our design metaphors to elaborate the guiding metaphor for the *expert workplace*. While folders, forms, and program code fall under the term *materials*, things like debugger, profit model calculator, and room planner belong to *tools*, according to our definition. For example, Figure 3.1 shows a simplified workplace for an account manager in a bank.

In the course of our training seminars and courses, we have observed that *assigning* daily work objects to either the tools or the materials category appears to be intuitive, with occasional discussions about whether or not a folder should belong to tools or materials. We also often notice that a pen is considered a tool when students use it to write on paper, but not when they use it in combination with a sharpener to sharp it. In summary, we need to bear in mind that assignments will have to be oriented to the specific work situation in a project.

What should be understood for now is that when experts are completing their daily tasks, they use tools to manipulate materials, which eventually become part of the finished work result.

3.5.2 Background: The Supportive View

The guiding metaphor we presented in the previous section is linked to a primary objective and set of values that we call the *supportive view*. By this view, qualified human work is seen as irreplaceable in many disciplines. Many human engineering scientists and economists have suggested that the objectives of the process-controlled view is no longer a desirable work environment, at least in large part. Rather, they suggest that human work should be seen as an important factor that can contribute to securing market positions and producing high-quality services. Wherever these objectives are in the foreground, then human work should not be replaced by software automatons but instead be supported by flexible software components. All of this changes the image of the 'naive' user or 'operator' to a domain expert who cannot and should not be replaced by software.

Another characteristic of the supportive view is that the tools and materials are of utmost importance in modeling the application system, while generic routines and processes are less important. The reason is that the domain expert should be able, in any situation and based on his or her experience and skills, to do what needs to be done to complete the tasks on hand in a creative or self-regulated way. This important characteristic of expert activities has to be encouraged rather than blocked by an application system. The logical consequence for our approach is that humans should maintain control of the use of tools and materials.

Expert activities and plans

Expert activities are not planless actions, but the plans on which they are based have no rigid instructions for action. Rather, they help the expert in a form of objectified experience. Like a city map, they give some orientation, without dictating each turn to take. Like a recipe, they provide step-by-step instruction, but no detailed directives for those with no cooking experience at all. Section 10.4 will discuss ways that we can integrate plans into the T&M approach based on the supportive view.

Excursion

Brief Excursion into the Cultural and Philosophical History of Tools and Materials.

Cultural history tells us that human work has been characterized by handling tools and materials. Although the question whether the use of tools contributed to the evolution of humankind remains unanswered, there is agreement about the fact that human work is essentially based on the use of tools. For example, in his *Being and Time,* Martin Heidegger suggested the complementary significance of using tools and materials in the context of work. In particular, he argues that humans take the use of tools and materials (the 'stuff') for granted. This applies not only to handicraft or producing disciplines, but also to office work. Other authors, including Lewis Mumford have suggested the important role of containers in human culture.

3.5.3 Discussion: Metaphors and Patterns

Design metaphors help us to better understand and analyze an application domain. They also play an important role when we speak of designing and using application systems. This describes the domain-specific meaning of design metaphors. However, design metaphors are not sufficient for us in our job as software engineers. After all, we do not want to leave up to individual developers how a metaphor is turned into a software design and how this design should be implemented. Apart from all the domain-specific differentiation required in each application system, we want our design metaphors to give some guidance for how they can be turned into a software design and fitting architecture. This is the point where design patterns and frameworks come in handy. Chapter 4 is dedicated to patterns and frameworks. For now, it is important to understand what design metaphors are, how they relate to the T&M approach, and more specifically, how they prove useful in designing an expert workplace. The following sections define what tool, material, work environment, automaton, and container all mean, and then discuss how they can be used as design metaphors and how they relate to the T&M approach.

3.5.4 Definition: A Tool

This section defines the term 'tool' in the context of daily chores, such as, handicraft, household, or office work.

> **In this context, a *tool* is an object that people can use to alter or probe materials to complete specific tasks.**
>
> **Tools are normally suitable for different domain-specific purposes and to handle different materials. Naturally, the main prerequisite is to handle them properly. Tools are particularly useful for recurring actions that they, to a certain degree, embody.**
>
> **Many conceptual properties of (hand) tools can be transferred to software tools, but it normally makes no sense to directly map the way a tool is handled and its form to a software system.**

3.5.5 The Tool as a Design Metaphor

Altering and probing

When talking about the characteristics of a tool it is important to understand that *altering* and *probing states* cannot be separated. For example, only when we hear the sound of a hammer and see and feel the nail's resistance can we continue driving a nail into a wall without incurring a problem.

A tool is linked both to a domain function ("it serves a specific purpose") and a specific way to handle it ("it is easy to use"). Obviously, this emphasizes the domain functionality. If you want to drive a nail into a wall, then you have to use a suitable tool. However, when you have more than one option to complete this chore, you may want to choose between different types of hammers.

Handling tools

Handling tools normally requires some experience. For example, you would not be able to trim wood unless you knew how to use a chisel. Depending on the situation and the job we want to get done, we normally decide when to use what tool; in other words, we usually use several tools alternately to handle materials.

A tool embodies a physical routine or series of actions. For example, a hammer embodies the act of hammering. Unless you know what hammering means, you won't have a clue what a hammer is used for.

3.5.6 T&M Design: Software Tools

In this section, we look at tools in the context of computer software. A software tool should be able to probe or alter a material. We, as users, decide when to pick it up or lay it aside, depending on the situation. A software tool should be suitable for various purposes and materials.

The downside is that for a large number of common physical tools it is rather useless to simulate them in computer software. When we try to transfer a physical tool to software, we usually lose most of its characteristics. Even if we can transfer the domain concept behind the routines to be completed by means of a tool, the way that tool will be handled will change. Along with change to the way a tool is handled, we also have to change the tool's shape. For example, it wouldn't make sense to simulate a pen, with its usual shape and handling, on a computer. The attempt to combine such an electronic pen with a computer mouse and use it for writing will result in illegible gibberish. (Those who remember the first version of MacDraw on the Macintosh will know what we are talking about.) It is important to understand the different functions of a pen, that is, writing, drawing, marking, pointing, striking, to be able to implement it in one or more software tools.

The EMS example

Let's assume that the device manager in our EMS example has used until now only a few simple, general-purpose tools, such as a pen, ruler, and eraser, for the sake of simplicity. When we take these tools as the starting point of our domain design, we have to understand the functions of these tools in view of the tasks to be completed by using them. Some of these tasks are: update the room plan, and concile procurement processes and device lists. Our software tools should offer better support for many things the device manager had to write down on notes or memorize. Assume that we have developed a Device Organizer tool. This tool should be used similarly to a pen to add device identifiers to rooms. Additionally, the tool should probe and indicate whether or not the information added to a device identification card matches the specified room sizes and number of employees.

3.5.7 Definition: Material

According to our basic definition, tools and materials are complementary concepts.

> **A *material* is an object that will eventually become part of a work result. Materials are handled by use of tools and automatons and embody domain-specific concepts. They have to be suitable for a given task.**
>
> **The properties of existing objects of work can often be usefully transferred to software materials.**

3.5.8 Material as a Design Metaphor

It is important to note that a material, like a tool, has a domain functionality. We think of specific ways of using a material for different purposes. As we usually use tools to work on a material, it has to be suitable for being handled by use of the tools. Depending on the task at hand, we will often want to handle different materials to achieve a specific result.

A specific work situation will show us clearly what is a tool and what is a material. However, it can happen that a thing is needed as a tool in one situation ("using a pen to write"), while it is a material in another ("using a sharpener to sharpen a pen").

3.5.9 T&M Design: Software Materials

Software materials

Practical experiences have shown that it is easier to transfer objects of work than tools from an application domain onto the computer. In an initial orientation, always look at the materials existing at a specific workplace as potential candidates for implementing software materials. But note that materials in the real world and in software are always more than “data bags.” In order to design good software materials, it is crucial to understand the different ways, that objects of work are handled in an application domain.

The EMS example

If we return to our EMS example, we can immediately think of a few materials, such as the room plan itself and the device identification cards, which hold information about devices and their performance features and procurement processes, as well as employee information. When considering these conventional materials, we try to understand their specific use in the application domain. This means that we cannot content ourselves with generic operations, such as create, modify, and delete, because new devices are *purchased*, employees *move* to other workplaces, and a purchase order may be *canceled*.

3.5.10 Definition: The Work Environment

The guiding metaphor of an expert workplace shows clearly that another important set of design metaphors is the workplace and its environment, that is, the work environment, which represents the location where a job supported by our application software is completed.

A *work environment* is the location where tools, materials, and other objects pertaining to a task are available and arranged in a domain-specific manner.

The actual work is done at the *workplace*, while the work environment as a larger concept includes additional locations accessible within this workplace.

The (individual) workplace is normally protected against unauthorized access. When only the work of a single user is to be supported, then the workplace and work environment are usually identical.

3.5.11 The Work Environment as a Design Metaphor

We learned from the above definition that the *work environment* is the location where tools, materials, and other objects pertaining to a job are available. We said we distinguish between the immediate *workplace*, where the job itself is completed, and the *work environment*, which may include additional locations (rooms and places) relating to that job. We take it for granted that we equip our workplaces according to our tasks, order principles, and preferences. During our work, we usually prepare the objects we need to produce the desired work result. Other things that we may need in rare cases or when problems arise are normally within reach, that is, in our work environment, or we know where to find them.

To start with, let’s assume that our software system has to support a single workplace used by one person. In this scenario, we ordinarily use the terms “work environment” and “workplace” synonymously. As soon as we have to support cooperative work, we will have to distinguish between work environment and workplace (see Section 4.1). Then it is important to decide which locations are for individual use only and which ones are shared or public spaces.

The work environment metaphor introduces a spatial concept to our approach. This understanding of an environment as a location fits well into the guiding metaphor described thus far. The environment represents an important conceptual and spatial dimension with specific limits. First of all, we can see our individual workplace as our private work sphere, where we can use various tools to handle materials. Within the environment of our workplace, we have additional tools and materials that we don't need to use often. Then we have places where we can deposit and access shared materials. We also need locations or media to exchange information about our work.

3.5.12 T&M Design: The Work Environment

This section explores how a work environment is seen in our T&M design. The concept of the environment has not been fully utilized in most current application systems. What you mainly find are electronic desktops. What you seldom find is an extended concept of an environment, except in computer games.

When we try to transfer the spatial concept to a software system, we have to deal with several effects on our design. For example, it is hardly conceivable to use tools available at different workplaces concurrently to handle a specific material. And it is equally hard to imagine that a single material could be in two environments at the same time. This means that we have to see the concept of an environment in a larger context to be able to support cooperative work processes. It will then make sense, for example, to distinguish an individual workplace from jointly used rooms.

The EMS example

We will meet a different design challenge when developing software in the domain of technical embedded systems. Here it frequently pays to design the individual workplace as a control station. Nevertheless, we want to maintain our spatial and temporal concepts.

In our EMS example, we could simulate the work environment as an electronic desktop. Considering that this system is mainly used for office work, a desktop would indeed fit into the device manager's familiar work. On that electronic desktop, we could make tools and materials available that are useful and required to manage devices. The device manager can arrange these objects on the desktop according to his preferences.

3.5.13 Definition: Automatons

Not everything we need in a work environment to complete certain tasks can be classified into our tools and materials categories. Whether we have an office or a workshop, we will find machines doing tiresome routine jobs for us. We call these machines automatons.

An *automaton* is a device used to handle a material. It relieves us from tedious routine work, normally consisting of a defined and recurring sequence of steps, leading to a predefined result, mostly without human intervention.

Automatons normally run in the background, once they have been set and fitted in the work envrionment. Their state can be checked and set to produce a specific result.

3.5.14 The Automaton as a Design Metaphor

Characteristics of automatons

Automatons are well-known concepts in our everyday lives, although we mostly use the word "machine." We speak of a cigarette machine and washing or vending machines, but we mean the same concept.

One of the most important characteristics of automatons is that they complete their work based on a number of setting options and over lengthy periods of time without human intervention. An automaton virtually works at the push of a button.

However, such an automaton will seamlessly fit into our work environment only if it runs a routine activity or a defined process with well-known results. So, today, copiers, printers or fax machines are automatons in almost every office. While the use of automatons is simple and straightforward in standard cases, it can be difficult to take an automaton back into operation when it fails or breaks down, then it normally requires an expert.

3.5.15 T&M Design: Software Automatons

The automaton design metaphor fits well into our guiding metaphor of an expert workplace, because it assumes routine work assigned by the user. This means that the user explicitly transfers tiresome repetitive steps or work processes to an automaton to free time for more creative work. We refer to such automatons as *small automatons*.

Like tools, automatons can handle materials, and they also have a domain-specific functionality. However, they are not handled directly by users. More specifically, a user starts an automaton and intervenes when necessary. As mentioned before, automatons run in the background. This suggests a software-specific modeling of automatic processes.

A different kind of automaton is what we refer to as *big automatons*. A big automaton dictates the speed or pace of work. As in factories or control plants, the user inputs specific data and work steps and controls the automaton's work result.

The EMS example

At first sight, it seems hard to imagine an automaton in our EMS example. This is quite a common situation with expert office work. However, our system in the EMS example has to check all devices to determine whether or not they have to be updated or scrapped in regular intervals. For such a recurring task, we could use an automaton that runs in the background, doing these checks monthly or upon each system start.

This could lead to a T&M design guideline: for supporting expert work, think of designing tools working on materials; for supporting routine tasks, embody the routines as an automaton.

3.5.16 Definition: The Container

When we developed the T&M approach, we initially used containers as materials. Such a container held other materials and was handled by the use of tools. Their only characteristics were that they played a major role in many different application contexts, such as folders or files, and that they had to be modeled explicity for material collections in object-oriented systems. Over time, however, their significance emerged.

A *container* can hold, manage, order, and dispense materials. To handle these tasks, the container normally includes tables of contents.

You can store many similar objects or a defined set of different objects in a container.

Containers often represent processes or workflows (e.g., a credit file is both a *collection* and a *process*) and contribute to cooperation and coordination.

3.5.17 The Container as a Design Metaphor

Obviously, containers play a major role in organizing our everyday working life. First of all, they are useful for storing things that we are not currently working with. But eventually we need to find and retrieve these things again. Then it is important that containers embody specific ordering principles. Tables of contents are another important characteristic of containers.

When we look at cooperative work, containers show additional characterstics: they temporarily combine those materials that are needed for a specific cooperative task or workflow. Frequently, routing slips fixed to such a container help to coordinate distributed work.

Note that containers, in our view, are clearly domain-related and have little to do with data structures like arrays or doubly-linked lists—although we might use these data structures to implement the storing functionality of a container.

3.5.18 T&M Design: Containers

In our effort to transfer the concept of containers to a software system, we primarily take their contents and to a lesser extent their forms. As just mentioned, containers collect different materials, putting them together to a domain entity, such as a customer file. Such a collection is normally listed in a container's table of contents. It is easy to update tables of contents for electronic containers. They show the documents that a container holds. Here, for example, software containers go beyond physical ones, for we expect a container to automatically update its table of contents. In a similar way, many domain containers help to maintain consistency of materials. For example, a clerk can see what documents are still missing, or whether or not a document has been temporarily removed.

Ordering containers

Containers can represent one or more orders, such as a registry composed of folders and files. Beyond the pure ordering of the materials they hold, in alphabetical or chronological order, for example, domain containers can also represent different objects. For instance, it is customary to keep pending applications and forms separate, where so-called resubmission or hold files are often used. In contrast, archives normally hold completed and closed files that are kept either because they may serve as background information later on or because required by law.

Containers and processes

The credit file example just mentioned shows another important aspect of containers. Domain-specific containers often represent workflows or processes, that is the granting of a credit in the bank example given in Section 3.3.4. This means that containers make a process better understandable and reproducible. When we consider that processes are rarely represented by conventional object-oriented methods, we can see the importance of domain containers for the T&M approach. You will frequently find conventional approaches where workflows are realized as processes in the sense of a program procedure, that is a predefined sequence of functions applied on data. Section 4.1 discusses the role of containers in relation to our cooperation and coordination principles.

In summary, there is no doubt that containers represent a conceptual and linguistic enrichment of our set of design metaphors.

The EMS example

In our EMS example, we will encounter several containers. Of course, we need containers for storing our device identification cards and business cards, and most probably they will be designed as files or card boxes. We will design a more elaborate container for storing devices, where its table of contents will show, for example, purchase dates as well as the dates for the next update check. One could even discuss looking at the room plan as a very specific type of container that can be rearranged and that holds domain knowledge about room occupants, devices, and capacity rules.

3.5.19 Discussion: Design Metaphors

A closer look at the characteristics of the design metaphors introduced so far shows that as far as the concepts are concerned, we are focusing on the daily work of people in the application domain. This is the core of application orientation, but it should not lead us to conclude that we are merely mapping existing situations to software systems. Software development always means that new or additional tools, automatons, and materials have to be integrated based on the existing work concepts. This approach may lead to totally new workplaces with new tasks and work processes. But then, these new workplaces and tasks should also fit smoothly into the existing concepts, experiences, and system.

The supportive view (see Section 3.5.2) encourages us to turn away from looking at the use of computers and application software as something exceptional. Software as seen from an application-oriented viewpoint, should be just another means of coping with well-known tasks in a familiar way. Developers should ensure that the domain requirements are implemented in that computer system in an easy-to-understand and intuitive way. Design metaphors help to reach this goal.

3.6 WORKPLACE TYPES

We mentioned earlier that the T&M approach was originally designed for individual workplaces in the financial and service industries. Later, the T&M approach was also used to develop laboratory systems and health care systems. In the course of these applications, we were challenged to review our guiding metaphor of an individual expert workplace for autonomous activities.

We didn't want to force a universal guiding metaphor onto all application domains, which would have conflicted with our application-orientation idea; our software should reflect the real-world needs of each workplace in an organization. Of course, we can't take each single workplace and model and support it in our software, but we can identify groups of workplace types within an organization.

The following sections describe several workplace types that have proven to recur in different application domains, and discusses the most important influences that a workplace type can have on your design of an application software. The reader should note that the approach of identifying and supporting workplace types for software systems is radically different from the one-size-fits-all approach of many application software system. Although the latter strategy seems to be inevitable for most "shrink-wrapped" software, it goes well beyond the possibilities for tailored workplace software. In order to combine an optimum of workplace support with a minimum of development efforts, the T&M approach combines workplace types with a service architecture (see Section 7.10).

3.6.1 Definition: Workplace Types

Depending on the different tasks and work situations to be supported by a software system, we distinguish between different types of workplaces. Each workplace type can be based on its own guiding metaphor.

A *workplace type* supplies a basic abstraction of the corresponding real-world workplaces. We distinguish workplace types by the following characteristics:

- **The amount of flexible and repetitive activities involved in a workplace type.**
- **The kind and extent of equipment involved in a workplace type, including materials and tools.**
- **The amount of domain knowledge and skills of the employees that use a workplace type.**
- **The amount of IT knowledge and skills of the employees that use a workplace type.**

3.6.2 T&M Design: The Expert Workplace Type

Within our T&M design, this workplace type is directly assigned to our original guiding metaphor, that is, the *expert workplace* for people who have to complete different and complex tasks in a flexible way. To better illustrate this workplace type, we have used a few practical examples, for example, account managers in a bank, or the device manager in our EMS example.

Our definition of a workplace type in the previous section tells us a lot about the characteristics that distinguish the number of workplace types. This means that we can characterize this workplace type based on four basic aspects:

1. *Amount of flexible and repetitive activities:* High flexibility, low rate of routines for many and complex activities.
2. *Kind and extent of equipment:* Extensive equipment with tools, materials, and small automatons; usually a combination of individual workplace and common work environment.
3. *Amount of domain knowledge:* High degree of domain skills and knowledge; little need for domain help systems.
4. *Amount of IT knowledge:* Medium to high, as the user has been trained utilizing the application software; low importance of process control or self-explanatory features.

3.6.3 T&M Design: The Functional Workplace Type

This workplace is one of the more common types. We have identified this workplace type in most of our projects, including medical laboratories and health care systems as well as banks. Let's take a look first at the guiding metaphor behind this workplace type.

This is a workplace for experts who should get optimal support for repetitive but less specialized tasks. Examples would be an X-ray analysis workplace for an X-ray physician, or a laboratory workplace for a complex semiautomatic blood analysis, operated by a qualified lab technician, or a workplace of a stock trader in a bank.

We can easily see that we have to select a different guiding metaphor, because the characteristics of the expert workplace don't apply. What still remains is the basic concept of an electronic workplace, but with different types of tasks and a different usage model. We have to differentiate with regard to the range of different tasks and the support needed to meet the requirements. We also have to consider the amount of computer knowledge and software skills required at this workplace. Accordingly, we identify the following characteristics:

1. *Amount of situational flexibility and repetitive activities:* Low flexibility; few tasks with highly repetitive character.
2. *Type and amount of workplace equipment:* Few materials; few otpimized special tools or automatons.
3. *Amount of required domain skills:* Medium to high, that is, the system does not have to support domain guidance for the user.
4. *Amount of IT skills required:* Low to medium, that is the system should focus on self-explanatory features and well-built process control.

We can see that the functional workplace is much more a means to an end than the expert workplace. X-ray physicians, lab technicians, or stock traders don't want to deal with the details of our application software. The most important thing they expect from a software system is to get support in their tasks, such as analyzing X-ray images or blood samples, or stock trading information at minimum expense.

We can conclude clear requirements for our software design. The few tools required should be highly specialized. We will probably also need a control automaton to cover all standard situations. In special cases, this control automaton could be disabled so that the user can manually intervene. Usually, the workplace will be designed more like a control panel than a flexbile desktop.

3.6.4 T&M Design: The Back-Office Workplace Type

This is another very common workplace type, particularly in offices and service companies. The underlying guiding metaphor tells us that this is a workplace for employees who are expected to complete a manageable number of routine tasks without special skills.

One example would be the large number of different back offices in banks or insurance companies, where clerks compile customer files, edit forms, and do routine domain tasks.

This workplace can be modeled as an electronic desktop. The following characteristics distinguish this workplace type from the other types:

1. *Amount of situational flexibility and repetitive activities*: Limited flexibility within defined routine tasks; average amount of repetitive work in a manageable number of activities.
2. *Type and amount of workplace equipment*: Few materials and simple tools, including automatons.
3. *Amount of required domain skills*: Rather low, that is, the software system should provide a high amount of domain user guidance.
4. *Amount of IT skills required*: Low to medium, that is, the system should focus on self-explanatory features and well-built process control.

Obviously, the back office workplace should be easy to handle. It seldom dictates the exact steps and can be optimized only to a limited extent. As soon as the employees see that there are nonstandard processes that they cannot complete independently based on their skills and authorizations, they should be offered a way to delegate their 'semi-finished' work to others, for example, using normal office communication tools.

Again, the above characteristics give us an idea about the requirements for our software design: The tools are adapted to the domain skills of the employees. These tools are standard, but small automatons will often help check the work results or subtasks. The software system has to support cooperative work. Here, checklists and routing slips are valuable means for supporting and standardizing workflows.

3.6.5 T&M Design: The Electronic Commerce Frontend Workplace Type

The last workplace type introduced in this section, the *electronic commerce frontend* workplace, is based on a completely different guiding metaphor. It is most commonly found in electronic banking applications. First of all, an electronic commerce frontend is not a workplace in the traditional sense, because it is normally only occasionally used by customers for (banking) services. Such a workplace provides services that are exactly tailored to customers by the organization, such as money transfers and simple investment transactions.

Our guiding metaphor for this workplace type is an application environment for the customers of an organization who do not have to have special skills to be able to use these services.

Depending on the available frontend technology, we have to think what metaphor would be suitable for this application system. This can range from a room concept (i.e., the service providing organization as a building), to an electronic desktop with a few simple tools and materials, to an Internet browser interface based on forms. We obtain the following characteristics for the electronic commerce frontend workplace type:

1. *Amount of situational flexibility and repetitive activities:* The full range from high flexibility to a few well-defined routine tasks; tailored for occasional users.
2. *Type and amount of workplace equipment:* Few materials and simple tools, including automatons; mostly requiring comfortable presentation.
3. *Amount of required domain skills:* None to average, that is, the software system should offer an easily adaptable user interface and high domain user guidance.
4. *Amount of IT skills required:* None to average, that is, the software system should be self-explanatory and have a well-structured process control.

When we look at an electronic banking frontend as one of the typical examples for this workplace type, we can see that, in contrast to the bank account manager's workplace, the individual materials are not freely accessible. They are allocated to a service, and that service is presented in such a way that the customer does not need to know what materials are required for the service. An underlying control automaton will assist the customer in obtaining the desired information or services.

Though users of this workplace type are responsible for what they expect and accept from the service-offering company, they are always customers who do not normally want to deal with the details of the system they use, but instead obtain information or use a service quickly. In addition, we cannot assume that these customers will acquire specialized skills to be able to use our system. This means that the service-providing end has to ensure high quality of the software system and particularly of its user interface.

3.7 REFERENCES

R. Budde, H. Züllighoven: "Software Tools in a Programming Workshop." In Ch. Floyd, H. Züllighoven, R. Budde, R. Keil-Slawik (eds.): *Software Development and Reality Construction*. Berlin, Heidelberg: Springer-Verlag, 1992.

Here are the basics of the Tools & Materials approach.

S. Cotter, M. Potel: *Inside Taligent Technology*. Reading, Mass.: Addison-Wesley, 1995.

This work describes the "People, Places, and Things" concept, which has interesting metaphors for application design.

C. Floyd, W.-M. Mehl, F. M. Reisin, G. Schmidt, G. Wolf: "Out of Scandinavia: Alternative Approaches to Software Design and System Development." *Human-Computer Interaction*, 1989, Vol. 4, No. 4, Hillsdale, New Jersey, England: Lawrence Erlbaum, pp. 253–350.

Basic article about alternative approaches to software design.

S. Maaß, H. Oberquelle: "Perspectives and Metaphors for Human-Computer-Interaction." In C. Floyd, H. Züllighoven, R. Budde, R. Keil-Slawik (eds.): *Software Development and Reality Construction*. Berlin, Heidelberg: Springer-Verlag, 1992, pp. 233–251.

This article considers various guiding and design metaphors for software development.

M. Heidegger: *Being and Time*. Blackwell Publishers, 2000.

L. Mumford: *Myth of the Machine: Techniques and Human Development*. Harcourt Brace, 1983.

This classical contribution in the area of the sociology of technology also discusses the meaning of containers for human culture.

B. Shneiderman: "Direct manipulation: A step beyond programming languages." *IEEE Computer*, 1983, Vol. 16, No. 8, pp. 57–69.

This is an important article about direct manipulation as a design metaphor.

Taligent Inc.: *Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++*. Cupertino, Calif., U.S.A.: Taligent Press, 1994.

Taligent famous approach to designing application frameworks.

R. J. Wirfs-Brock, B. Wilkerson, L. Wiener: *Designing Object-Oriented Software*. New York, London: Prentice-Hall, 1990.

This book about object-oriented design is still worth reading; it explains the guiding metaphor and object worlds.

This page intentionally left *blank*

4

Patterns, Frameworks, and Components

4.1 INTRODUCTION

Chapter 2 introduced the basics of object-oriented programming, describing the fundamentals at the programming level. Now we move on to the next level of object technology. This chapter discusses the concepts and terms relating to patterns, frameworks, and components. These topics have gained increasing importance for software development over the last few years. Today they form the advanced handcraft of every developer who wants to build high-quality object-oriented software.

Patterns objectify the experience of software engineers in a structured and easy to access way. Patterns, being more abstract than program code, show us the idea behind a good solution for a recurring construction problem. But most importantly, they provide us with a language to talk about the different aspects of the design and construction of large software systems. In this chapter we break down the concept of patterns into design patterns, conceptual patterns, and programming patterns. Next, we describe what pattern collections are and what they are used for.

Frameworks, now sometimes called platforms, are prefabricated software structures that usually implement design patterns. They realize essential structural and dynamic parts of an application and thus enable the reuse of concepts and software on a high level. We will explain the difference between concept frameworks, application frameworks, and black-box and white-box frameworks. We will also learn what pieces are required to connect frameworks.

Components are software building blocks that can speed up the development process considerably while increasing the quality of the final product. As we will explain components, as we use them in our T&M approach, should be combined with framework technology.

This chapter includes important terms and concepts necessary to understand our approach. First we relate these technical topics to the domain issues of application software development. By the end of this discussion the reader should understand how all these terms and concepts fit together within our application-oriented approach.

4.2 BACKGROUND: PATTERNS, FRAMEWORKS, AND COMPONENTS

Patterns, frameworks, and components are closely related, because they are all integral parts of the second-generation of object orientation. The first generation can be thought of as the period where "simple" object-oriented programming gradually evolved into a comprehensive approach including all phases, from analysis to design to construction. The second generation is characterized by a transition from purely object-oriented programming and class libraries to patterns, frameworks, and higher-level design and construction units, such as components. Patterns have been widely accepted to describe concepts and design decisions behind frameworks and components.

This chapter is important for better understanding the T&M approach, because the following sections describe and discuss patterns, frameworks, and components of interest in our T&M architecture.

4.3 PATTERNS

Patterns became a hot issue in object-oriented software development when Gamma et al. introduced their seminal book. We took an active part in this discussion from the very beginning, because we consider patterns to be an important concept for software architectures. This is the reason why we decided to present the constructive part of this book, that is, the description of the important elements of a T&M model architecture, in the form of patterns (see Chapters 7 and 8).

It should be understood that this chapter is not intended to provide a comprehensive discussion of patterns. Instead, what we should take home from this chapter is a sound understanding of the set of essential terms and concepts of our T&M design.

4.3.1 Definition: Patterns

Let's start with a general definition.

A *pattern* in the generic sense is an abstraction of a concrete form, occurring repeatedly in certain nonarbitrary contexts.

The definition that software developers are familiar with relates to software design. Christopher Alexander, a California architect, whose seminal work is appreciated by many software engineers, describes patterns as follows:

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.[1]

Alexander also explains that a pattern represents a relation between different forces recurring repeatedly in certain contexts, and in a configuration that binds these forces. In addition, he sees a pattern as a rule describing how this configuration can be created.

1. Quoted after Gamma et al., p. 2.

4.3.2 The Characteristics of a Pattern

Patterns have forms and purposes.

- The *form* of a pattern consists in a number of identifiable and distinguishable components and their connections.
- The *purpose* of a pattern is to create, identify, and compare instances of that pattern. The pattern supplies a term or notion allowing us to discuss a solution to a problem.
- A pattern occurs only in certain *contexts* that determine the underlying forces, which produce the pattern in its specific form.
- Although the form of a pattern is *finite*, this does not mean that the number of instances of that pattern is finite. The context of a pattern is also potentially unlimited.
- A pattern can be understood only on the background of *experience* and reflection within the application domain. A pattern should be presented in the terms and concepts of the intended application domain.

We define the form of a pattern by its representing elements, their relationships, and their interactions. This means that a pattern has both structural and dynamic properties.

The elements of a pattern are not necessarily software components. They can also be other technical and nontechnical objects. This section explains the elements that different kinds of patterns are made of.

The EMS example

Let's return to our EMS example. When we analyze the application domain using the T&M approach, we will more or less consciously divide all objects, like room plans and devices, into tools and materials. When we look at a room plan as a material and transfer it into an appropriate piece of software, we are using a metaphor. This leads to our concept of design metaphors (see Section 3.5). Using metaphors to analyze the elements of an application domain opens up new insights. The dominant relationship between most elements is the fact that we use tools to handle materials. In this sense, we can think of the interaction of tools and materials as a fundamental pattern. Thus, using metaphors has led us to a general solution of a recurring problem: People use tools to work on materials in order to cope with their work tasks. We have formulated this solution as a so-called conceptual pattern (see Section 4.3.4).

A pattern, in general, is a concept based on experience. Reflecting about our experience, we can recognize recurring patterns. Once we have understood these patterns, then they will guide our perception. We then use a pattern to recognize its occurrences or instances. More strictly speaking, patterns exist only in the form of their instances. But a pattern can also be used constructively to create an instance of it.

A pattern guides our perception; it predefines the "structures" we will realize in our environment. When we name a pattern, we introduce a notion that characterizes both the problem related to the pattern and a solution to this problem. Patterns have a language-building character, which means that they are important both in the real world and in our software worlds.

The EMS example

Let's go back to our EMS example. A conceptual pattern, say the *interrelation of tools and materials*, helps us to identify the concrete occurrences of the pattern. On that basis, we think of a pen as a tool that we can use to change information in a room plan as the underlying material. We could discuss with the device manager which tools and materials are relevant elements of the work and of the future system. We can see that patterns contribute to understanding an application domain.

We can also use the above example to show the constructive or generative function of patterns. This means that we could use our conceptual pattern, *interrelation of tools and materials*, to support the design. We thus design a specialized graphic editor, let's call it Device Organizer, as a tool that will be used to edit an electronic room plan. Continuing along the line of this domain-specific conceptual design, we will then need more design patterns to build a sound software architecture, where tools and materials are again the most important components on the object and class levels.

Context of a pattern

We know from Section 4.3.1 that a pattern is a specific form in a specific context. We could alternatively say that a pattern always appears on a certain background. To better understand this idea, visualize the drawings of M. C. Escher or picture a puzzle for a moment. Both have this in common: we first have to understand the background before we can identify the things in the foreground. This means that a pattern always has a background or context. A context defines the pattern; a context produces a pattern, while a pattern matches only a specific context. If we change the context, then the pattern will change too, and once the pattern has changed, the context will change.

The EMS example

In our EMS example, if we identify the room plan that is currently handled manually by the use of a pen, then we can relate the room plan (material) and the pen (tool) to a certain type of work context as the background. We assume that the device manager understands her trade, that is, that she is able to independently procure a new device, update existing devices whenever she thinks this is necessary, select the tools she thinks are appropriate, and so on. This tells us how we should understand the conceptual pattern, *interrelationship of tools and materials*, on the background of a specific type of human work.

Elements of a pattern

The description of a pattern consists of a finite number of elements and relationships. However, concrete variants of a pattern can have an arbitrary number of elements, which are recursively defined and created upon demand.

For example, the work of Gamma et al. describes a pattern called *chain of responsibility* with *client, handler*, and *successor* as its elements. This finite pattern can be instantiated into a recursive chain of handler-successor pairs in an arbitrary length.

Understandable pattern descriptions

Patterns grow from experience and can be applied only by experienced people. The author of a pattern description will want to include what he or she considers relevant and meaningful in describing the context. For a newcomer to this discipline, this can mean that this person won't understand the problem to which that pattern is supposed to provide a solution. In addition, when using a pattern, there should always be sufficient experience to be able to develop the correct variant for the context at hand. There is no question, however, that all experiences distilled into a pattern are useful for everybody, including newcomers.

The *interrelation of tools and materials* pattern assumes that its users are familiar with the basics of developing interactive software for workplace systems. Otherwise, they will not be able to implement the interaction between tools and materials as interactive software. These prerequisites given the pattern helps you to identify relevant elements of the application domain and introduce them to your software design.

Application orientation is extremely important, even if we do not want to restrict the use of patterns to software development. Obviously, the prime use of patterns is to facilitate the pattern author's work. The true potential of a pattern shows when it is accepted and reused by others. This proves that it is indeed a generalized form to solve a recurring problem in a specific context.

Intuitive pattern names

An important thing to remember is that we should name our patterns, and that these names should be descriptive enough to ensure that everyone participating in a project will understand what we tried to introduce in the design.

Conceptual patterns, like the *interrelation of tools and materials* pattern, are normally named with plain language. This will ensure that these patterns can be understood in many different application domains. In general, in the application-oriented T&M approach all conceptual patterns should have names that are meaningful in the respective domain language.

4.3.3 A Taxonomy of T&M Patterns

Chapter 6 will describe different models that play an important role in our software development approach. We will distinguish between the application-domain model, the design model, and the implementation model. It has also proven useful to distinguish the different types of patterns, with their respective purposes based on these models. We distinguish patterns as follows:

Conceptual patterns **describe elements and concepts pertaining to the application domain. They are based on corresponding design metaphors. Although design metaphors provide a basis for discussion among all participating groups, conceptual patterns help the developers in particular in modeling their application domain.**

Design patterns **describe the software system on software levels. They are micro architectures for software construction. The most prominent examples are listed in the book of Gamma et al.**

Programming patterns **are the most concrete and technical patterns. They describe constructs or idioms on the source code level, for example:**

```
for (int i = 0; i < max; i++) {. . .}
```

4.3.4 Conceptual Patterns

A model of an application domain (see Chapter 6) should be designed so that all participants can understand it. Patterns can increase the readability of a model, if they use concepts and metaphors specific to the application domain. We call such patterns *conceptual patterns*, as already defined. They are also used in the usage model.

Conceptual patterns reach beyond design metaphors, because they are intended for developers who support individual activities during the application domain modeling process. They also guide us during the analysis of the application domain. Conceptual patterns help us to see better which objects of an application domain—and which relationships between these objects—are relevant for the underlying model. We learn about the relevant tasks involved in our application system and make this explicit in our model. At the same time, we normally select those elements and patterns for our model that we can use in our domain-specific and software-specific design of the future application system. Conceptual patterns are a guideline to perceive, interpret, and change the application domain.

Once again, this shows that, for a developer, the analysis and modeling of an application domain is always connected to the design of the future system.

In our approach, however, this is not a task only for the developers. The author-critic cycles of our evolutionary approach require all participants to actively take part in the design process. For this purpose, we have to ensure that the conceptual patterns are closely related to the design metaphors.

The EMS example

The conceptual pattern *interrelation of tools and materials*, introduced in Section 7.3 is a basic pattern for application modeling. When writing scenarios (see Section 13.1) we use instances of this pattern such as room plan and pen, to describe how tasks are completed by use of tools and materials. When designing our future system, we have to ensure that this specific type of interaction is not lost in the system visions, which means that a tool, such as the device organizer, can be used to handle a material, such as an electronic room plan, in a similar way.

Considering that conceptual patterns, like all other patterns, are not universal, we have to allocate them to a specific context. The author of a conceptual pattern usually tries to avoid the extremes of being too general or too special.

We think that a pattern like *actively cooperating objects* would be too general to be a useful conceptual pattern. To turn *device organizer and room plan* into a conceptual pattern wouldn't make much sense either, because we only use it in our example. The conceptual pattern *interrelation of tools and materials*, however, has proven to have a suitable abstraction level for office systems. In fact, it is similar to other conceptual patterns, such as *agent* or *medium*. On the other hand, an attempt to transfer the *interrelation of tools and materials* pattern to real-time systems with their inherently rigid temporal requirements would not be very successful.

4.3.5 Design Patterns

The design model occupies a central position between the application domain model and the implementation of a software system (see Chapter 6). It represents our understanding of the application domain and already takes the restrictions and formal requirements of a software system into account.

- Design patterns connect the domain statements of the conceptual patterns with the constructive requirements of an object-oriented design.
- They help you analyze and reorganize existing software designs.
- Design patterns help you to understand software architectures (e.g., class libraries, frameworks), as long as the architectures are documented using well-known design patterns.
- You can think of design patterns as "micro architectures" that experienced developers can take and map onto an implementation, similarly to components in a larger software architecture. This means that you can reuse design solutions rather than code.
- Design patterns represent the elements of a language that we use to think and communicate about software architectures.
- And finally, design patterns support us in refining the design model and increase the quality of our designs (e.g., reusability and scalability).

We use *design patterns* to describe and to understand our design model. Design patterns define the structure and dynamics of the element that a design model contains, and they help us better understand the interaction and responsibility of each of these elements. From the view point of software construction, they take the conceptual and structural requirements of the conceptual patterns an important step towards executable software code.

In our approach, it is essential that the application domain model and the design model for our application contain as few semantic differences as possible. To ensure this, the design patterns should be closely related to the conceptual patterns and the design metaphors upon which they are based. The elements and relationships defined by design patterns should then allow easy mapping to the implementation, without losing their structure.

Design patterns and frameworks

Design patterns are closely related to *frameworks*. Frameworks contain and instantiate design patterns. In an ideal situation, frameworks are developed on the basis of design patterns. At a minimum, the documentation of a framework should specify the design patterns that were used in building a framework.

The literature and practice of software development deal with design patterns in totally different ways, that is, by allocating them to different levels, ranging from problems of large-scale software architectures to so-called micro architectures.

Example

The conceptual pattern *interrelation of tools and materials* describes the basic design elements, tools and materials, in the context of expert work. With a design pattern called *coupling tools and materials*, we take the *interrelation of tools and materials* pattern a step further: we describe the explicit interface of two design units (a tool and a material) and specify that they should be loosely coupled. For this purpose, we recommend specific object-oriented constructions. Finally, we use our framework-based model architecture to show where the general interaction between tools and materials can be encapsulated within a framework.

4.3.6 Programming Patterns

In addition to the usage and design models, we define the implementation model as our third software development model (see Chapter 6). This model is normally not separate from the program code but "embodied" in the program sources. The programming language we select supplies the notations for this model. To better describe and understand this model, we use *programming patterns*. In other words, whenever we write program designs and the actual implementations we use programming patterns.

Programming patterns

Programming patterns differ, depending on the programming language and culture. It is worth noting that even on this relatively low technical level, there are such things as programming cultures and styles. Programming patterns are often referred to by different names; Coplien, for example, calls them *idioms*. A detailed discussion of programming patterns would go beyond the scope of this book. However, we will give a rough idea of what programming patterns are all about.

The following small example for a programming pattern is a loop structure known from C; you will find a similar syntax in any Java, C, or C++ program.

```
for (int i = 0; i < max; i++) {...}
```

Above the programming language level, you will find the classification of class interfaces based on the programming pattern *instruction–function–predicate*, which we proposed in Section 2.1.8, following Meyer's work. This pattern fits the design-by-contract model proposed by Meyer, which can be thought of as a design pattern.

4.3.7 T&M Design: Design Patterns

In the context of this book, T&M design patterns consist of a general descriptive part and one or several construction parts.

- The general *descriptive part* of a T&M design pattern defines the elements available for software design regardless of a specific programming language. For this purpose, it uses object-oriented concepts (e.g., use, role, components, generalization, specialization).
- The *construction parts* of a T&M design pattern describe the elements for your software design. The form is oriented to the concepts of the selected programming language (e.g., procedures, objects, classes, inheritance).

This breakdown of design patterns is in line with the general T&M approach. Although we generally orient to specific application domain, that is, the financial and service sectors, we do not orient ourselves to a specific programming model. Nevertheless, we need a guideline for constructive solutions that take the particularities of specific programming models into account.

Generic description part of a design pattern

Thus, each design pattern is described on a level independent of any programming language. On this level, T&M design patterns show the basic constructive ideas behind concepts, for example, the tools, materials, and automatons within a work environment.

Example

Section 8.3 introduces the design pattern *aspect*. The general desciptive part of this pattern builds a relationship between a tool, a material, and aspects. Next, this pattern is concretized by the use of four different construction parts. This example shows how we can implement alternative aspects in different languages.

The construction parts of a design pattern

A construction part uses the elements and relationships of the selected programming model to better explain the structure and dynamics of a design. A construction part is always the concrete part of a T&M design pattern. It shows the implementation alternatives for a design pattern. In this book, we use construction part examples for programming models in C++, Java, or Smalltalk. In a real-world project, design patterns are often described directly in the form of construction parts, particularly when the programming language basis is inflexible.

In practice, it is also sometimes necessary to develop construction parts for languages other than object-oriented languages, as we did in several projects. In such cases, we always had to clarify whether and what object-oriented design concepts should be retained for the implementation level. We intentionally don't use such construction parts in this book, because this issue could easily fill a separate book.

4.3.8 T&M Design: Models, Metaphors, and Patterns

Obviously, there is a relation between design metaphors, patterns, and models in software development based on the T&M approach. We have defined a design metaphor as a visual and objective idea that fits and concretizes a guiding metaphor from both the domain and software views. A design metaphor helps all people involved in the design of a software system to find a common basis of perception and understanding. Design metaphors are based on everyday language, making the application domain easier to understand and facilitating the future handling and functionality of the application system under development.

In contrast, patterns are primarily relevant for the activities of the developer. As early as the domain modeling phase, developers should try to concretize design metaphors, because they need to know what things in the application domain are relevant in view of the future system and to what detail they should be described. Conceptual patterns can help them find an answer to this question.

In the T&M approach, it is important to understand that design metaphors are related to technical interpretations in the form of *construction guidelines*. Conceptual patterns bridge the gap between the domain-related view of design metaphors and the construction perspective of design patterns. Design patterns describe the basic elements of the software design. These elements are then instantiated by the use of construction parts and programming patterns.

Our guideline for handling patterns is this: *Analyze the application domain and work out the application domain model by the use of design metaphors and conceptual patterns; then create the software-specific design by the use of design patterns; and finally, use the programming patterns to implement these design patterns*.

Example

Let's have another look at our tools and materials example. Tools and materials are common-language design metaphors that should be familiar to every developer or user. The conceptual pattern on this level is the *interrelation of tools and materials* pattern, as described in Section 7.3 and shown in Figure 4.1. Within an office work context and expert activities, it appears meaningful to view the relevant tasks on hand in terms of using tools to manipulate materials. This is expressed by the *conceptual pattern*.

For the software design we use the design pattern *aspects*. We refine the concept pattern by first introducing two design components, *tools* and *materials*. Their interaction

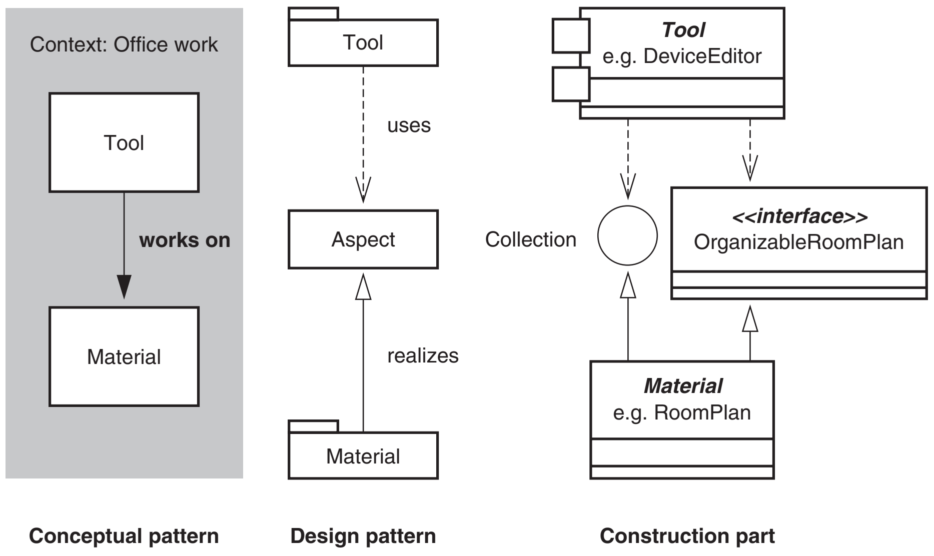


FIGURE 4.1
Example of conceptual pattern, design pattern, and construction part.

and the way in which one element fits to the others is expressed by *aspects*. An *aspect* tells us the properties that a material should have so that a tool can be used to manipulate it. On the other hand, a tool knows all of its materials only under a specific aspect. This relationship is specified in the general part of our design pattern. In our example, we could think of a naive instantiation of the pattern in that a device organizer knows only the aspect `OrganizableRoomPlan` but not the concrete material `RoomPlan`.

Finally, we have to implement our design on the constructive level. More specifically, we have to define how aspects can be implemented by the use of features provided by the programming language. This is described by a construction part. The example in Figure 4.1 is a solution that uses aspects as interfaces.

4.3.9 Background: Pattern Form

Hoare once said that each abstraction needs a representation to give it the necessary form. This statement also applies to patterns. In general, the literature agrees that patterns should also be described in a structured pattern form.

If we compare the large number of different pattern descriptions, we can see that whether the pattern description you select is suitable depends largely on the intended purpose.

Following Alexander's classic definition, a pattern description consists of three segments: *problem, context*, and *solution*. This form proves useful when you use a pattern to work out solutions to a given problem.

The *problem* segment briefly describes the problem to be solved. The *context* segment identifies typical situations where the problem occurs, as well as the forces and conditions hindering a possible solution. And finally, the *solution* segment tells you how to restrict or eliminate these forces in this particular context. One major characteristic of this form is that it can be used for generic problem-solving tasks. The solution is normally described so that the pattern for a specific problem can be easily instantiated. Other pattern descriptions are primarily of a descriptive nature, that is, they focus on the structure and dynamics of a construction pattern.

4.3.10 T&M Design: Pattern Form

The description for patterns used in this book is loosely based on the form proposed by Gamma et al. We structure pattern descriptions with the following segments, although not every pattern has to include all segments:

- A *name* for the pattern.
- The *problem* to be solved by the use of that pattern.
- The *context* around the problem to be solved, and its counteracting forces.
- A *solution*, representing the elements of the pattern in their roles and how they interact, and the way they interact.
- An *example* to better illustrate the issue; we use the EMS example throughout the book as well as small examples from our projects.
- A *discussion* of the different construction approaches.
- The positive and negative *consequences* of using patterns.

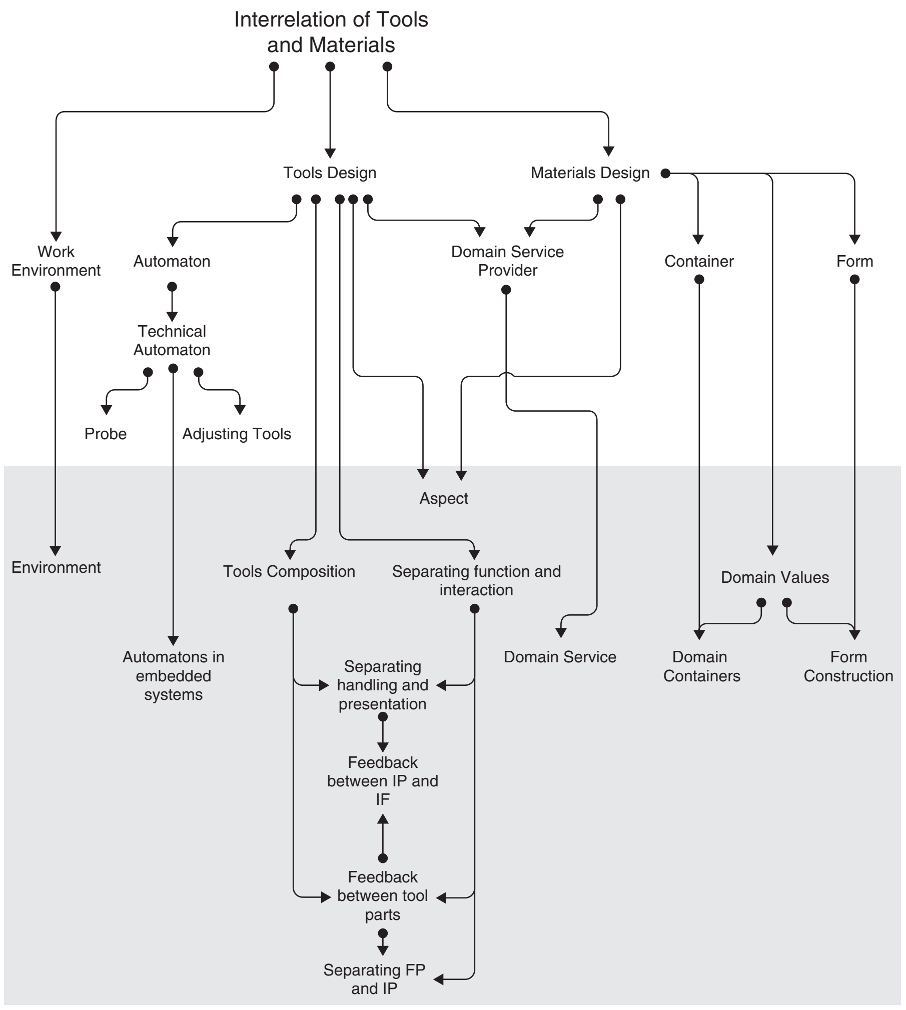


FIGURE 4.2
The T&M pattern Roadmap.

4.3.11 Pattern Collections

Patterns like the ones described in this section do not exist in isolation. They are connected within their common application domain, and they follow the mesh of relations given by the selected design metaphors. In the T&M approach, patterns are linked, that is, one pattern includes another one or interacts with others.

Example

If patterns include other patterns, they can be arranged as *pattern collections*. Our pattern collection is hierarchical, which means that the context of one pattern includes the context of another pattern. Understanding the contextual pattern helps us to understand the pattern that it includes. In this sense, the more comprehensive pattern precedes the included pattern logically and in its description. In the T&M approach this also applies to a pattern type; for example conceptual patterns precede the related design patterns, and domain-specific patterns include the corresponding software-specific patterns. In short, the order is conceptual patterns before design patterns before programming patterns.

The conceptual pattern *interrelation of tools and materials* describes the context in which tools and materials interact. This basis is necessary for us to meaningfully discuss the related design pattern *aspect*. This, in turn, forms the conceptual basis for understanding design patterns as *tool construction* and then the pattern *separation of interaction and function*. This latter pattern precedes the design pattern *feedback between tool parts*. In their construction parts, the respective technical solutions use well-known patterns, for example the *observer*. Figure 4.2 shows how all of the T&M patterns described in Chapters 7 and 8 relate. A diagram like this is often called a pattern roadmap as it shows potential ways through a collection of related patterns.

4.4 FRAMEWORKS

This section begins with some historical background about frameworks and the concept of class libraries, which is important for understanding frameworks. Later sections explain the difference between and purpose of application frameworks and black-box and white-box frameworks. Finally, as a practical example, we describe a layered architecture with the JWAM framework.

4.4.1 Background: Class Libraries

Re-use was an important technique before the first object-oriented programming languages were introduced. In traditional procedural languages, you can use construction units, such as modules, only in the way they were provided. If you want to use the concept in a slightly different form, you have to copy and modify the program text. The two object-oriented constructs, class and inheritance, allow you to reuse a class in a subclass. More specifically, you take the class and redefine it for your specific use, without changing the original class. Meyer calls this the *open-closed principle* of object orientation (see Section 2.2.3).

The open-closed principle made class libraries an integral part of object-oriented programming languages.

A *class library* groups a set of individually usable classes.

Class libraries are used either directly or by subclassing.

A class library does not define the control flow of an application system.

Unlike module libraries, class libraries are not unordered collections of ready-to-use elements; rather, they form a flexible hierarchy consisting of extensible building blocks and concepts. This results in two things: first, the given programming language is extended on type level, and second, class libraries form the basis for new user-defined type and class hierarchies.

In general, you can use each class from a class library, regardless of the other classes included in that library. For this reason, objects created by classes of a library do not automatically cooperate with other objects instantiated from the same library. The interaction between these objects, or the control flow, is defined in the program code of the application system.

A good example of this is the container library, which expands the classic data-structure library. In container libraries, container classes can be utilized on a basic level by instantiating and using container objects.

But recent container classes are more than a group of passive data structures based on access operations. These container classes add explicit markers, cursors, or so-called robust iterators to data structures, so that containers can be easily searched and edited. That gives us not only a container class, but an interaction between a container and one or more iterators. Also, specific requirements for the elements maintained in a container are formulated in an abstract superclass. This interaction can generally be expressed in class structures and the objects created from these classes.

4.4.2 Definition: Frameworks

The preceding example of a container shows us an interrelated functionality that is reflected in the structure and dynamics of the elements involved, for example, in the construction's control flow. It means that such container collections are a transition to what we normally call frameworks.

A *framework* is an architecture of class hierarchies that offers generic solutions to similar problems within a specific context.

With frameworks you reuse the entire construction of interacting elements rather than single classes. This means that a framework defines the control flow for your application.

A framework can be developed into a specific application by specializing selected classes or creating predefined parameter objects.

How frameworks and patterns relate

This definition indicates how frameworks and patterns relate. A pattern is an abstraction of a concrete form that recurs in certain nonarbitrary contexts. A framework is more concrete than a pattern, as it is given not only in verbose form but as a "semi-finished" implementation of class structures.

On the one hand, the class structure of a framework identifies the architecture and control flow of your application system. On the other hand, it has specific points (Pree calls them "hot spots") where you can attach your specific solution.

4.4.3 Application Frameworks

Frameworks can be used to solve problems in all kinds of domains. A framework that defines the design and relevant constructions for a complete application system is called an application framework.

An *application framework* is intended for developing application software within a given application domain.

It specifies the software architecture and the relevant domain-specific abstractions in the form of a generic solution.

From the technical view, an application framework is normally structured into several subframeworks.

Currently, technical frameworks intended to implement graphical user interfaces or a persistence mechanism have been used most frequently. Application frameworks are much more complex, because they have to define both an abstraction of a concrete application domain and a suitable software architecture. For this reason, an application framework is normally structured into several frameworks, each representing a separate aspect of the application. This book describes general design and architectural principles for application frameworks within the T&M approach and by using the Java JWAM framework as an example (see Section 8.16).

4.4.4 Black-Box and White-Box Frameworks

Frameworks are often divided into black-box and white-box frameworks on the basis of their use.

A *black-box framework* is normally used when you build a complete application by configuration rather than by additional programming. For this purpose, objects are created from predefined classes, which are normally parameterized by specific configuration objects, so that you can adapt them to application-specific requirements.

In contrast, *white-box frameworks* are designed so that you can derive subclasses from specific classes. The objects of these subclasses can be used directly in application systems or in other frameworks. They can also be used to configure objects, which were also created by the classes of a framework.

Using black-box frameworks

This means that, when using a black-box framework to develop your application system, you create objects from framework classes available especially for this purpose. In many cases, you can use other objects to configure these objects. Such configuration objects are also instantiated from classes of the same or another black-box framework.

Let's use a simple example to see how this works. Assume that we use the *interest calculator* shown in Figure 4.3. If a software developer wants to create a savings account based on decursive interest, then he could use the `account` *and* `interest calculator` frameworks. First, he would instantiate an object from the `SavingsBook` class; then he would pass an object from the class `DecursiveInterestCalculator` to the former object. This means that our developer sets the interest calculator used for `SavingsBook` to decursive interest earning. Calling `calculateInterest` of `SavingsBook` would call the operation `calculate` when the program reaches the `DecursiveInterestCalculator` interest strategy.

Using white-box frameworks

Although you don't need to know the internals to use black-box frameworks, white-box frameworks can only be used when you know the internal structure of a framework, in particular, the interaction of the objects modeled in their respective classes. White-box frameworks are normally used to develop applications so that specific classes of the framework are specialized in subclasses that thus connect the framework to the other classes of your application system.

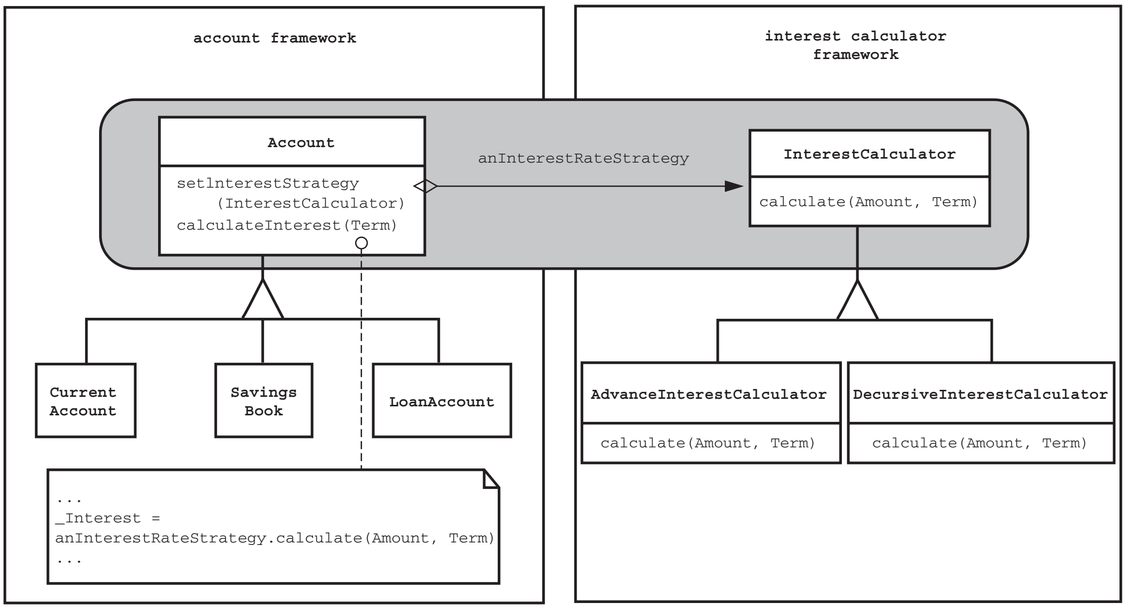


FIGURE 4.3 Example showing an interest rate calculator.

Objects created from the subclasses of a white-box framework can be used in two different ways:

1. You can use the objects created directly in your application system. For example, the `account` framework could be designed so that it implements the `account` class and the abstract interaction with the objects of the `interestCalculator` class. The concrete subclasses of `account` would not be included in the framework, so that you would have to build them by subclassing. A subclass would then inherit the interaction with the `interestCalculator` from the framework, so that the subclass doesn't have to implement this interaction itself.
2. You can use the objects of your subclasses for configuration purposes. For example, the framework could supply concrete account classes (`checkingAccount`, `savingsAccount`, etc.), but no interest-producing strategies. In a real-world application system, you would have to build a suitable subclass from the abstract class `interestCalculator`, where the objects of this subclass would then be used to configure specific `account` objects.

Comparing black-box and white-box frameworks

We can see in these examples that black-box frameworks are easy to understand and use, while white-box frameworks are somewhat more complex but more flexible. Experience from real-world projects has shown that black-box frameworks often evolve from white-box frameworks by successively adding standard solutions. This means that software developers do not use frameworks only to build application systems, but also as a basis for developing other more specialized frameworks.

Note that there is no rigid separation between black-box and white-box frameworks. In fact, frameworks are normally grouped depending on the way they are used in a software project. Though frameworks designed for white-box use are mainly used as such, they can also contain standard solutions similar to black-box frameworks. In turn, most black-box frameworks can also be used as white-box frameworks.

4.4.5 Connectors between Frameworks

We know from Section 4.3.5 that design patterns can be used to describe complex interactions between elements. For example, design patterns can describe responsibilities and the cooperation of a number of classes that solve a specific design problem. If the classes of a design pattern are distributed over several frameworks that we want to connect, then this design pattern describes the protocol we should use to allow interaction between these frameworks.

Framework *Connectors:* Interacting frameworks can be coupled or connected by simply calling the interface of a corresponding framework class. But often the interaction between frameworks is more complex, so that it should be described on the interface level by a design pattern. We call these design patterns framework connectors.

To give an example of framework coupling, let's have another look at our `account` and `interestCalculator` frameworks (see Figure 4.4). The `interestCalculator` is a black-box framework, and used to build the `account` framework. The `interestCalculator` classes can be used as *strategies* in the `account` classes. To simplify the example, we will create the `strategy` objects that we need in the concrete `account` classes. Now, if you look at the strategy pattern as our connector, you can see which classes of the `account` and `interestCalculator` frameworks have what responsibilities within the `strategy` pattern. The `account` class is responsible for the *context*, while the `interestCalculator` class is responsible for the *strategy*.

Note that the construction in the figure does not define the specific interest-producing strategy from the `interestCalculator` framework that will be used to implement an `account` class within the `account` framework. This information has no impact on the architecture; it merely represents domain-specific details, which can be used later when we build the concrete `account` class.

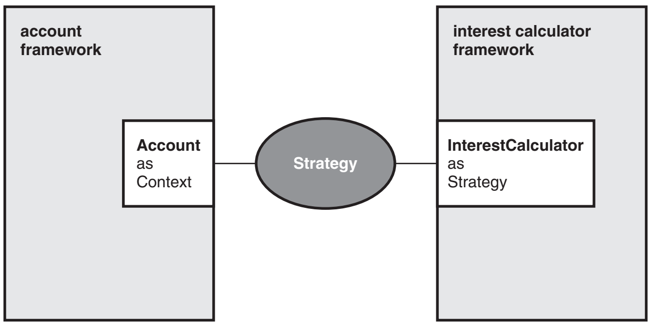


FIGURE 4.4
A strategy pattern connecting two black-box frameworks.

4.4.6 JWAM Framework: Layered Framework Architecture

Application frameworks are normally very complex. To facilitate handling and using them in software projects, such frameworks are often organized in layers. Section 9.3.5 will describe layered architectures in detail.

The JWAM example

In large-scale software systems and frameworks, a layered architecture assumes the role of a corporate organization chart, that is, it maps units with specific responsibilities and regulated communication between these units for the domain-specific tasks.

A layer organizes the software-specific components in a design and construction unit, based on domain-specific and software-specific motivations. The layers of an architecture are organized in a hierarchy. We use class libraries and frameworks as the components of a layer. In this sense, frameworks could also contain other frameworks.

To support software developed by the T&M approach, we built a generic application framework in Java called *JWAM*. This framework is generic, that is, it is not tailored to a specific application domain, but it has the basic features required to build interactive application systems. Figure 4.5 shows the layered architecture of the JWAM framework.

In the example shown in Figure 4.5, each layer contains frameworks for specific purposes.

- The *domain-specific application* layer is not part of JWAM. It is developed within an application project and includes frameworks and class libraries relating to the respective application domain.

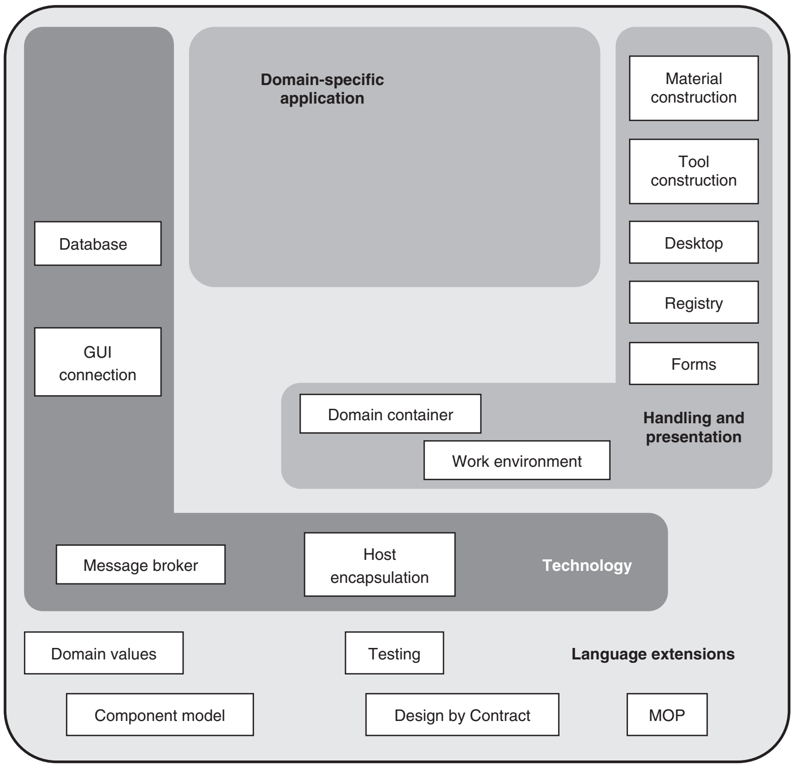


FIGURE 4.5 The JWAM framework structure.

- The *handling and presentation* layer includes frameworks that implement the *reusable elements* of interactive application software. For example, these frameworks include those that implement the control flow within a tool based on the observer pattern.
- The *technology* layer includes frameworks that encapsulate the technologies used in the way we would like to use them at higher layers. Thus underlying technologies can be exchanged more easily.
- The *language extensions* is actually not a layer. Java doesn't support some language elements that we would expect from a good object-oriented language. For this reason, we have implemented these elements in the form of frameworks, that all layers can access. This includes basic concepts such as domain values, Meyer's contract model, or a metaobject protocol.

Note that such layered architecture serves as an example; it does not have to be identical for all projects. Other application frameworks, such as IBM's San Francisco, are based on different architectures. Depending on the specific layers used by an application framework, the objective is always to keep the different dimensions of your application software independent of one another. We will address the issue of these dimensions of software construction in Section 6.9.

4.5 COMPONENTS

This section begins with some background on components, and then defines the term *component*. Next, we describe how components and frameworks relate, and finally we look at components in the T&M example. The reader will thus understand that combining components with frameworks provides the best solution for supporting the construction of large software systems.

4.5.1 Background: Software Components

Software components have been desirable elements for many developers, especially IT managers. This desire is based on the need to decompose software systems so that you have ready-to-use components for many different application domains. The literature speaks of *software ICs*.

The discussion on components has recently heated up over the idea of a booming component market where IT managers would be able to buy low-cost turnkey subsystems.

The daring among the component advocates promise that this would mean a dramatic change in application software development in the near future. They argue that trained users should select standard components, which they could then edit and adapt in a graphical editor to build complete systems. We don't think so.

Nevertheless, disregarding overly optimistic hopes and ideas, there are still a large number of arguments that may make it worthwhile to take a closer look at components. In addition, some commercial component products, for example, Microsoft's Distributed Component Object Model (DCOM) or Visual Basic Extension (VBX) (see Section 4.5.3), show that turnkey software components can indeed be useful. The introduction of IBM's Eclipse component model as an Open Source product has been another major step towards making components a technically and commercially feasible solution.

4.5.2 Definition: Components

The literature uses a large number of different component concepts. For example, here is a well-known definition proposed by Clemens Szyperski: “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” p. 34.

In contrast, the definition of components proposed by Oscar Nierstrasz and coauthors is more general, basically specifying a service that will be bound later, from the client perspective, and which has plug-compatible interfaces. Although there is a wide spectrum of different definitions of components, we can derive the following rough classification:

***Programming components*: These components are available in the source text, representing a technical or domain-specific solution, and they are suitable for reuse in a similar context. They normally become (an undistinguishable) part of the application program.**

***Implementation components*: These components are normally available in binary form. They are produced independently of one another and can be purchased and integrated into the system at compiletime.**

***Runtime components*: These components are available in binary form; they can be purchased and used directly. They are normally added at runtime.**

More recent discussions propose the following main characteristics of components:

- Components can be reused.
- Components have one or more explicit interfaces, which they use to offer coherent domain-specific services.
- Over an explicit interface, components specify the services they need from other components or from the environment that embeds them.
- Components hide their internal structure and their implementations.
- Various versions or variants of a component can be selected “late”, that is, when the component is installed or loaded, or while the application system executes.

Our approach is based on an application orientation, so our general definition of components is as follows:

A *component* represents the software solution of a technical or domain-specific problem.

For use in an application system, a component is equipped with a domain-specific interface. The component uses this interface to offer its services. When components require other components, they have to declare this over defined interfaces. Components can be interchangeably fitted into the interfaces of a framework.

Depending on the time and the way in which they are integrated into the application system, we distinguish between programming, implementation, and runtime components. We see components as coarsely granulated units or modules that can have several interfaces and classes.

4.5.3 Current Component Products

We have observed in a number of our real-world projects that components normally come in two sizes, that is, either small or very big, such as visual components used to display and select the date on a calendar sheet, or as a complete enterprise resource planning (ERP) system, such as SAP R/3.

Many component models are based on the assumption of a booming component market, but profitable markets seem to have opened up for just a few domains. Probably the most successful component market is that of graphical user interface (GUI) elements, where VBX (or OCX) of Microsoft or JavaBeans of Sun Microsystems have been among the most successful products.

For these component kits we frequently find tailored development environments allowing direct manipulation of components to build a composition graphically. For example, the event outputs of one component can be graphically linked to the event inputs of another component. Our experience has shown that this is useful only for very small applications, even when prototyping, because developers can easily lose track of the overall picture. This situation suggests that components should better be linked by the use of well-written glue code.

Enterprise JavaBeans represents a component kit based on binary compatibility but not especially designed for a general component market. The so-called EJBs are primarily used to deploy large-scale software systems and for distribution over a network. A system is structured into domain-related components without a graphical representation (see also Section 8.14) or those parts implementing specific handling and presentation. For handling and presentation, either JavaServlets, Java Server Pages, or Swing, are used, depending on the technology used.

IBM's Eclipse component model was originally designed as the basis for the Eclipse Java IDE, a flexible programming environment with plug-in technology. Since IBM decided to publish Eclipse as an Open Source product, the underlying component model was freely accessible and soon became a basis for many new components and plug-ins, mainly in the area of design and development tools. As the Eclipse component model is small and well designed, and as it can be used completely independently of the Eclipse IDE, we expect it to become quasi-standard for components and plug-ins.

In summary, we see a relevant trend for the use of components as design and construction units to build large-scale distributed systems. In addition, there is a well-established market for generic GUI components, and an emerging market for development tools and IDE plug-ins. It currently does not seem that there will be a significant market for domain-specific application components.

4.5.4 Components and Frameworks

Component and framework approaches appear to pursue the same goals, at least at first sight: extensive reusability and better structuring of application systems. From the technical stance, these two approaches are different. When you use frameworks, you have to write code. Predefined classes are used or inherited for individual application software projects. When you use components, you work by the idea of composition rather than writing code. Inheritance is not used at all, and the components are treated as black boxes. The programming part is normally minimal, that is, just enough to write the glue code required to link your components.

Frameworks could also tend to develop in this direction. On the other hand, components used for complex distributed architectures, such as EJBs (Enterprise JavaBeans), can also be thought of as a kind of white-box framework.

It seems, however, to be useful to distinguish between components and frameworks. According to our understanding, frameworks determine the overall architecture of an application, supplying the important abstract concepts of that architecture. In contrast, components encapsulate additional solutions for specific technical or domain-related purposes for the application developer to select.

We think that the component and framework technologies can complement each other well. You can use one framework as the interacting platform for a family of components, where the individual components can build on a uniform technical infrastructure of the programming language and the component model used. This includes the basic concepts and abstractions of the framework. This also means that you can define domain interfaces on a higher level than the ones typically defined with JavaBeans. The Eclipse platform works this way, and the components of the JWAM framework for the T&M Approach use this concept as well. For example, a component in the JWAM framework can rely on the existence of the concepts *tool* and *material*.

4.6 REFERENCES

C. Alexander: *The Timeless Way of Building*. New York: Oxford University Press, 1979.

C. Alexander, S. Ishikawa, M. Silverstein: *A Pattern Language*. New York: Oxford University Press, 1977.

Two classic works by C. Alexander about the use of patterns in architecture that have triggered the discussion of design patterns.

K. Beck, R. Johnson: "Patterns Generate Architectures." ECOOP'94, Lecture Notes on Computer Science 821, Conference Proceedings. Berlin, Heidelberg: Springer-Verlag, 1994, pp. 139–149.

An important contribution on patterns and frameworks.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester, N.Y.: Wiley & Sons, 1996.

A well-known book on patterns.

Communications of the ACM. Special Issue on Agents. July 1994, Vol. 37, No. 7.

This issue describes different conceptual patterns such as *agent* and *medium*.

J. Coplien: *Advanced C++: Programming Styles and Idioms*. Reading, Mass.: Addison-Wesley, 1992.

Programming patterns in C++.

J. Coplien, D. C. Schmidt (eds): *Pattern Languages of Program Design*. Reading, Mass.: Addison-Wesley, 1995.

The first known collection of pattern languages.

M. Fowler: *Analysis Patterns: Reusable Object Models*. Reading, Mass.: Addison-Wesley, 1997.

A well-known book about business patterns.

M. Fowler: *Patterns of Enterprise Application Architecture*. Boston, Mass.: Addison-Wesley, 2003.

An important contribution to the literature for the software architects.

E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*. Reading, Mass.: Addison-Wesley, 1995.

The key reference for this chapter.

C. A. R. Hoare: *Notes on Data Structuring*. In O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Structured Programming, London: Academic Press, 1972.

R. E. Johnson, B. Foote: "Designing Reusable Classes." *The Journal of Object-Oriented Programming*, June/July 1988, Vol. 1, No. 2, pp. 22–35.

More on the subject of black-box and white-box frameworks.

T. Lewis (ed.): *Object-Oriented Application Frameworks*. Greenwich: Manning, 1995.

An important early work on frameworks.

B. Meyer: *Object-Oriented Software Construction*. Second Edition. New York, London: Prentice-Hall, 1997.

The information about the contract model is particularly important.

Oscar Nierstrasz, Simon J. Gibbs, Dennis Tsichritzis: Component-Oriented Software Development. *Communications of the ACM*, 1992, Vol. 35, No. 9, pp. 160–165.

W. Pree: *Design Patterns for Object-Oriented Software Development*. Reading, Mass.: Addison-Wesley, 1995.

A presentation of patterns with a high degree of abstraction.

D. Riehle: "The Perfection of Informality: Tools, Templates, and Patterns." *Cutter IT Journal*, Joshya Kerievsky, September 2003, Vol. 16, No. 9, pp. 22–26.

This is a paper clarifying the difference between design patterns for human use and design templates which can be used automated software construction and conformance checking.

D. Riehle: *Framework Design*. Dissertation, Swiss Federal Institute of Technology (ETH). Zurich, No. 13509, 2000.

D. Riehle, H. Züllighoven: "A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor." Chapter 2, pp. 9–42. In J. O. Coplien, D. C. Schmidt (eds.): *Pattern Languages of Program Design*. Reading, Mass.: Addison-Wesley, 1995.

D. Riehle, H. Züllighoven: "Understanding and Using Patterns in Software Development." *Theory and Practice of Object Systems*, Vol. 2, No. 1, 1996, pp. 3–13.

Three of our contributions to the patterns and frameworks discussion.

M. Shaw, D. Garlan: *Software Architecture: Perspectives on an Emerging Discipline*. New York, London: Prentice-Hall, 1996.

Primarily a book about software architectures, but important here because of the pipes and filters pattern.

C. Szyperski: *Component Software*. Reading, Mass.: Addison-Wesley, 1998.

Fundamental book about software components.

5

Application-Oriented Software Development

5.1 INTRODUCTION

In object-oriented software development, software as a technical product is often the focus of the entire development work. We believe that this is not sufficient to ensure the development of high-quality software. Therefore, we propose an application-oriented approach. The main arguments are as follows:

Motivation for application-orientation

- Object-oriented software development should always be a means to an end. For application software this goal usually is supporting the customer-related business processes and services of an organization or company.
- Application software should primarily be targeted at the usage quality of the product. Usage quality covers those characteristics of software that are important and can be assessed by actually using it in a work context.
- High usage quality can be achieved only through an application-oriented evolutionary development process.

The reader will learn that in order to develop application software of high usage quality it is necessary to focus on the tasks and concepts of the respective domain. Combining a high level of usage quality with desirable technological characteristics like changeability, re-usability, or understandability leads to the principle of structural similarity. In the final sections of this chapter we explain that good software can only be developed if we find the right ways and means to actively integrate the users and other domain experts into the process.

5.2 APPLICATION-ORIENTED SOFTWARE

This book is about the development of interactive application software. The specific ways and means we employ to develop this type of software in the T&M approach go under the label of "application orientation." We explain this concept starting with the term application software.

5.2.1 Application Software

What is application software?

In our definition, *application software* is a purposeful means to support domain-specific tasks in one or several application domains. For this purpose, existing or new problem-solving strategies are implemented in software.

In other words, application software helps people to cope with their tasks in a more effective, less strained, or more satisfying way.

Application software models a section of the real world, oriented to a usage concept. This section of the real world can be specific (for a dedicated system) or generic (for standard software).

Application software serves to control technical processes or support work processes. When designing application software, it is important to ensure both the appropriate domain logic and easy manipulation and presentation for the user.

The implementation of application software requires a system base, normally consisting of hardware and the underlying system software.

Regarding application software, we usually take the following prerequisites as given:

- Application software development is professional team work.
- The software product and its development process are closely related.

5.2.2 Definition: Application Orientation

Application orientation, to us, is a specific way of looking at software development. We see it as the required connecting piece between customer orientation as a corporate strategy and object-oriented software development as a construction technique.

Within a software development process, *application orientation* is aimed at the following software characteristics:

- **The functionality of a software system is oriented to the tasks involved in the application domain.**
- **The manipulation of a software system is user-friendly.**
- **The processes and interactions defined within a software system can be easily adapted to the actual requirements of a working context.**

Application orientation and tasks

Though these characteristics may seem trivial at first sight, we will discuss some particularities to show their importance.

One elementary requirement is that application software has to have the specified functionality. However, we define this functionality[1] primarily according to the *tasks* to be completed within an application domain rather than to a specific *workflow*.

1. The functionality is not *part of the tasks* but is *oriented toward the tasks* (rather than to a specific workflow).

To better understand this idea, we first give a brief definition of tasks.

A *task* is handled or addressed by a qualified person within an application domain. A task forms a meaningful and targeted unit of work, which can normally be named. A person will choose the adequate activities or steps to complete a task. This sequence of activities or work steps may change according to the specific situation at hand, so there may be different ways to arrive at the result of a task.

Usage model

The handling and presentation of an application system should not be based solely on general ergonomic principles. Every application system needs a comprehensive usage model. This usage model should reflect the working experiences and domain concepts of its users. In this connection, we must think carefully of the objects and work processes that form the basis for manipulating our future application system.

In summary, application orientation means that the wishes and requirements of our customers should be optimally met. To achieve this goal, we have to offer fitting software support, where we must decide which processes in the system should be predefined and where the user should be able to work in a more flexible way, depending on the specific working context.

5.2.3 Background: Application Orientation

Human routine work

Application software has traditionally (and successfully) automated human routine work. Human work has been replaced or reduced to a few necessary input activities by software. Good examples can be found in the banking industry, where programs are used to transfer money or calculate account fees. These programs show little application-specific characteristics or usage quality. In fact, they have traditionally been oriented to pure data-processing or number-crunching tasks. A closer look at their user interfaces reveals their typical mainframe-oriented combination of menus and screens, or what could be called "data windows."

Support of daily work

This situation changed dramatically when graphic applications on workstations and personal computers were introduced. The focus shifted from automating human routine work to supporting daily jobs. This shift came about naturally, because the first graphic user interfaces (GUIs) for office applications were intentionally built around the desktop metaphor (see Section 3.5.12). Graphic user interfaces represented an important step towards improving the usage quality of application software. Suddenly, software design became visible and "touchable." Daily working activities were simulated on computers. Users could use several tools to work on different objects to complete their tasks. They could put things aside and continue using them later, just as in the real world.

This also had implications for software developers. They obviously had to deal more intensively with the issue of which tasks people have to handle as part of their work, in particular what objects they use to complete these tasks. This means that application orientation became an important requirement in software development. We define application orientation in the form of a leading metaphor and design metaphors (see Section 3.3).

5.2.4 Usage Quality

Another result of developing tools and materials on an electronic desktop was that it was no longer necessary to specify routines to automate tasks in software. Instead, users can themselves determine when and with which tool they want to process an object. This removed a major problem in software development, that is, writing algorithms for complex and situation-specific work processes. On the other hand, if software components are to be used for different work situations, then their usage quality becomes a critical quality characteristic.

Usage quality is important for the use of interactive application software. In simple terms, usage quality is what you see and realize when you are working with a piece of software to do your work. Thus, usage quality is determined by the users and other parties involved, based on external quality characteristics in real-world use. According to international standards, the evaluated characteristics are suitability, transparency, controllability, fault tolerance, self-description, conformity with expectations, and fault robustness.

5.2.5 T&M Design: Structural Similarity

The main idea behind the T&M approach is to take relevant objects and concepts from the application domain and use them as a basis for the development of the software-specific models (see Section 6.8). However, the result should be more than an application-oriented system from the user's perspective. We also require a close correspondence between the concepts and the terms of a domain (now often called "ontology") and the software architecture.

Within the T&M approach, _structural similarity_ refers to the relationship between the software and the application domain. The technical components of a software system should model the relevant concepts and objects of the application domain. Thus, the architecture of the application system should reflect the most important relationships between the concepts and objects of the application domain.

Benefits of structural similarity

This structural similarity offers two main benefits. First, the application system represents the objects of its users' work in familiar terms, so that they can organize their work in familiar ways. Second, the developers can relate software components and application concepts, and identify mutual dependencies when they have to implement domain-specific and software-specific changes.

Structural similarity has been a primary object-oriented design principle for years. In the 1970s, Alan Kay coined the phrase "computing is simulation." But this principle was mainly used for modeling an object-oriented system "in the small." So we would expect to find domain concepts like *account* or *bill* or *truck* as classes of an application system. In this book we show how to apply structural similarity to the architecture of large interactive and distributed system (see Chapter 9).

5.3 THE DEVELOPMENT PROCESS

If application orientation, usage quality, and structural similarity are important characteristics of application software, then this also influences the software development process itself. With the Unified Process becoming quasi-standard, iterative strategies are accepted as state-of-the-art in software development. We interpret iterative strategies in an application-oriented way. This means that we propose to add techniques and document types to the process in order to establish close feedback cycles between the developers and the domain experts.

5.3.1 Definition: Software Development

We begin with a definition:

The term *software development* describes all activities that lead to a software system that is being used. We distinguish these activities as follows:

- **Product-oriented activities, such as analysis, design, and programming, and their results, as in the form of defining documents directly used in the product.**
- **Process-oriented activities, which support product-oriented activities and include cooperation and coordination during a project as well as product management and quality assurance.**

Note that the identification of requirements is already stamped by these characteristics, because our context is the daily work situation:

For developers, *application orientation* means that they have to understand the tasks involved in each of the workplaces in an application domain.

To be able to identify and understand the domain-specific tasks, developers analyze how objects are handled within the current work processes at workplaces.

Modeling is aimed at reconstructing the domain-specific language of the users, that is, both the models and the software system express the relevant terms and concepts of the application domain.

5.3.2 The Application-Oriented Development Process

A new application system normally has a major impact on the work environment and the tasks involved, and the use context often changes within a project, so that it is necessary to ensure that both the product and the development process are application-oriented.

Characteristics of an application-oriented development process

An application-oriented development process is characterized by the following characteristics:

- Increased involvement of the users over the entire project.
- Continuous feedback about the results in easily understandable documents and prototypes.
- Reorientation of the developers away from a technical construction view at the computer to interaction with the users and other participating groups.

To ensure that our development process is application-oriented, we have to elaborate a set of prerequisites, which will be explored further in the course of this book:

Prerequisites for an evolutionary development process

- On the one hand, an evolutionary process should be flexible to allow all participants to participate actively in the project. This means that the process should be such that it can be adapted to changing situations. On the other hand, we want to ensure that our project can be controlled and planned (see Section 12.5).
- Application-oriented analysis and development documents are the prerequisites for the active cooperation of all participants (see Chapter 13).
- The developers should be able to implement the structural similarity between domain-specific and technical models (see Chapters 7 to 9).

5.3.3 Discussion: The Development Process

Software development should be objective and independent of people

We normally expect the *objectivity* and *independence of specific persons* in our development processes, based on two questionable assumptions. The first assumption is that application software has to represent a solution to a given problem, as when solving a mathematical problem. Section 12.2.2 discusses the basic issues of this assumption, and Section 12.2.4 relates it to development strategies.

The second assumption is linked to the first one. It assumes that software, on the basis of unambiguous descriptions, can be constructed based on a division of labor and in a temporal sequence similar to assembly belts. Section 12.2.1 discusses this assumption.

Software development is a communication and learning process

Let's look at these assumptions from the standpoint of our notion that software development is first and foremost a *communication and learning process* between the participants. We said that this is the only way for developers, who are normally not experts in the application domain, to acquire the domain-specific knowledge required to develop an application system with high usage quality. However, the future system is not a "unique solution" to a well-defined problem. All groups involved have to discuss and agree on both the future tasks and the support the system under development should provide. Some aspects of this process can be specified in documents, so that these documents can be understood and used by other people. On the other hand, the essential experiences, views, and values that come with every development project cannot be described in these documents.

Document-driven development process

The application-oriented development process we suggest cannot be handled in ad-hoc ideas and procedures. It does not means just "muddling through" such that all parties just talk to each other. In fact, the T&M development process is "document-driven," that is, the elaboration of application-oriented documents is one of its major requirements. This orientation corresponds to what is known as "use case driven" in UP, where business use cases describe the work processes and tasks, while use cases specify the functional requirements of the system under development, based on UML documents. In the T&M approach, we add more application-oriented document types.

A suitable choice of documents (see Section 5.3.11) should ensure that the most important domain-specific and technical issues are included and dealt with.

Application-oriented and technical document types in connection with a leading metaphor and matching design metaphors provide developers with the necessary basis for working towards their ambitious goal: high usage quality. Whether or not this goal can be achieved depends on the adequate realization of the underlying concepts.

5.3.4 The Author-Critic Cycle

In our view, it is essential to give the developers a guideline for their design of the development process. The fundamental principle is to think of software development as a communication and learning process. This principle has to be encouraged by continuous feedback between the participating groups. We ensure and structure this feedback in so-called author-critic cycles.

The *author-critic cycle* means that all documents and operative software versions are included in an evolutionary process between all participating groups. Alternating between analyzing, modeling, and evaluating work, the participants exchange feedback on the results and goals within their software development process. This is based on the rule that the authors of a result should never be their own critics.

By *analyzing* the tasks within a specific application domain, the developers gain initial access to the application concepts and the objects and processes involved. This understanding is represented as domain-specific and technical *models* in the form of documents and prototypes. *Evaluating* the modeled aspects means that the critics become involved. Critics are primarily the users of the system under development, but also technical experts, such as IT organizers or network administrators, who are contributing to the development process from their specialized disciplines.

The author-critic cycle is not about sequentially working on milestone documents in a fixed order. It is the continuous exchange of feedback between all participating groups, as illustrated in Figure 5.1. This obviously conflicts with the classic waterfall or phase models (see Section 12.1), but we think it is a fundamental prerequisite to ensure successful handling of an application-oriented software project.

5.3.5 Discussion: The Author-Critic Cycle

To better understand our approach, it is important to realize that the problems identified in the feedback process determine the subsequent activities and the selection of documents to be edited. For this reason, the author-critic cycle is not based on a defined and rigid sequence of documents and phases. In the ideal case, you can edit any document at any given point in time. Naturally, all activities within a specific project

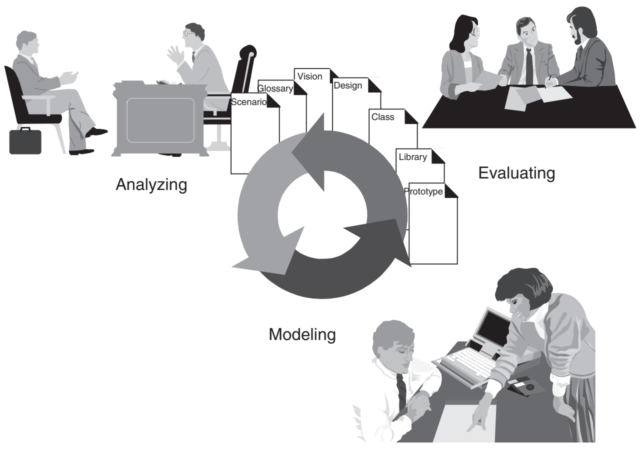


FIGURE 5.1

Schematic presentation of the author-critic cycle.

have to be planned and controlled; they cannot be subject to ad-hoc ideas. However, it should be clearly understood that the process itself is basically determined by application-oriented issues rather than by software-specific mechanisms.

The EMS example

Let's see this principle in the context of our EMS example. Assume that our developer team interviews the person in charge for equipment management at their workplace (analysis). The results from this interview are documented in scenarios, glossary entries, and in a business use case diagram (modeling). These documents are then reviewed and commented on by the interview partner and developers (evaluation). These comments encourage the developers to further familiarize themselves with the task on hand, that is, device procurement, and to conduct additional interviews if deemed necessary. Next, they will write system visions and create use case diagrams and an initial presentation prototype. This presentation prototype is then evaluated by the device manager and the developers in a workshop. Finally, the workshop participants agree that they overlooked the necessity of moving staff to different rooms. They arrange for a meeting with the corporate management and follow up on whatever steps may be involved. This is how the author-critic cycle evolves.

5.3.6 Evolutionary System Development

The view of a software development process described here fits well with the leading metaphors and design metaphors described earlier (see Chapter 3). The discussion in Section 3.3 about the roles in a software project showed that developers are not specialists in every conceivable domain. They are the partners of future system users. This means that application orientation involves both a new construction method and a matching development strategy. This is based on the general principle of evolutionary system development.

***Evolutionary system development* is aware of the evolutionary character of software. Software is subject to frequent changes, so that evolutionary system development deals with such changes in a meaningful way.**

- **Mutual learning between developers and users is introduced.**
- **Changes in the domain-specific and technical contexts are taken into account systematically.**
- **The actual use of the system under development will lead to new requirements, in turn triggering additional development cycles.**

5.3.7 Documentation in Software Development

Although "documentation" is probably one of the most frequently used catchwords in software development, software engineers do not always agree about the meaning of documentation and how it is best used when designing software. We define documentation as follows.

***Documentation* denotes the *process* that includes the activities required to create and modify documents (e.g., "I have to finish this specification"). Documentation denotes the *product* that represents a set of documents for a specific purpose at a specific time (e.g., "These are the documents we should use for review.")**

A document is a record representing a fact. A document can include text and other elements. In software development, documents are used to represent the dynamics and statics of the software system under development. Additionally, there exist metalevel documents (i.e., documents about documentation).

5.3.8 Discussion: Documentation

Documentation problems

The only point on which IT experts agree when it comes to documentation is that it is important, but problematic. Those involved in designing and implementing a software product normally consider documentation a nuisance. For developers in their role as potential authors, documentation is an additional cumbersome task that seems to keep them from their actual development work. Users, on the other hand, when working with prototypes and application systems, become frustrated when they find that there is not enough documentation available.

Traditional milestone documents based on fixed schedules have proven to be too inflexible in software development. Documentation should no longer be an end in itself. Strategies like eXtreme programming (XP) show that so-called lightweight or agile development processes for very flexible application systems manage with a minimum of conventional documentation. However, software cannot be developed professionally unless there is specific and systematic documentation.

Writing documentation while a project progresses is time-consuming and sometimes beyond the point of justification. When subsystems are already documented but then are not accepted by the users, we think we have wasted time in documentation.

Documentation that is developed during the course of your software project tends to be very complex. Therefore, it is difficult to see whether or not all models have been fully and consistently described so that they can be understood at the end of the project. Many developers tend to misinterpret this point. The consequence is that important development decisions and results cannot be reconstructed and thus documented at the end of a project.

5.3.9 Application-Oriented Development Documents

So far in this chapter, we have emphasized the importance of application orientation for our approach. This section transfers this concept to the documents used in our development process.

> ***Application-oriented development documents*** **are common objects of work for developers and users. The representations, concepts, and notions used are oriented to the domain language. Application-oriented development documents model the domain tasks, objects, and concepts in a way that all participating groups can easily understand. They are created, evaluated, and edited in the course of author-critic cycles.**

Elaborating documents is an effort primarily focused on the underlying communication and learning processes. To achieve this goal, our documents have to represent the domain, the application system, and the development process in an adequate form.

Documents written in special language of the domain

All application-oriented development documents have to be written in the special language of the application domain. This rule distinguishes application-oriented development documents from the other technical documents involved in the development process. When using UML, this means that only a few documents or diagrams defined in UML (e.g., business use cases, use cases, and diagrams) are suitable for cooperation between users and developers (see Chapter 13, which is dedicated to T&M document types). Note that the respective documents in themselves do not represent any value, but rather form the basis for our communication and learning processes.

To ensure the systematic use of our documents in the course of a project, it is important to classify and describe them as document types, in the sense of the UML standard. However, while UML concentrates strongly on diagrams, we normally add standardized text to these diagrams. For this reason, we define document types as follows.

A *document type* defines a set of similar documents in the object-oriented sense, that is, the description of a document type specifies how the instances of such a type behave. As with every type definition, the *internal structure* of these documents is not important, but their possible and meaningful *forms of behavior is*. Descriptions of document structures are meant as examples.

5.3.10 Discussion: Application-Oriented Document Types

When we say that all relevant development documents should be common objects for developers *and* users, we encounter another problem, in addition to the general problems relating to documentation already mentioned.

Common project culture

Let's first see what we mean by one of the important aspects of application orientation: establishing a *common project culture*. What does this mean? It means primarily that the participating groups speak a common language. If we achieve this goal, then the *developers* will understand the tasks and problems involved in the application domain, and the *users* will understand how the system under development will support their work.

Unfortunately, this project culture doesn't come about on its own. We have to work to establish it, based on the communication and learning processes discussed above. The ideal learning goals could be formulated like this: The developers of a software component know the tasks and activities involved in the application domain to the point that they can estimate and discuss the domain-specific implications of each of their technical design decisions. The users have a sufficiently precise picture of the possibilities offered by the software system under development, so that they can articulate their domain-specific requirements.

Application-oriented development documents

To work towards this ideal situation, we use different application-oriented development documents. These documents become the basis for our common work, that is, they are created, evaluated, and revised within the author-critic cycles.

It should be understood that application-oriented documents were not invented to keep ourselves busy. In fact, it is serious work aimed at eventually creating an application system with maximum usage quality and optimum utilization of all human and material resources available. To achieve this goal, our documentation has to represent the different aspects of software development. In our approach, these are the *product-specific* and the *process-specific* aspects. We further divide the product-specific aspects into modeling of the existing application domain and the future software system.

Zull-05.qxd 26/8/04 11:31 AM Page 111

All documents existing for an application system are created with respect to constructive quality assurance. This means that we have to consider quality assurance when writing and revising them in the author-critic cycle. How the appropriate level of quality can be reached via so-called base lines and thereby becomes part of project planning is described in Section 12.8.3. This kind of constructive quality assurance is very important for ISO9000 certification as well.

Constructive quality assurance

For the actual implementation of software, eXtreme programming has become an issue of hot discussions. We have used this approach in several software projects with good results and will explain it in Section 12.3, which discusses development strategies and quality assurance.

5.3.11 T&M Design: Application-Oriented Document Types

This section gives a brief overview of the characteristic document types used in the T&M approach (see Figure 5.2). We will introduce only the important application-oriented document types. Chapter 13 is dedicated to document types, including a detailed discussion of UML within the T&M approach.

Development documents represent the application system during its entire life cycle. This means that these documents have to be seen on a different level than user documents (e.g., user manuals or help documentation), which will not be described in this book.

Historically, scenarios, glossary entries, and system visions have been the core of application-oriented document types in the T&M approach. We will briefly describe

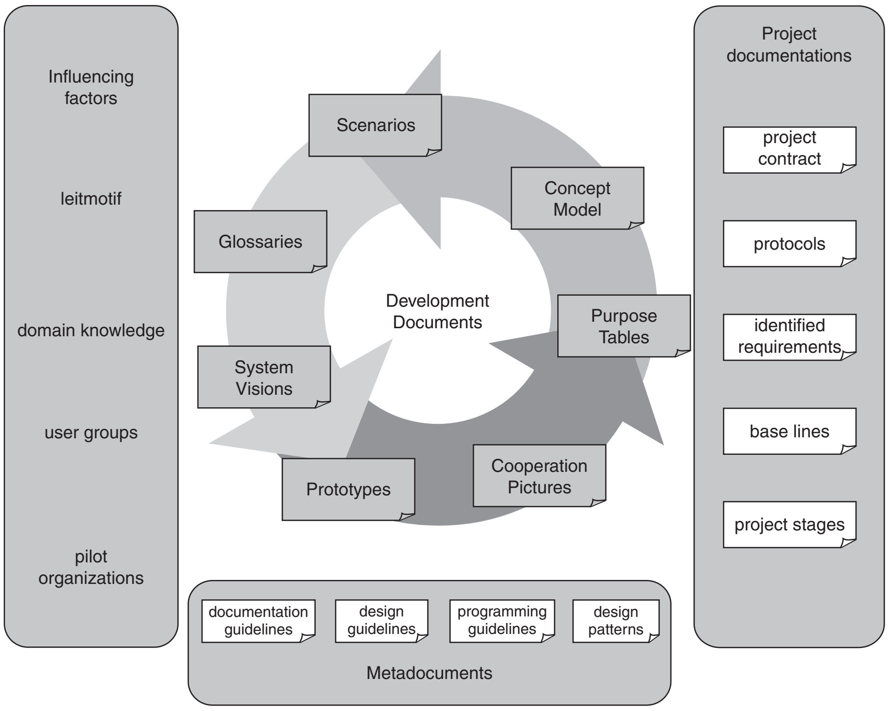


FIGURE 5.2
Documents in the development process.

these fundamental concepts:

The basic T&M document types

- *Scenarios* describe the current work situation. They focus on the tasks involved in the application domain and the way these tasks are completed. Scenarios can be compared with the business use cases of UP.
- A *glossary* defines and reconstructs the relevant concepts and terms of the domain language. Though the use of glossaries plays an important role in UML and UP, they are not "official" UML documents.
- *System visions* are at the transition between the analysis of the application domain and the construction of the future system. Based on scenarios, system visions anticipate the different characteristics of the future application system. This means that system visions roughly correspond to use cases in UML. They are specialized according to our application-oriented view.

Using the T&M design in a wide range of different domains, we have seen that additional application-oriented document types were required. For the analysis and modelling of cooperative tasks, we found that we didn't have appropriate representations on hand. Cooperation pictures and purpose tables have proven to be important additions to our choice of application-oriented document types. They represent a graphical and textual model of the cooperation between workplaces or roles, showing more clearly the kinds of objects and information that should be exchanged and for what purpose. This means that they reach further than UML use case diagrams and are much more application-oriented than most diagrams, such as collaboration diagrams. Chapter 13 describes cooperation pictures and purpose tables in detail.

Classification of document types

Development documents can be classified or grouped in different ways. For example, we have document types (e.g., scenarios, cooperation pictures, purpose tables, and glossaries) that basically describe the *actual state* of an application domain. Sections 5.3.12 and Chapter 13 will show the importance of these documents. In contrast, there are other document types (e.g., system visions, software designs, and various technical documents) that describe the different (future) *development* stages of an application system, from the initial design to the final installation.

In contrast, the difference between the *statics* and *dynamics* is of rather theoretical interest in connection with our models. Similarly to how we document the dynamic properties of an object-oriented program at runtime and the static structure of the program sources, for example, by interaction and class diagrams, we also concentrate on either of the two aspects in our document types. It shows that we have a much larger choice of application-oriented development documents to describe the dynamics (e.g., scenarios, system visions, purpose tables) than to describe the statics (e.g., glossary, some forms of cooperation pictures), although these aspects are more balanced for technical development documents.

An important approach is to divide documents by their *ranking within the communication process*. From this vantage point, it initially appears that application-oriented document types (e.g., scenarios, glossary, cooperation pictures, and prototypes) are more important that those document types that primarily address the developers, like class diagrams and framework architectures. On the other hand, structural similarity implies that technical document types become more important. In fact, the main part of this book describes the principles and constructions of these technical models.

In line with the increasing importance of the *support of cooperative work*, we see a growing need for such document types, which are suitable particularly for software development work in this area. We will discuss the documents relating especially to the support of cooperative work in Sections 13.6 to 13.8.

5.3.12 Discussion: T&M Document Types

The document types we have proposed and the related evolutionary strategy should be thought of as integral parts of the T&M approach. For this reason, it doesn't make sense to see them as individual document types that can be arbitrarily selected and used in isolation. On the other hand, we are not trying to say that they should be used *exclusively* in this approach. It means that you should carefully adapt and handle them if you intend to use them in a different context.

Many of the document types we introduce in this book are described in the literature in similar forms but with different names. For example, "scenario" is often used for completely different document types. We will not discuss this issue at length, but rather comment where necessary to ensure better understanding. Note that the document types we propose are not based on doing everything in a new and different way. As software engineers, we don't want to "reinvent the wheel," but rather rely on proven technical tools, such as UML diagrams. What's important to understand are the new accents or flavors that application-oriented document types add to proven notations or diagrams. Thus we want to encourage a "rethinking" process.

T&M documents are not like milestone documents

At this point, a word of caution must be noted. Those who are familiar with traditional document types, or so-called milestone documents, will tend to transfer the document types described in this book to their familiar background, only to find out that nothing "really" new is proposed here. Such an attempt risks maintaining the traditional way of analyzing, designing, and documenting a software system. This risk is not trivial. For example, it could mean overlooking that a method exists only within the tension between given techniques and tools, a view and a concrete usage context. In this respect, we strongly distinguish the application-oriented document types introduced here from conventional documents. Application-oriented document types were developed during several concrete projects and in line with the application-oriented perspective presented in this book. We often observed in real-world projects that most developers found it a little hard to get used to our document types, but that they found them very helpful as they went along in the project.

5.3.13 Project Documents

The processes within a project have to be documented, just as the software under development, to ensure that it can be planned, controlled, and evaluated. This issue has been dealt with in the object-oriented literature only recently, while much work is oriented to traditional project management concepts. For this reason, we dedicated an entire section to project management (see Section 12.5). Here, we simply give an overview of the document types relevant in this respect.

Project contract

A project or a subproject should be explicitly initiated. This means that it should always be based on a *project contract*, even for in-house development projects.

This contract regulates the project details and responsibilities. The project should clearly state the entities who are responsible for resources, staff, design decisions, and acceptance of the system.

A project contract should document the system to be developed, the entities who develop it, and what resources will be used to develop it. In this respect, the principles of evolutionary system development require a formulation of the contract as a letter of intent rather than as a complete project specification. In the center is the goal and the vision rather than a detailed description of the properties of a system and how it will be implemented.

Identified requirements

A document type called *identified requirements* describes the concepts, properties, and requests relating to the system under development as they result in the course of the project. The identified requirements represent something similar to a diary about the results of discussions relating to the system characteristics. System characteristics can include both domain-specific and software-specific aspects. In contrast to the project contract, the identified requirements are continually updated to reflect the history of system requirements as they change.

Project control documents

For daily work within a project, the actual *project control documents* are important. They describe the so-called *base lines* and *project stages*. For a microplan of the construction process, the story cards from eXtreme programming have proven to be useful. Section 12.8 will describe different methods for project control based on temporal and qualitative guidelines.

5.3.14 Documentation Guidelines

No project will automatically produce documents in a uniform and ordered way. What we need are instructions describing the documentation process and a common understanding of all document types used. The document types that hold these descriptions are called *metadocument types*. All metadocument types obey the following rule: the shorter and more practical they are written, the better. This means that it is intuitive to always have only the most current version of such a document type out in use. It also means that it is not sufficient to use UML descriptions as your only documentation guideline for a project.

Documentation guidelines

Documentation guidelines describe different documentation types. This refers to the outer form, the structure, and the contents. In addition, there are regulations and comments about the documentation process itself, notes about how these document types should be created. You may think of Chapter 13 as a general example for documentation guidelines.

Design guidelines

Another important element in many conventional projects are *design guidelines* for the design of interactive systems and their use interfaces, such as common user access (CUA) guidelines of IBM. We have observed in our real-world projects that design guidelines can be formulated briefly and easily, if a guiding metaphor, including its design metaphors, is part of the development culture of an organization. It will then become clear that design guidelines represent more than the user interface layout; they also include fundamental ergonomic aspects of interactive software systems. In the professional development of interactive systems, this issue, which will not be further dealt with in this book, has at least been recognized, though most companies do not have a qualified staff necessary to support it.

Another important element are *programming guidelines*. No major software project would work over the long run without programming guidelines. This book focuses on design patterns (see Section 4.3.5), so programming guidelines will not be discussed further here.

Programming guidelines and design patterns

5.4 REFERENCES

K. Beck: *Extreme Programming Explained*. Reading, Mass.: Addison-Wesley, 2000.

The standard work by the "inventor" of eXtreme Programming.

K. Beck, M. Fowler: *Planning Extreme Programming*, Reading, Mass.: Addison-Wesley, 2000.

The book that is most frequently quoted for planning with XP.

G. Booch: *Object Solutions: Managing the Object-Oriented Project*. Reading, Mass.: Addison-Wesley, 1995.

More on the topic of project management in OO projects.

G. Booch, J. Rumbaugh, I. Jacobson: *The Unified Modeling Language*. Reading, Mass.: Addison-Wesley, 1999.

The current standard work on UML.

A. Goldberg, K. S. Rubin: *Succeeding with Objects: Decision Frameworks for Project Management*. Reading, Mass.: Addison-Wesley, 1995.

More on the topic of project management in OO projects.

I. Jacobson: *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, Mass.: Addison-Wesley, 1992.

More on the topic of project management in OO projects.

I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*. Reading, Mass.: Addison-Wesley, 1999.

Currently the standard work on procedures in projects in the context of UML.

A. C. Kay: *Microelectronics and the Personal Computer*. Scientific American, 1997, Vol. 237, No. 3, pp. 230–244.

The paper outlining the visionary ideas of Alan Kay.

G. Succi, M. Marchesi: *Extreme Programming Examined*. Reading, Mass.: Addison-Wesley, 2001.

Based on contributions submitted at the first XP conference (Italy, June 2000), this book offers an overview of conceptual and practical discussions on the topic of eXtreme Programming.

6

Software Development as a Modeling Process

6.1 INTRODUCTION

This chapter describes application development from a software-engineering view. We discuss how software development can be seen as a modeling process in the sense of the Unified Process. We distinguish the domain model from the application system model, and we discuss the actual state of the application domain and the design of the future system.

The message of this chapter is that you should have an explicit model of the application domain in its actual state. This model helps to understand the relevant tasks and the concepts behind them. In our application-oriented approach, understanding the tasks and concepts naturally leads to the design of the future system.

6.2 A SIMPLIFIED SOFTWARE DEVELOPMENT MODEL

One view of software engineering sees software development primarily as a *modeling process*, because different executable and nonexecutable models are created in the course of a project. This view has become popular with the proliferation of UML and UP.

This section begins with a simplified model of the object-oriented software development process. It shows the segments of reality that we are dealing with (sometimes called the "Universe of Discourse") and the models we build from this reality for our software development. Finally, we will look at contexts as they influence our modeling efforts. We show that there are two fundamental models, the domain model and the application system model, that you need when developing software.

6.2.1 Discussion: A Descriptive Software Development Model

A descriptive software development model gives no instructions

Here, we will not attempt to present a development strategy giving instructions for concrete software projects, or what should be done by whom and when. These questions will be dealt with in Section 12.8. What we need in the context of this chapter is a *descriptive model* that can represent issues and questions that we have identified so far for our software project as a basis for discussion.

Figure 6.1 shows such a descriptive model. It represents the application domain as its starting point, that is, the segment of reality for which we develop an application system, and the resulting application system itself.

To be able to build such an application system, we basically need two models: one domain model, and one application system model.

Contexts for modeling

Two contexts are particularly relevant for modeling. First, the desired handling and presentation of the application system, or something that is often called the "look and feel" of an application. And second, we need the technologies that we intend to use in our development, for example, the systems, languages, and tools to make our application system workable.

Obviously, the application domain takes the dominant position within the modeling and development process. To keep our example simple, we left out the differentiation of individual modeling activities or the required author-critic cycles. The example in Figure 6.1 is further simplified, because we left out the feedback in the course of the application system under development (e.g., prototyping) on the models and the application domain.

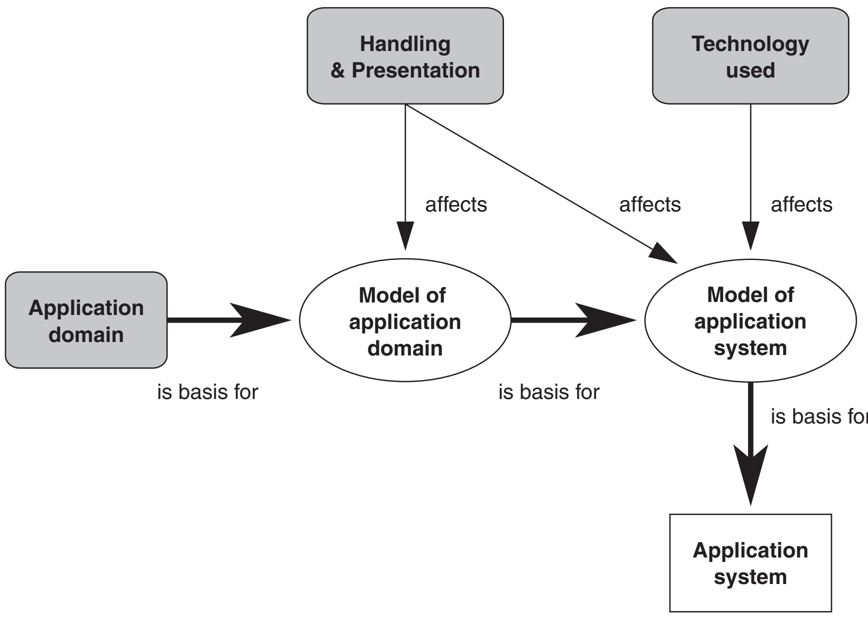


FIGURE 6.1 Simplified model for object-oriented software development.

6.3 THE APPLICATION DOMAIN

One of the prerequisites for the development of a software system is that we have a definition and a clear understanding of the contents of the application domain concerned. This is the part of an organization for which we are to develop application software. This means that the application domain is our starting point and the context for our software development.

Many development methodologies take this understanding of the application domain for granted. They assume that the developers somehow know what domain they have to deal with. In our experience, however, this is a crucial issue within a software project, so we make it explicit in our application-oriented approach.

6.3.1 Definition: Application Domain

Application domains can either be very extensive or very limited. In this book, we see application domains mainly in the context of commercial or governmental organizations, but they could easily be in social or private organizations. Application domains include banks, insurance companies, or hospitals. In this book, equipment management for a small software company is our main example. Internet applications have become increasingly important, especially for the home and entertainment domains.

An *application domain* is the segment of reality for which a software system is developed. It is the background or starting point for the actual-state analysis and the creation of a domain model.

An *application domain* can be an organization, a department within an organization, or a single workplace.

The concept of an application domain is at least as wide, so that the domain concepts and relations relevant for the construction of models can be well understood during the analysis of the actual state of the domain. On the other hand, its extent should always be limited, that is, never be too complex.

An application domain normally includes a domain-specific language. This means that people in this domain use specific terms and concepts and think about their domain in a specific way.

6.3.2 Discussion: Analyzing the Application Domain

The *application domain* is very important, because it is critical for our application-oriented way of developing software. The developers analyze and describe the actual tasks and situations (see Section 6.41) that characterize the application domain in the domain-specific language. This corresponds to an actual-state analysis, where the typical processes and the objects used in these processes are represented in their domain-specific use contexts, for example, as scenarios or glossary entries (see Section 5.3.11). It is similar to the business model in UP. At the same time, the application domain is the basis for the construction of a domain model. Together with the analysis and description of the actual state within the application domain, we are gradually building the *domain model*.

Building a domain model

This model represents the segment of the actual application domain to be supported by the software system under development. Obviously, our domain model is similar to the domain model in UP. Later in this chapter, we will explore the question whether or not the UP domain model can meet all our requirements, that is, easy to understand and easy to construct, as *one* model.

This interest of developers in an application domain is not limited to the segments that will be mapped in the domain model; they will also deal with segments required to understand the current work situation within the actual-state analysis. It is at once a risk and an art to find the right limits for the actual-state analysis, and not to let it get out of hand both in terms of time and content. Experiences from traditional data-modeling projects have shown that it is simply not possible to achieve a complete analysis of the entire application domain. The first notion of the future application system will help developers find the right point to stop analyzing. This means that the developers have to gain quick insight about the points of the application domain where software support could be useful and feasible.

The EMS example

Our recurring example has made clear so far that the software support for equipment management in our small software company has been rather poor. Once we discussed the possibilities of supporting the management's work with the employees, we soon found out that they wanted us to design a more extensive planning system for all tasks involved in the company's business. However, the developers found that the additional domain issues and problems would far exceed the project is resources. For this reason, we knowingly reduced the analysis of the relevant management and planning tasks to the actual equipment management. Issues like scheduling or job distribution were no longer taken into account.

6.4 THE DOMAIN MODEL

Reconstructing domain terms

In the T&M approach, object-oriented software development means the *reconstruction of domain terms*. As trivial as it may sound, this requires that the developers understand the underlying concepts.

Developers create an explicit *domain model* with all the domain objects, terms, and relationships existing in that application domain. This model is oriented to the existing situation in the application domain and the domain business processes.

Besides the Unified Process not very many development strategies acknowledge an explicit model of the application domain. But within an application-oriented approach, modeling the application domain and designing the future system are intertwined. Thus a well-designed domain model is a strong foundation for the software system to be built.

6.4.1 Modeling Your Application Domain

The domain model describes the elements and relationships to be supported by our application software from the domain view. In our T&M approach, we begin with the tasks and then identify *relevant objects* and their *usage forms or interactions*.

An *interaction* is a characteristic action that can be done on or with an object.

Mapping domain objects to model concepts

Once we have identified the objects and interactions and abstracted the terms, we map them to the domain-specific language and model their relationships, using object-oriented concepts like *generalization, composition*, and *dependence*. This means that we model the application domain.

The *domain model* includes all aspects of the application domain to be potentially supported by an application system. This model is oriented to the tasks and the relevant objects, including their interactions. Generalization and composition are used to represent the related terms in a domain concept model. When creating this model, we orient ourselves to a guiding metaphor and its design metaphors.

The domain model consists of the concept model and selected application-oriented documents (e.g., scenarios, glossary entries, and cooperation pictures).

When modeling the application domain, we usually write scenarios and glossary entries (see Sections 13.1 and 13.4). This corresponds to what is called "business modeling" in UP. As we continue dealing with the application domain, we identify dependencies and similarities behind the key terms and notions in these documents and eventually collect them in a separate *concept model* (see Section 13.3). This could be compared to the business modeling in UP.

When identifying the relevant objects, terms, and relationships, the guiding metaphor and its design metaphors (see Chapter 3) are extremely helpful, as they guide us in dividing all the things relating to a workplace into categories, such as, tools, materials, automatons, or containers, and to look for appropriate properties that characterize them.

The Bank example

The example in Figure 6.2 shows a segment from a bank-specific concept model, or, the securities hierarchy for a loan system. In this scenario, a bank requests some form of security before it grants a credit to provide collateral in case the borrower defaults in payment. The model's broad differentiation in width and depth means that several author-critic cycles have been completed.

As we work towards a concept model, another useful tool is CRC cards. We can use CRC cards to identify services offered by a domain object. We also include those

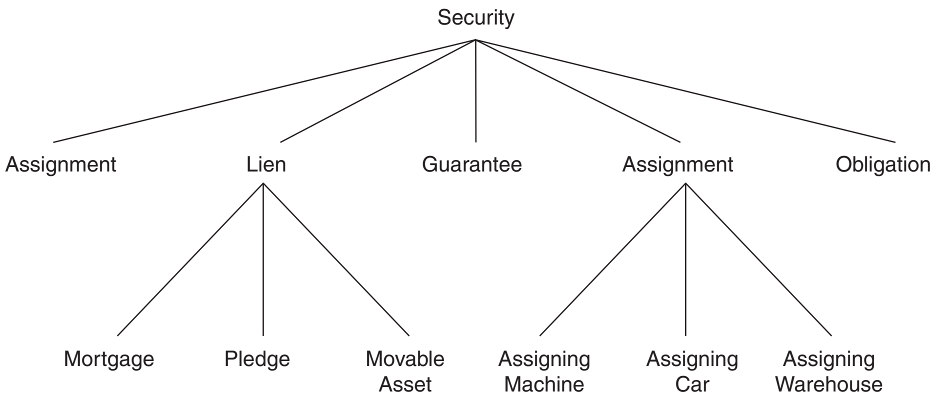


FIGURE 6.2

Example showing a bank-specific concept model.

The EMS example

Device	
classify a device	**Signature**
order a device	
approve an order	**Device State**
upgrade a device	
approve an upgrade	**Date**
check the status of a device	
dispose of a device	

FIGURE 6.3
A CRC card for our EMS example.

objects that are used by an object to provide its own services. Figure 6.3 shows a CRC card from our EMS example.

6.4.2 Discussion: Modeling Your Application Domain

To be able to limit the extent of our application domain model, we have to organize a negotiation process between all participating groups to define a domain frame for the software system under development. This effort will usually put us in a contradictory situation:

The paradox of software modeling

- To develop our application system, we need a domain model of the application domain.
- To model the application domain, we need to know the domain frame of the future application system.

This paradoxical situation cannot be solved by structuring our modeling work in a linear sequence, that is, we cannot create the entire domain model before we deal with the application system model. This is another reason why we recommend a cyclic approach.

6.5 THE APPLICATION SYSTEM MODEL

Working from the logical basis of our domain model, we create an *application system model*. In this model, we add the required technical characteristics, as given by the programming language we use or a database, to our design with its application-specific characteristics.

This section demonstrates that there need not be a wide structural or semantic gap between the model of the application domain and the more technical models of the software system.

6.5.1 Context: The Application System Model

Structural similarity

We saw in Section 5.2.5 that we need to achieve a high structural similarity for our application system model. Accordingly, we transfer the domain concept model to a similar class model. However, an initial inspection shows that the application system model includes not only classes and operations already identified in the concept model, but also additional classes and operations motivated by the underlying technique. For this reason, we can speak only of a structural similarity between the domain and software models. In addition to the technology we use, we also have to consider different forms of manipulation and presentation in our application system model.

6.5.2 Definition: Application System Model

We know that the application system model should be a basis for common work that, involves both users and developers, so it has to unify domain and software aspects. These aspects require different kinds of submodels or representations.

An *application system model* represents both the domain and software view of the system under development.

This model shows a clear structural similarity with the domain model, particularly with regard to the concept model.

In order to represent the different perspectives, we can divide the application system model into

- **a usage model,**
- **a design model, and**
- **an implementation model.**

These three submodels are described in the following sections.

THE USAGE MODEL

The close relation between design metaphors and patterns plays an important role for the usage model (see Section 3.2.1). For example, objects previously identified in the domain model, such as tools or materials, can be transferred in a structurally similar form to the matching concept and design patterns.

The usage model shows how the functionality of the system is represented at the interface and how it can be accessed by its users. It should be formulated explicitly. For this purpose, we can use different types of system visions, which we can treat here in the sense of domain use cases.

Presentation prototypes are an important means to highlight the dynamics of a usage model. Our usage model should clearly show its relation to the domain model. In addition, we have to deal with independent software or ergonomic aspects.

THE DESIGN MODEL

In addition to the domain usage model, the application system model is represented by two *software models*, the design model and the implementation model.

The *design model* is a software model showing the interrelations between the software construction units (e.g., classes, patterns, frameworks) as well as their domain and software properties.

Representing the design model, we can use suitable UML diagram types, such as class, object, state transition, or interaction diagrams, in addition to technical system visions.

Note that there does not necessarily have to be a design model for the entire system. Particularly for developments in well-known domains, we can often translate the structures of the domain model directly into program code, and the specific design model is then limited to segments of the system. This also holds true for teams familiar with XP (see Section 13.5.2), where story and engineering cards can be interpreted as an incrementally growing design model. All T&M projects have shown that it is a good idea to build explicit functional prototypes (see Section 13.6) to dynamically model the design.

The Implementation Model

Although the design model is oriented to the domain and software concepts, the implementation model is written as code in the selected programming language.

An *implementation model* implements the design models in program code. This code is transformed into executable software by use of appropriate tools (e.g., compilers and linkers).

Examples

The Bank example

Figure 6.4 uses the concept model from Figure 6.2 to show a class model for a bank loan system. Figure 6.4 shows the structural similarity between the concept model and the software model very clearly.

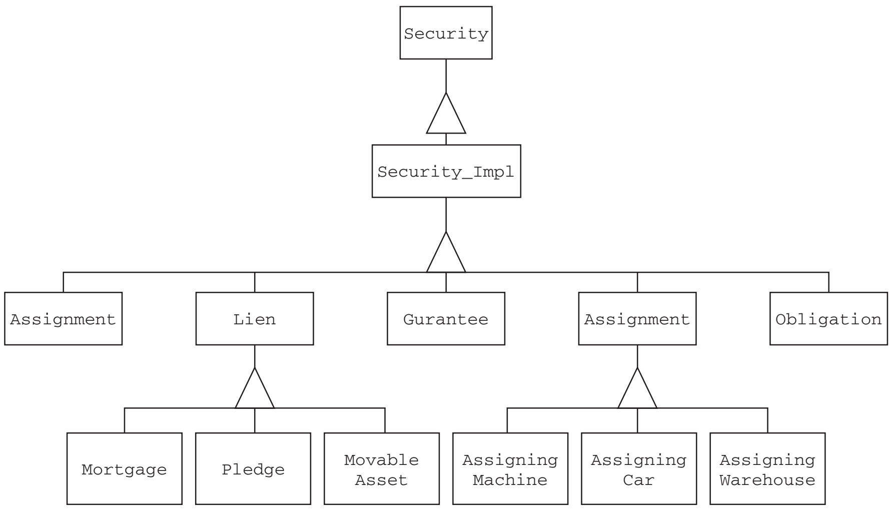
The EMS example

Let's see how these things would look in our EMS example. Assume that the analysis of our equipment management domain showed that a device is a key concept, as it has particular significance to the tasks to be supported. For example, think of the following typical ways of working with a device:

- classify a device;
- check the status of a device;
- buy a device;
- upgrade a device; and
- dispose of a device.

FIGURE 6.4

Example for a bank-specific class model.



Zull-06.qxd 20/8/04 7:37 AM Page 125

If the software developer has understood how devices are to be handled in the equipment management system, he or she can now create a class model with structural similarity based on our guiding metaphor and its design metaphors. For example, the developer could create a material class, `device`, with operations like `classify()`, `purchaseDate()`, `signPurchase()`, `isValidDeviceStateSuccessor()`.

However, these operations are based on pure domain motivations, and are not sufficient to create an operative software system. We want to display a device on an electronic desktop for the device manager to be able to manipulate it (see Section 3.5.12). To *represent* a device on the desktop, the developer decided in this case to use an icon, in addition to a description of the device. For this purpose, the developer has to connect the domain material, `device`, with its graphical representation. A simple solution could *expand the class interface*, by adding an operation, `getIcon()`, to the class interface of device. We have chosen a more general construction, where each thing on the desktop can return a `thingDescription`.

When trying to implement the room plan, the developer finds that the room plan requires only a small fraction of a device's functionality. In addition, working with room plans is often based on fictitious devices used to run plan simulations.

On the above grounds, the developer decides to consider the set of potential interactions depending on the work situation. He or she implements a class, `deviceProxy`, in addition to the `device` class. This new class can be used as a substitute for the actual material class for room plans. For the attributes in `device` and `deviceProxy`, the developer uses domain values, implemented by use of the Flyweight pattern proposed by Gamma et al.

Figure 6.5 shows an interface definition of the device class, which is very similar to the description of the device on a CRC card, as shown in Figure 6.3.

6.5.3 Discussion: Structural Similarity and Macrostructures

In our examples in this chapter, structural similarity refers to the model elements described in Section 2.1.5, such as, software objects vs. things and classes vs. concepts. In software systems, these model elements are also called *microelements*.

As useful as these microelements may be, they are not suitable for large-scale software systems with thousands of classes. What we need are higher structuring forms, or *macrostructures*, to be able to handle large systems. The application orientation of the T&M approach gives us a direction in which to search for such macrostructures. Section 9.3 will discuss the macrostructures we could use in the application domain to design large software systems. In this context, we use the term *software architectures*.

6.6 THE APPLICATION SYSTEM

Once the application system model is ready to a point where we have an implementation model, we can use this model in combination with the required libraries and frameworks, and use development tools to create an operative application software system.

```
public class Device extends RegisterableImpl implements
  CopyAble
{
  // copy and name
  public CopyAble copy()
  public void rename(String name)
  public dvDeviceName deviceName()
  public void nameDevice(dvDeviceName deviceName)

  // describe
  public String description()
  public void describe(String description)
  public dvRegisterableDescription registerableDescription()

  // classify
  public dvDeviceClass deviceClass()
  public void classify(dvDeviceClass deviceClass)

  // handle life cycle
  public dvDate purchaseDate()
  public void purchaseOn(dvDate purchaseDate)
  public void signPurchase(dvUserIdentificator signer)
  public dvUserIdentificator purchaseSigner()

  public dvDate upgradeDate()
  public void upgradeOn(dvDate upgradeDate)
  public void signUpgrade(dvUserIdentificator signer)
  public dvUserIdentificator upgradeSigner()

  public dvDeviceState deviceState()
  public boolean isValidDeviceStateSuccessor(dvDeviceState
    deviceState)
  public void changeState(dvDeviceState state)
  public dvI36U18State i36U18State()
}
```

FIGURE 6.5

Domain-specific interface of the device class.

6.6.1 Definition: The Application System

We use the terms *application system* and *software system* interchangeably, ignoring the fact that the software part is normally closely linked to specific hardware components, especially in embedded systems.

An *application system* is composed of new and existing software, reused in the form of libraries, frameworks, and components, including their connections to the system base elements.

An application system represents the domain and software models, and at the same time influencing these models. When used, the application system changes the application domain.

In the T&M approach, an application system is normally designed and installed as a core system with extension levels (see Section 12.7).

Zull-06.qxd 20/8/04 7:37 AM Page 127

6.6.2 Discussion: The Application System

Code generation and CASE tools

We could argue against our understanding of application systems that it is oriented too closely to the traditional “programming” of software. New concepts like the tool-based generation of code from higher models, or the composition of applications from turnkey components, play an increasing role in our approach. However, we still have certain reservations about the generation approaches of modern CASE tools as they are mostly unsuitable for cyclic (or round-trip) software engineering. On the other hand, we have observed in real-world projects that the Enterprise JavaBeans component kit is widely accepted in practice.

Software development influences the application domain

Another aspect of this issue is that the application system influences the application domain and application system *models* even before it is completed, because it evolves as an idea or vision in the early phases of analysis and domain modeling.

The application system that is actually used changes the situation and the work processes in the application domain directly. This should be taken into account, especially in an evolutionary approach where system versions are installed gradually, starting with a core system (see Section 12.7.1). We observed in many projects that even prototypes can cause changes in the application domain. While evaluating, prototypes users immediately start rethinking their work situation and start getting accustomed more or less consciously to changed processes.

6.7 SOFTWARE DEVELOPMENT CONTEXTS

When modeling an application system, we have to take a few important factors into account. These factors influence the structure of the application system and its functionality. We call these factors together the *system development contexts*, which we will explain and discuss in the following sections.

6.7.1 Discussion: Software Development Contexts

We have identified the following *contexts*:

- the *application domain* (e.g., equipment management in a software company);
- *handling and presentation* and its technical implementations (e.g., electronic desktop with icons that the user can manipulate);
- *applied technique* within a development method, including design patterns and a software architecture (e.g., the *flyweight* pattern and its language-specific implementation).

These contexts influence the software construction, depending on the project. We often want to have a specific context in the foreground that affects the structure and dynamics of the system. For example, the design patterns that developers use to discuss their design will determine the implementation of domain classes.

6.7.2 The Application Domain Context

The *application domain* forms the future usage context of the software system under development. This context provides the basis with which we start constructing the *application domain model*, along with the *application system model.*

6.7.3 Discussion: The Application Domain Context

The application domain is the primary context of our software development model. The other contexts have a different influence on modeling and construction.

Regardless of this differentiation, we have to bear in mind that domain modeling determines technical modeling. Note that this is not a one-to-one transformation of all domain terms into classes and all interactions into operations. Nevertheless, the concept model as part of the application domain model can be transferred to classes and class relationships without violating the model or the desired structural similarity.

6.7.4 Applied Technique

The *applied technique* is the context that primarily characterizes the application system model. Based on the architectural concepts, design guidelines, and technological models on this level, and the specific technical components of our system base, the applied technique is an independent dimension of the application system.

Applied technique **includes first of all the generic tools and methods important in software development but not part of the application system itself. This type of applied technique is** ***independent*** **of the domain requirements of the application system under development.**

Applied technique can also flow into the product either directly or indirectly. It is then used to design the application system so that it meets the domain requirements.

We can treat the applied technique as a concrete component or as a technology.

The EMS example

Notice that the applied technique is not totally independent of the domain context. Take our EMS example: when designing an electronic workplace as a desktop, we are not free to select an arbitrary screen size and user interface. Still, the applied technique offers some freedom of design that we can utilize.

The technical concepts we use and their descriptions, such as, programming guidelines, flow through the application system model into our product. In addition, things like the GUI system, the underlying design guidelines, the relational model, and a database, all become part of our application system. These technical concepts and tools should never be seen in total isolation from the domain requirements.

The IDE or the office system used by the software team are applied techniques that are quite independent of the application domain and will not become part of the application system.

THE SYSTEM BASE

If the applied technique for our system is available in the form of real components, then we can use it directly. In the implementation model, these components are then used through interface libraries or frameworks. All these components combined are called the *system base*.

The ***system base*** **includes the set of all interfaces to the (external) technical components used in the implementation model.**

To be able to use a software system in different hardware and software environments, the system base is normally encapsulated in a portability layer. But an explicit system base layer isn't necessary when portability is supported by a large number of commercial APIs (Application Programming Interface), as in Java.

THE TECHNOLOGY

The system base and the technical concepts used are normally connected, because the developer has to have *models* of the applied technique to be able to build a design or implementation model.

> ***Technology*** **denotes the developer's conceptual knowledge of the applied technique. When creating the application system model, this knowledge has to reflect in an explicit or implicit model of the applied technique.**

The EMS example

If we consider this statement in our EMS example, we could implement the registry for devices and room plans by the use of a relational database. However, it is not sufficient simply to install such a database. The software developer has to have an implicit or explicit model of this technique. This is the only way for him or her to identify the need to add mapper classes to the domain classes of his or her model, so that the database can be linked. The same applies to the use of program libraries or frameworks; you can hardly use them meaningfully unless you have an idea about software architectures and design patterns. This knowledge of the underlying concepts behind the technical components is called *technology*.

6.7.5 Handling and Presentation

The third context, *handling and presentation*, is relevant in the development of interactive application software. This context denotes how a software system is presented to the users and how users can manipulate it.

> ***Handling and presentation*** **is the way in which we implement the usage model of an application system.**
>
> - **The usage model of an application system shows in its handling and presentation.**
> - ***Handling*** **defines the ways how you interact with a software system, while its** ***presentation*** **defines how it looks and feels from the users' perspective.**

For a single system functionality (as part of the business logic), there may be more than one type of handling and presentation within an applications system, depending on the workplace type and guiding metaphor. The same also applies to a certain extent to applied technique, which means that given a specific technique you have several options to realize the handling and presentation of a system.

There are separate concepts and models for the handling and presentation of an application system. Concepts like electronic desktop, direct manipulation of graphical icons, and mouse interaction are obvious components of a system's handling and presentation.

For systems based on the T&M approach, handling and presentation is a particularly important context, because these elements clearly show the fundamental concepts of T&M, like structural similarity and application orientation.

6.7.6 Discussion: Handling and Presentation

To combine the different concepts for handling and presentation into a uniform picture, we select a guiding metaphor with matching design metaphors (see Section 3.3). Though a guiding metaphor with its design metaphors does not define a specific usage model for an application system, it gives the developer some orientation for designing the software. This guideline is strengthened by a selection of the matching workplace type. Nevertheless, considerable freedom of decision remains for the design of an application system's handling and presentation.

Handling and presentation is an independent design context

The fact that handling and presentation form an independent context, similar to the application domain and applied technique, is often overlooked. The reason is the matter-of-factness in which we use interactive software today, as well as the uniformity of current workplace systems, at least as far as this aspect is concerned.

However, we should bear in mind that the selection of so-called "middleware" and frontend technologies have a strong influence on how a software system can be manipulated and presented. For example, an Internet-based PC application implemented in HTML and Java Server pages offers very limited interactive feedback to user-input. Though a WAP (wireless application protocol) solution for mobile phones uses an interaction model similar to the PC-based browser solution, the much smaller display size limits your freedom of design.

The EMS example

When designing application systems like the one in our EMS example, interaction with the system is often a major problem. How should we represent the room plan and the available devices? What manipulation can we support for the tools? How would working with the room plan influence the staff list? The answers to these questions may be totally different, even for the same application domain and the same applied technique, but our practical experiences have shown that they will always contribute to the usage quality of an application system.

6.8 CONTEXTS INFLUENCING THE SOFTWARE ARCHITECTURE

We know from previous sections that the individual contexts involved in a software development project are relatively independent so that they can change independently. When preparing an architectural model, we should take these potential changes into account. The idea is to isolate the influence of the different dimensions from the software architecture. According to the principle of structural similarity, every context dimension should be reflected by a set of system components. Changes to one context have to be encapsulated in the components of your application system in such a way that they will not affect other components. This section discusses how contexts may influence your software architecture.

6.8.1 Discussion: How Contexts Influence Your Software Architecture

Contexts change independently

Experience has shown that contexts can change more or less independently. For example, window systems with new functions are introduced to the market; some business requirements in the application domain have changed; users want to see modern interaction forms (e.g., hyperlink navigation in the World Wide Web) implemented in their application system.

The contexts as semi-autonomous dimensions

A closer look reveals that, although the three software development contexts discussed here are autonomous, they are not entirely independent of one another. Accordingly, as soon as there is a change to one context, we have to find out how this change may affect the application system and the other two contexts. To facilitate this job in software projects, we clarify and discuss the costs, problems, and influences of such a change.

The EMS example

Let's see how these issues might look in our EMS example. Assume that some people suggest in the planning phase that we should use Java instead of C++ for our equipment management project. At first, this change appears to affect applied technique and thus only the implementation model. A closer look reveals that we cannot link all existing components in Java, because some proprietary interfaces may not be available. Considering that Java and C++ use different object metamodels (e.g., no multiple inheritance), we have to change the software architecture and the patterns. Switching to current GUI libraries results in additional new handling and presentation opportunities that we should utilize.

In this example, the impact on the domain model is negligible. After all, we want to develop an interactive application system, and Java is suitable for our application domain. If we transfer these considerations to a simplified dimensional model of the three contexts, we get the scenario shown in Figure 6.6.

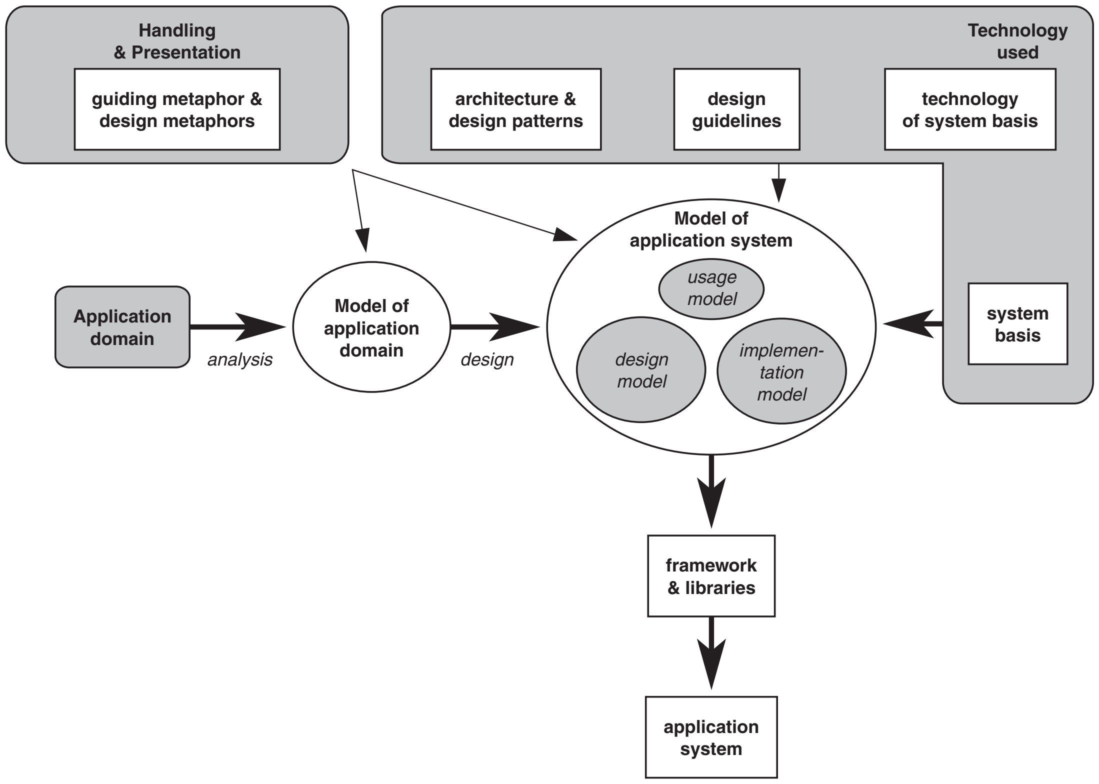


FIGURE 6.6 Software development contexts and models.

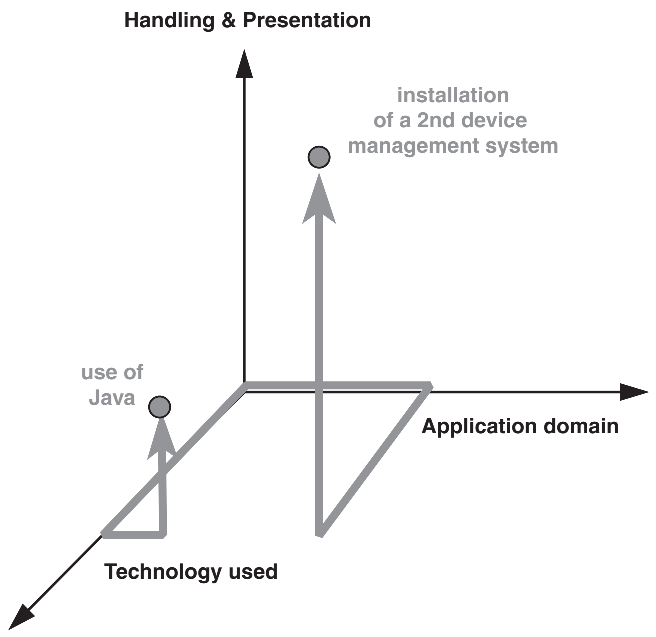


FIGURE 6.7

Examples of influences between the development contexts.

Let's return to our EMS example. Once the equipment management system has been used successfully at the device manager's workplace, the device manager suggests making the system available as well for secretarial workplaces. He argues that this would allow the secretary to enter personnel changes and moving requests, things handled by the secretary, directly into the room plan. Naturally, the change resulting in the application domain has a direct impact on our domain model. If we are to implement this idea, we need at least a simple client-server model and an appropriate implementation. In addition, it turns out that we should implement at least a minimal cooperation model on the handling level to allow the device manager and the secretary to concurrently use the staff list or the room plan. Though this is primarily a domain-specific change, it affects the other contexts, as shown in Figure 6.7. We have to think about extent that these changes should have over the other dimensions, and how this would influence our application system.

6.9 REFERENCES

D. Bäumer, G. Gryczan, R. Knoll, C. Lilienthal, D. Riehle, H. Züllighoven: "Framework Development for Large Systems." *Communications of the* ACM, October 1997, Vol. 40, No. 10, pp. 52–59.

An article on modeling in framework design.

E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*. Reading, Mass.: Addison Wesley, 1995.

Some patterns described in the book are referenced in this chapter.

Zull-06.qxd 20/8/04 7:37 AM Page 133

I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*. Reading, Mass.: Addison-Wesley, 1999.

This book also contains important aspects of software modeling.

A. Krabbel, I. Wetzel, H. Züllighoven: "On the Inevitable Intertwining of Analysis and Design: Developing Systems for Complex Cooperations." In G. van der Veer, A. Henderson, S. Coles (eds.): DIS'97 *Designing Interactive Systems: Processes, Practices, Methods, and Techniques*, Conference Proceedings, Amsterdam, The Netherlands, August 1997, pp. 205–213, 1997.

A more detailed description of our approach to modeling.

J.J. Odell: "Managing Object Complexity, Part I: Abstraction and Generalization; Part II: Composition." *The Journal of Object-Oriented Programming*, September, October 1992. Vol. 5, No. 5 & 6.

An important work on object-oriented modeling.

This page intentionally left *blank*

7

T&M Conceptual Patterns

This chapter builds on the T&M guiding metaphors and design metaphors introduced in Chapter 3 by describing them as conceptual patterns. The sections of this chapter deal with tools and materials, automatons, containers, and the work environment. While Chapter 3 showed how metaphors can help to build a common universe of discourse between developers and other concerned parties in a software project, this chapter takes these everyday metaphors into the realm of software design.

The conceptual patterns discussed in this chapter form the basis both for the technical construction and the usage model of a system, occupying an important intermediate position. Chapter 8 will discuss the important conceptual patterns as *design patterns*, as they support our technical construction of an interactive application system.

By the end of this chapter readers will better understand the basic idea of application-oriented development. You start with relevant and familiar things, terms, and concepts of the application domain. Then, in an interative process, you evolve these elements into a model and implementation of the software system, avoiding conceptual and structural gaps.

7.1 CONCEPTUAL PATTERNS

While design metaphors provide a common point of view and terms to discuss the application domain, conceptual patterns are mainly for the use of developers. Conceptual patterns take the objects and concepts of the application model and help us design our visions of the future system. This means that conceptual patterns are both the means and guidelines at the important connecting point between analysis and design, or actual state and planned state.

T&M conceptual patterns do not describe the details of the internal construction of application systems; these details are described by design patterns (see Chapter 8). Conceptual patterns describe those elements of the system that are visible to users in the usage model; they describe the overall design and architecture from the domain view.

7.1.1 Conceptual Patterns in the Development Process

Within the development process, conceptual patterns come into play when we begin dealing with the domain concepts of the application, and when we have finished identifying the relevant work objects, based on T&M metaphors. Let's look at the situation from the logical view rather than as a temporal project phase.

Setting the scene

- The application domain is identified and the workplaces in each department to be supported by the application software are clear.
- The tasks completed at these workplaces and the way they are completed are clearly understood. We have already gained an initial notion of the workplace types involved (see Section 3.6).
- The materials relevant for these tasks are understood and described.
- The concepts behind these tasks and materials are understood.

Problems addressed by conceptual patterns

In the next step, we have to design the components of the future system and their interaction to technically build the system. At this point, we have design metaphors available for modeling. However, as we know from the introduction of T&M metaphors (see Chapter 3), it is normally not meaningful or possible to directly transfer daily work objects to software components. Conceptual patterns are intended to help us solve the following design problems:

- How can an application system be divided into manageable components? How can tools, materials, automatons, and containers be arranged to represent the domain work environment?
- What is the relationship between the objects and concepts of the current application domain and the components of the application system? Can all things be transferred?
- How are work processes supported by the system components? Which processes should be modeled by the use of automatons? Which processes should be supported by tools?

In the following discussion of conceptual patterns, we will try to find answers to these questions.

7.1.2 The T&M Conceptual Patterns

This section introduces the following conceptual patterns based on design metaphors:

- interrelation of tools and materials (Section 7.3);
- material design (Section 7.4);
- tool design (Section 7.5);
- work environment (Section 7.6);
- container (Section 7.7);
- form (Section 7.8);
- automaton (Section 7.9);
- domain service provider (Section 7.10);
- technical automaton (Section 7.11);
- probe (Section 7.12); and
- adjusting tool (Section 7.13).

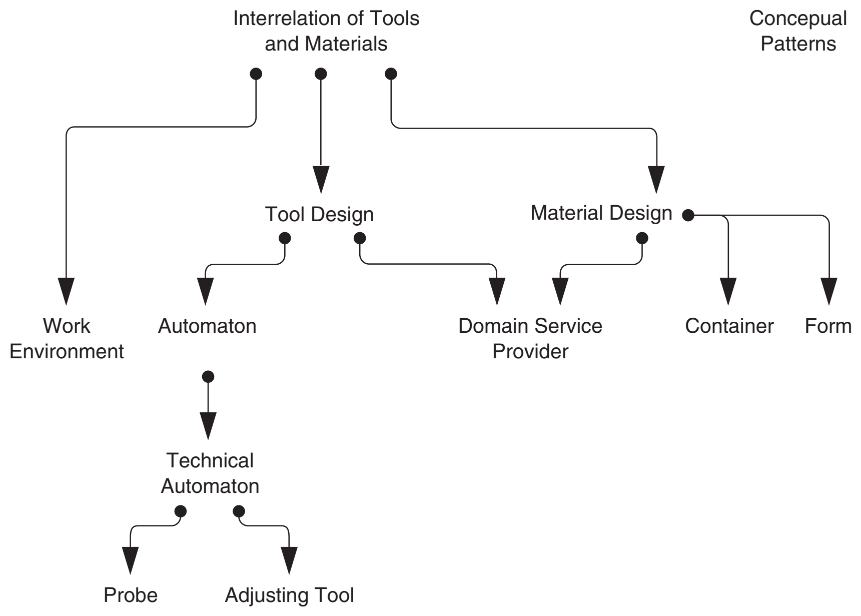


FIGURE 7.1
Hierarchy of T&M conceptual patterns.

Each conceptual pattern is described separately. We take up the general pattern form from Chapter 4.3.10 and modify it due to the character of conceptual patterns.

- Pattern name.
- *Intent*: What is this pattern good for?
- *Problem*: Which problem does the pattern solve?
- *Relate to*: Which other conceptual or design metaphor precedes this pattern?
- *Solution*: The central solution concepts.
- *Background*: What has led us to this pattern? (A section that can be skipped on first reading.)
- *Trade-offs*: Pros and cons of using this pattern.
- *Example*: An example, usually with a short discussion.
- *Rationale*: When should you use this pattern?
- *What next*: Which patterns will be useful next?

We begin this chapter with an overview of the T&M conceptual patterns as a guided tour. The reader will thus understand how the patterns of this chapter belong together.

The patterns discussed in this chapter can be arranged in as a pattern roadmap (see Figure 7.1). We will use Figure 7.1 for a reference as we continue describing each pattern in detail.

7.2 A GUIDED TOUR OF THE T&M CONCEPTUAL PATTERNS

THE INTERRELATION OF TOOLS AND MATERIALS PATTERN

The central pattern is the *interrelation of tools and materials*. It provides the main context for the design metaphor of tools and materials, as introduced in Chapter 3.5, to be moved forward into the system design process. Given the fact that tools and materials are closely related, we will think of them as design guidelines within one conceptual pattern. We will also try to find answers to recurring questions during the design of this basic pattern.

THE MATERIAL DESIGN PATTERN

This pattern focuses on the material design metaphor as introduced in Chapter 3.5.7. It helps us to transfer concrete materials found at the real workplace into materials for the software system. Design guidelines are discussed as well as the misleading designs of materials.

THE TOOL DESIGN PATTERN

The pattern of the tool design transfers the idea introduced by the tool design metaphor in Chapter 3.5.4 into the system design of the application. The important question is, what kind of tool might be the right one within your software application. This is of great importance because in contrast to the material design metaphor, tools cannot be translated directly from the real workplace into the software system. Typically, we design specialized tools to support the users task in an optimal way.

THE WORK ENVIRONMENT PATTERN

Like the tools and materials design metaphor, the metaphor of the work environment (see Section 3.5.10) is also used to design the system. Therefore, this pattern is described to let developers design a location where tools and materials are stored and used within the context of a specific workplace.

THE CONTAINER PATTERN

As we saw in Chapter 3.5.16, the concept of a container is relevant in the application domain and might be useful within the software system as well. This pattern describes how domain containers may be designed for the application under construction, and how this pattern relates to tools and materials.

THE FORM PATTERN

The form pattern is a special way of looking at certain materials of the application domain. We have observed that a large number of materials show behavior similar to that of what we call forms. In contrast to fully fledged materials, they have little material-specific interactions. They offer generic ways of handling, that is, to fill in some values in predefined fields and to read them. We dedicate a special pattern to the design and behavior of forms as a conceptual unit, because it is the most common generic material that can be combined with prefabricated generic tools in application development.

THE AUTOMATON PATTERN

The automaton design metaphor describes the main motivation for introducing automatons in the T&M approach. In this related conceptual pattern we discuss what kind of automatons we typically design in software systems.

THE DOMAIN SERVICE PROVIDER PATTERN

When we go from the design metaphors to the software design using the T&M conceptual patterns, we need to think about the basic structural layout of the application.

Topics like distribution and multichanneling arise. As the technologies behind these issues are often complicated and hard to use, they introduce new kinds of problems that might be relevant from the users' point of view. Therefore, we introduce a matching conceptual pattern that deals with these issues: the domain service provider. Although no direct design metaphor leads to this conceptual pattern, we can think of a service provider as a conceptual unit of pure business logic that provides and handles materials. Tools typically interact with domain service providers.

The Technical Automaton Pattern

The pattern called *technical automaton* was directly derived from the generic *automaton* pattern, which is, in turn, related to *probe* and *adjusting tool*. Both are based on the pattern *relating tools and materials* and the *work environment* pattern.

7.3 THE INTERRELATION OF TOOLS AND MATERIALS PATTERN

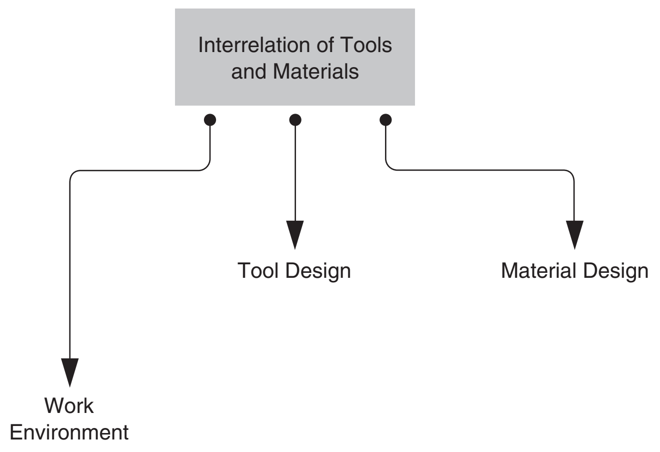


FIGURE 7.2
Interrelation of tools and materials.

Intent

Tools and materials are the central elements of an interactive application system designed by the T&M approach. Given the fact that tools and materials are distinguished by their respective work context, it is important to design them in a complementary way. Figure 7.2 shows how tools and materials interrelate.

Problem

We have to divide an interactive application system into elements so that users can complete their tasks as required in their work situation. The elements should be based on domain motivations and a visible context with the objects already present at the workplace. We have to transfer our tools and materials metaphors to the application system.

Relate To

This pattern is related to the design metaphors *tool design*, *material design*, and *work environment*.

SOLUTION

We model an application system from the user's view in the core as a meaningful collection of materials and tools suitable to complete different tasks at a single workplace.[1] For this purpose, we first define the current tasks to be supported by the future system and the task that will be handled in the same conventional way once the system is in place. The tasks to be supported define the work contexts in which users handle tools to manipulate materials.

Similarity of materials

Our application system should allow for completion of the tasks on hand so that it reflects the way these tasks were handled before the system was in place. This similarity is initially reflected in the application-oriented core of the materials. This means that the important materials in the application domain form the platform from which we start modeling the software materials. We always think of materials as objects in the computer, which have to be manipulated as before.

Tools

In this connection, we use tools that show us the relevant aspects of the materials manipulated and offer us the required work options. A tool should be handy so that we feel comfortable using it to manipulate materials. After all, we want to concentrate on the material rather than on the tool. The tool lets us access and interact with a material, but then remains in the background.

Tools are normally designed for recurring tasks; they are well-suited to complete the tasks at hand. Software tools are not images of real-world tools. They represent domain principles and recurring actions necessary for the completion of tasks. In this respect, they support work forms appropriate for specific materials and support possible interactions in software. Tools may be thought of as the means to support recurring activities, which may differ depending on the material and the situation.

EXAMPLE

The EMS example

Let's return to our EMS example. We have seen that the device manager has used until now a few simple, general-purpose tools, such as a pen, ruler, and eraser. But she has worked on a variety of materials: the room plan, device identification cards, and employee information. It seems sensible to start our design from these work objects and tools familiar in the application domain. The work materials look like good candidates for elements of our usage model for the application. At the same time, it is obvious that we cannot transfer the available tools such as a pen or ruler into one-to-one software elements. Nevertheless, the overall idea looks promising. We want to design an application where the device manager can work with appropriate tools on room plans, device identification cards, and so on. Even the basic interactions seem to be appropriate for our usage model: a tool should support the purchasing of devices; when an employee moves to another workplace there should be a tool to modify the room plan; and so on.

BACKGROUND

Objects (in the sense of things) are used in daily work to complete tasks. At our workplace we arrange these objects so that they are available, depending on the work situation. We normally have no problem classifying most of these objects as either a tool or

1. Chapter 10 will take a closer look at distributed and cooperative work situations.

a material. Depending on the work context, we select appropriate tools suitable to handle materials so that we eventually obtain the desired work result.

The conceptual pattern *interrelation of tools and materials* helps us to take these work contexts as our platform for the software design. The corresponding components of an application system should be understood as tools and materials and designed accordingly. In this attempt, we cannot simply transfer existing tangible tools and materials to software. In fact, this is not even desirable. Tools and materials should be designed to fit the potential of application software, where we have to observe several side conditions.

From analysis to a domain model

An analysis of the application domain results in a domain model. This model represents the relevant work objects and their usage forms or interactions. These interactions result from the way the objects are altered and probed when completing the relevant set of tasks. This defines the domain core of the materials, because we assume that the important tasks will be maintained beyond the introduction of our application system.

Objects and materials

There is an important difference to note when mapping work objects from the real world onto the domain model: when actively completing their tasks, users have to use software tools to be able to manipulate materials. Users cannot simply grasp software materials in their hands. This means that we have to design a tool suitable for each case. In your software system, a material is always the object that is manipulated by a tool (or an automaton) within a work situation.

Tools and materials

A tool can be understood by looking at the ways it is used to manipulate materials. Tools present materials fitting for the work situation, offering users the desired function. A tool represents recurring manipulations of materials. The quality of the work depends largely on how well the tools are suited to produce the results. In turn, the tool quality depends on properties like easy handling, efficiency, and consistency.

This means that tools and materials are closely related, so we always need to deal with them together. The same applies to the software design of tools and materials. A tool can be conceived only by its use to manipulate materials. A material is what we can do by the use of tools.

Using work relationships on design helpers

We can thus exclude the possibility of directly transferring existing objects to software components. For this reason, we have to look at the *concepts* behind the materials existing in the application domain, and the *activities* to complete the set of tasks. They represent the work relationships that we can use as guidelines in the design of tools and materials for an application system.

We further assume that a material can be probed and manipulated in different contexts, thus taking on different states. This corresponds to a normal work situation. Depending on the tasks at hand, users are interested only in specific properties or specific states of a material. This means that a material is normally manipulated by the use of different tools, and that we handle a material in different ways, depending on different states. For example, once a user has purchased a device, it should not be purchased again. On the other hand, a device should be planned, if it is to be positioned on a room plan.

A tool is not suitable for only one specific function. It is generic in that it can be used for different tasks and purposes, that is, it has more than one function. Only a few tools are designed for a special use and a single material. Matured tools can be used to manipulate different materials in a similar way.

Rationale

The concepts behind tools and materials are closely related to each other. We always have to look at them in combination. We have to define a coherent set of tools and materials to support the user in an optimal way in his or her daily tasks.

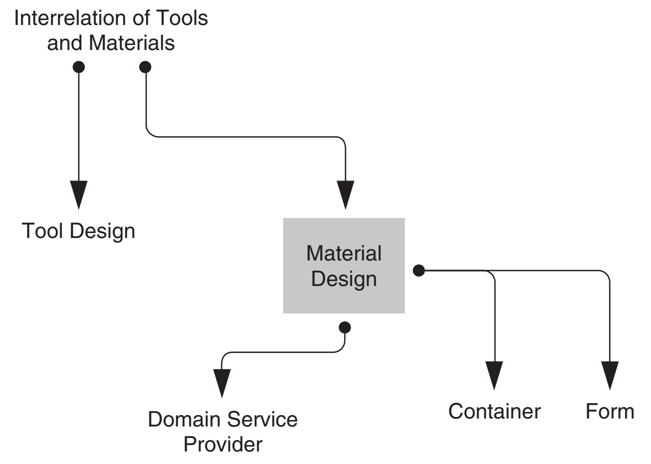
What Next

The patterns for *material design* and for *tool design* take a look at each of the concepts introduced here in more detail.

7.4 THE MATERIAL DESIGN PATTERN

FIGURE 7.3

Material design.



Intent

Materials are both objects and outcomes of work activities. They form the domain basis for the application system. Therefore, we take a closer look at how to choose and design material.

Problem

We have to bridge the gap between the work objects of the application world and the conceptual material units for the application system. Therefore, we have to solve the following design problem:

We want to identify the main material concepts of the application domain that are relevant to the daily tasks of the users. As materials should be conceptual units, we have to choose and design them well regarding the system under construction.

Materials are probed, manipulated, and embedded in other materials. They can take different work states; when working with them, users concentrate on manipulating them.

We have said that we use design metaphors to analyze the materials relevant in the application world. We have also used them to model appropriate usage models. We simply named materials, using names familiar in the application domain. In addition, we defined general concepts in the context of our application domain. These existing materials and general concepts form the basis for designing materials on another level—the material classes.

Relate To

The pattern material design is related to the conceptual pattern *interrelation between tools and materials* and the *tool design metaphor* (Figure 7.3).

Solution

Concentrating on the domain functionality

The T&M approach does not put the presentation and manipulation of materials in the foreground, because they are normally displayed and handled by a tool or an automaton within the usage model. Therefore,

> **We can focus on the domain functionality when designing materials. We identify the relevant functionality by looking at the various tasks, we work with the respective materials.**

For each existing material, we have to check the actual contribution it makes in completing tasks. In many cases, we will find that objects are used solely because they are required by existing (automated or manual) data processing forms. We think that this is the only explanation for why there are so many control lists and change request forms. Such things can often be removed by application-oriented software support.

One of the most important ideas behind the T&M approach is to add *abstract concepts*, represented as materials, to the system, in addition to the obvious things found in the application domain.

We briefly mentioned several examples of such concepts, such as bank accounts or credits, in previous sections. Note that these notions do not exist as concrete objects. You will not find an account in a bank as a "tangible thing." What you will find at the workplace are forms, for example a form to open an account or make a money transfer or deposit, and representations of these forms on the screen. However, bank employees have to handle accounts, and they have clear notions about these accounts, depending on the department and the workplace. This suggests that the account as an abstract concept should be modeled as an independent material rather than as a collection of related objects and representations.

Materials form a conceptual unity

Materials offer more than certain domain interactions, which can be modeled by operations of a class. Materials also represent a conceptual unit, encapsulated in the design. This means that, in our design, we also define the domain integrity of a material to ensure that it will be retained. For this purpose, we encapsulate the internal states of a material class, that is, we hide it from direct access, as opposed to those domain states that are externally visible. Depending on these material states, we can define reliable interactions. Next, we define the protocol for a material class, which ensures the internal domain consistency of a material. We can use state charts (see Section 13.9.5) in this modeling process. The internal consistency is ensured by assertions for all state-changing operations, based on the design-by-contract model (see Section 2.3).

Relationships between materials

Example

The EMS example

In the design of our EMS example, we first analyze the tasks at hand and how they are completed. We have already seen that the device manager handles several materials, including *room plans*, *employees lists*, and *device order forms*. We can elaborate a domain description of how these materials should be handled. We have several concepts, such as, *room*, *device*, *employee*, and *room occupancy*. We also have a pretty clear picture of the work situation and the steps involved: the room plan is edited; employees or devices are moved to different rooms; new devices are purchased; and the device statistics is evaluated.

From all the above, we can derive clear indications for our material design. We could adopt the existing materials and their domain interactions in our design. For example, we could model a class, `Employees`, to accommodate all information relating to a team member's employment contract. It also appears meaningful to reflect about implementing the *room*, *device*, and *room plan* concepts in software.

At this point, we will probably think about a way to represent employees in the room plan, where extensive information about the employment contract is not required. One solution would represent employees by business cards in the room plan. This would reflect a familiar notion for the set of information we want to find about an employee in the room plan.

This solution could be used to work out several domain interactions. For example, the device manager could look at a business card to see whether or not this employee is authorized to occupy a separate room, or whether he or she should share a room with other employees, depending on their positions. In the past, the device manager had to run these checks manually, but we can now use the potential of software and transfer this background check to the material in our design. As an object, the business card can tell us whether or not an employee is eligible for a separate room. This means that we develop domain interactions for a material beyond the material's current properties.

Similarly, we could discuss another class, `Device`, and a device identification card to represent a device object in a room plan.

TRADE-OFFS

We cannot look at each of the materials of an application system in isolation. There are essentially two different relationships between materials: a *contains* relationship and a *use* relationship (see Section 2.1.18).

For example, a room in the room plan of our EMS example can either contain or use devices. Whether a relationship is a *contains* or *use* relationship can be seen only from the domain-specific context. To facilitate this issue, it is often useful to ask whether or not a material would make sense both with and without its "neighboring" material in the application system. As far as our EMS example is concerned, we can say that devices can also be meaningfully handled even when they are not allocated to a room. For this reason, we model rooms that use devices.

Material and interaction

We have said that we can handle a material only provided that we have a tool (or automaton). Nevertheless, we have to bear in mind that a material should also be suitable for interaction. To support this aspect, the material should offer probing operations for all properties that users would like to see in that material. This means that we have to include operations that make a material suitable for interactive handling and presentation in our design. We conclude that the close relation between conventional materials and tools also applies to software materials.

On the other hand, even software materials are independent of tools in computers. The domain interactions are the stable elements in our modeling effort, and we classify and group materials based on the similarity of their interactions.

Material state

The fact that we have primarily discussed a material's interactions does not mean that the *material state* is negligible from the domain view. Structural information is required to be able to implement material states in the first place. We can combine materials from other materials or let them use other materials. However, the state of a material is eventually based on values. Each probing of a material returns state information, determined on the basis of the material's own values. For design purposes, the internal state should be separated from the externally represented state information in our implementation. This information-hiding principle is primarily implemented by the use of domain values (see Section 2.6.5).

DISCUSSION: GENERIC OPERATIONS

Each change to a material is effected by changing structural information. We saw in Section 2.1.10 that a class can define *generic operations*, which can be used by the operations of that class to access the internal structure of an instance of that class. Although this is often intuitive and supported by different development environments, when modeling materials, we pay particular attention to ensure that these generic operations are not made public to represent the domain interface of a material.

Many inexperienced developers tend to design materials as pure value collections. Occasionally, this indicates that the domain was not well understood, particularly if material designs are developed starting from a user interface. Such designs often resemble interactive tools from GUI elements, which present material information and make the material manipulable. What often happens in such designs is that the entire domain functionality and the integrity of materials reside in a tool and its GUI elements (widgets), while the materials themselves degenerate to containers of value lists.

Admittedly, modeling materials based on domain interactions at the beginning of a project is difficult in many cases. These conventional materials are often paper forms, consisting of a series of fields used to write or read values. Section 7.8 describes such forms in detail. Our experience has shown that forms-based materials often develop into material during the course of the development process, that is, when we add domain interactions to their interfaces. For this reason, we should not think of materials and forms as two totally different things.

RATIONALE

Designing materials for the application system is one of the central activities of developing an application-oriented system. Materials form the basis for the entire system. Therefore, materials should be designed in a careful and conceptually meaningful way.

WHAT NEXT

Closely related to materials are the tools to manipulate them in the application system (see Figure 7.3). Therefore the following *tool design* pattern should be read next. Managing large collections of materials is a task that we usually allocate to a domain service provider, as described in Section 7.10.

Special material conceptual patterns are *forms* and *containers*, as described in Sections 7.7 and 7.8.

7.5 THE TOOL DESIGN PATTERN

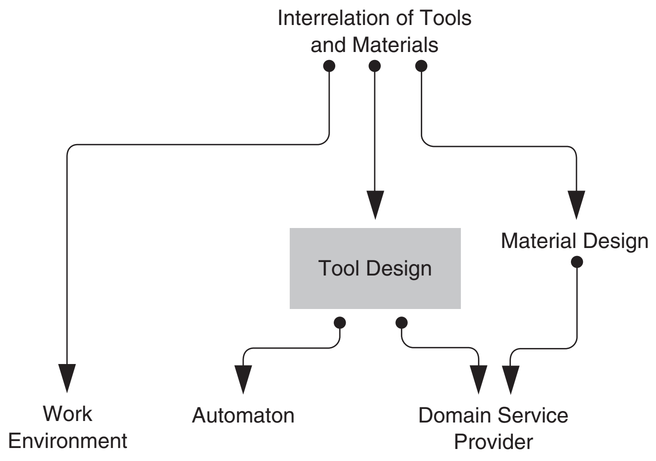


FIGURE 7.4
Tool design.

INTENT

Tools present materials and allow us to probe and alter materials in a tool-specific manner. They are the key elements of an application system and provide the central interface between the user and the system. Therefore, the tool design has a high impact on the usability of the application.

PROBLEM

Tools introduce special requirements to our design, which means that we have to solve two design problems.

First, we have to define the domain contents or functionality of a tool. Second, we have to give the tool a shape appropriate for representing it in software.

Obviously, we cannot solve these two problems independently of one another. Unfortunately, we don't have clear references along the lines of the Bauhaus slogan, "form follows function." So what shape or form results from the domain functionality of a tool for a bank account manager or a device manager?

RELATE TO

This pattern is related to the conceptual pattern *interrelation between tools and materials* and the *material design metaphor* (see Figure 7.4).

SOLUTION

Considering that we are dealing with software as the fundamental "fabric" that our tools will be made of, there are only a few "material-specific" restrictions to be taken into account. This means that, in designing software tools, we basically have to rely on a few good examples of interactive systems and our imagination.

Functionality of a tool

The domain *functionality of a tool* grows from our experience with recurring activities. For each tool, we have to identify all those recurring activities that should

be embodied as a function of that tool. This effort is aimed at providing a tool to simplify but not totally automate these activities. Consequently, tools should support short, interrelated activities that can be implemented by algorithms.

Automation and support

We have to find an answer to the question regarding what the difference is between automating or supporting activities. We can find an answer to this question in the granularity of the tool functions that we modeled. Each function should be atomic from the user's view; that is, it should be such that we can break it down into smaller, easier manageable subsets of the domain functions or activities. Of course, we can determine such a decomposition only on the domain level and not in code lines.

In our EMS example, if a device identification card is moved from one room to another, it is certainly atomic. The same is true in our bank example, when we calculate interest rates based on customer information. In this connection, it is absolutely irrelevant that the icons representing these identification cards are moved by the use of a computer mouse, or that the algorithm for interest calculation is composed of a sequence of instructions.

In contrast, the complete creation of a room plan, including all its rooms and devices and employees, can surely be thought of as automation.

Example

The EMS example

Let's look at these issues in the context of our EMS example, where tool design is no trivial task. We can exclude a simple transfer of the pen to edit a room plan or to fill out a purchase form from the outset, because these actions do not provide substantial support for the device manager's tasks. We know that we can utilize the potentials of software, to design a set of different tools for the device manager to facilitate his activities.

We first study the set of different scenarios to identify the tasks and activities involving the room plan. The device manager has to allocate employees and devices to rooms. For this task, the lists of available employees and devices should be visible. For a tool to be suitable, it should create a new room plan and edit an existing room plan. In addition, it may be useful to design a second tool to maintain the device park. This second tool could then be used to edit and display device information.

This solution appears to be better than creating a single complex tool to manage room plans and devices. However, we aren't done yet. What we also need to support are device procurement processes. The important thing here is to introduce *business transactions*. The state of affairs in a procurement transaction has to be represented in the tool intended for managing devices.

Let's summarize our ideas about the design of a tool for editing room plans. This tool, let's call it *Device Organizer*, should be as unobtrusive as a pen used to add the name of an employee to a list. On the other hand, we would waste the important technical options we have if we only provide for writing on a room plan. At the very least we would expect that when the device manager moves business cards between rooms, the tool should indicate whether or not such an occupancy might conflict with other options. This is a good point to note that although the basic business transactions remain, our software tools and materials offer more functions than their conventional counterparts.

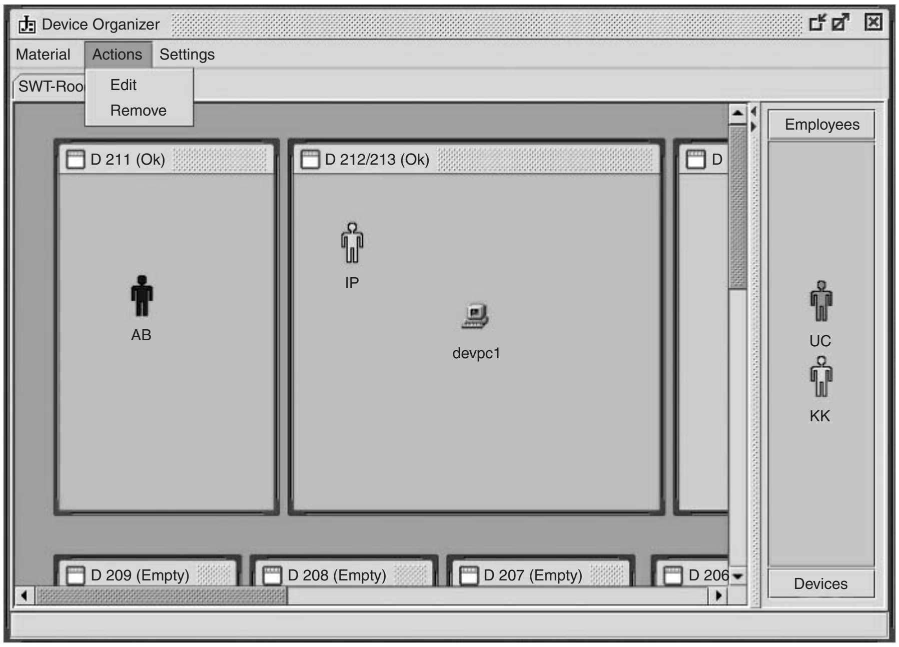


FIGURE 7.5
Designing the Device Organizer Tool.

Figure 7.5 shows an initial draft of our device organizer design with the functionality just described.[2] The right pane on this screen shows the list of employees, and the current room plan is displayed on the left. Note that two rooms of the room plan are already populated with employees and devices. The color and the description of the room show whether the selected occupancy can cause a conflict (red). In this example, the occupancy is alright (green). A mouse is used to manipulate the device organizer. For example, the device manager may select an employee, and a business card icon represents the mouse pointer. The device manager can drag and drop a business card to allocate an employee to a room.

Next, we have to match our tool design with materials. First, we check whether or not the room plan and the employee object already have the probing and altering interactions that we need in the tool. For this purpose, an employee object has to be able to return a business card object, but it also has to return the employee's name on request. Finally, we combine all interactions that a tool like our Device Organizer expects from a material like our Employee object into one aspect, for example, `EditableByOrganizer` (see Section 8.3).

TRADE-OFFS

Tools and processes

We combine meaningful activities to complete a set of related tasks, but tools do not define a number of fixed sequences of functions or processes. Although a tool should normally support specific tasks or subsets of tasks, we cannot define the exact sequences of work steps, because they depend on actual situations.

2. We want to underline again that these GUI examples are schematic sketches of the design concepts and should in no way be seen as paradigms of ergonomic interfaces.

Moreover, we have to bear in mind that a task is normally not completed in one shot, that is, without interruption or intermediate steps. If a tool is to support users' activities, we have to give the user enough freedom to decide the sequence in which they want to call the operations of a tool, and when they need to interrupt their work. This relates to the flexibility of a tool.

Tool state

A user should be able to call an operation of a tool independently of other operations. As for material designs, it is often meaningful to give the tool its own domain working states, so that certain tool operations cannot be executed at any time on a material. It is important to make these states visible at the tool's user interface. These logical dependencies between the operations of a tool can be expressed by means of the design-by-contract concept (Section 2.3) and a state chart (Section 13.9.5) in the software model.

Size of a tool

A tool is normally suited for a specific set of activities and not for everything that may arise in the context of a job. This statement gives us an initial indication of the size of the tools we want to design. Just as different tasks and activities are separated in a job organization, tools are developed for manageable and consistent tasks and activities; in addition, suitable tool collections can be created for the entire set of activities involved in a job. This means that instead of designing extremely complex megatools,

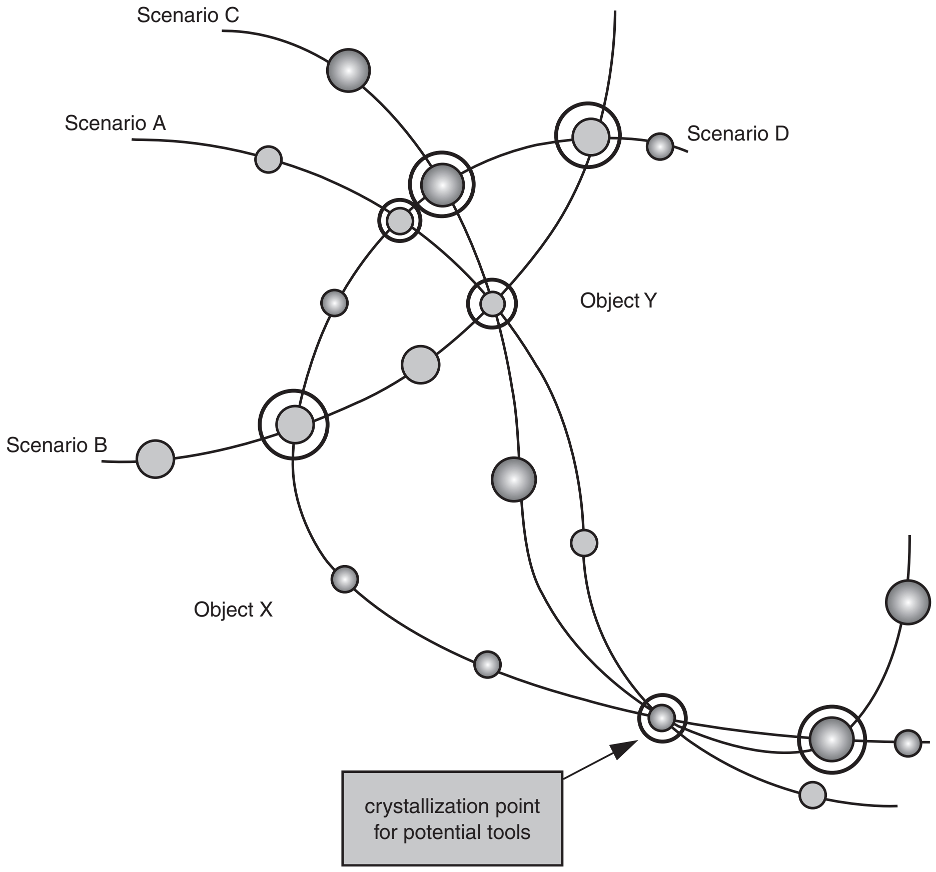


FIGURE 7.6
From scenarios to tools.

we create individual tools that can be easily combined. Again, it is hard to say when a tool is too complex and when it is too simple. All we can say is that tools that offer only one domain function are as poorly designed as tools that cover all requirements of a complex workplace.

Designing tools

How then should we design a tool? To answer this question, let's assume that we have identified all the relevant tasks involved in a workplace. Different scenarios show different processes, in which the same work object may occur. We know that scenarios describe how we handle work objects to complete specific tasks. In this connection, we can often identify similar usage forms, interactions, or elementary activities that are related to single or combined work objects in different scenarios. This "overlapping" of activities and processes in different scenarios form our initial crystallization points for tools. More specifically, we combine similar activities with similar materials and try to find out whether or not we can design a fitting tool for this combination. Figure 7.6 shows an example of such crystallization points.

DISCUSSION: DESIGN CHARACTERISTICS OF TOOLS

Unfortunately, the above crystallization points will not give us any clue about the shape or usage model of a tool. To identify the characteristics of a tool, we have to use a tool metaphor, from which we can derive several design characteristics:

- A tool can always be *identified* as such. It has a *name* and a *graphical representation*. It can be activated by its graphical symbol or icon.
- A tool always shows a *view* of the material currently manipulated and gives immediate *feedback* about changes to this material. A tool never *hides* a material.
- As long as it is actively used, a tool remains *visible*, that is, while manipulating a tool, you can always see the tool you handle. The illusion that the material can be manipulated directly, or without any visible tools, is created only for the purpose of moving and arranging materials on the desktop. The appropriate desktop tool is normally represented only by a mouse pointer.
- A tool supports *several handling modes* to allow flexible manipulation of a material. Normally, this domain functionality has to be explicitly represented, in contrast to traditional manual tools. For this purpose, we use graphical elements, such as buttons or sliders and menu options.
- A tool has a memory and can be adjusted for specific materials or functions, that is, it at least shows the settings for how a material is represented. In addition, we normally use a *status indicator* for tools to display the effects when the user presses buttons or selects menu options, or to display the status as a tool function progresses.

Figure 7.7 shows the schematic design of a simple tool; note that this is not intended to give a layout example.

As mentioned earlier, the way a material is represented and the feedback provided for users as they manipulate materials is very important for tool design. Both the kind of representation and the system feedback depend on the type of tool. When building a tool, we define which state changes of a material it should support and how these states are displayed. We make this decision from the perspective of the users and the activities we want to support. This means that we have to find an answer to the question of what a user expects to see when he or she uses a tool to manipulate a material.

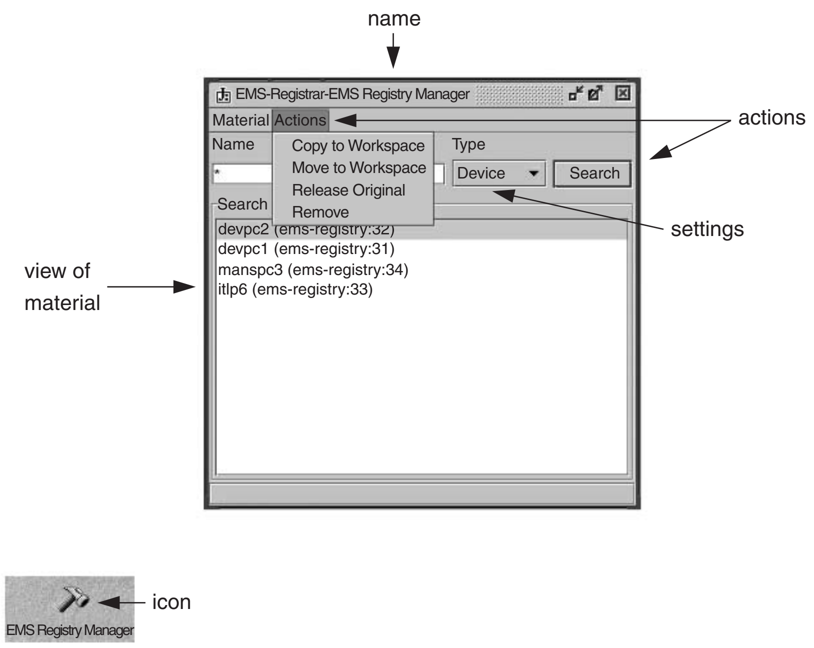


FIGURE 7.7

Schematic design of a tool.

Therefore, it is important to understand the work context for which our future tools are intended.

One frequent problem is to find out how a tool should be *initialized* when it is activated without a specific material. In many cases, users want to select a material, thus starting the allocated tool implicitly (e.g., by double clicking). However, it is sometimes more useful to directly start a tool to ensure maximum flexibility. One solution to this problem is the "empty" representation of such a tool, that is, *without* material. On the other hand, this may confuse users. An alternative solution represents a blank material or selection dialog, showing the materials suitable for that tool.

RATIONALE

A tool is, aside from materials, the second fundamental conceptual pattern for application-oriented software development using the T&M approach. Tools represent the interface used by the user to interact with the materials to complete daily tasks. Normally, tools are designed to match the users' needs in terms of tasks.

WHAT NEXT

The next conceptual pattern, *work environment*, describes the location where tools and materials are combined. In addition further components used by tools are automatons and service providers, which are both described as conceptual patterns later in this chapter.

A more detailed look at the technical construction of tools, as well as complex and compound tools, is provided in the discussion of various design patterns in Chapter 8.

7.6 THE WORK ENVIRONMENT PATTERN

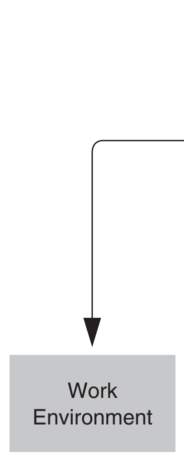


FIGURE 7.8
The *work environment* pattern.

Intent

Tools and materials do not exist in isolation; they are used in a work environment. The work environment represents a spatial notion suitable for a software system and, at the same time, is a conceptual shell for the organization of work.

Problem

We want to identify some kind of a conceptual work environment as part of the application. We thereby want to add a spatial dimension to software systems.

People are used to having a certain location where they can work. These people have to be able to set up their workplaces and work environment at that location. The location should allow them to organize their work, including whatever objects they may need. In this respect, spatial orientation plays an important role. Software systems do not have a physical notion of space. This means that they lack a crucial category that humans use to structure their workplaces.

Relate To

The *work environment* pattern directly relates to the *work environment* design metaphor presented in Chapter 3.5.10 and should be understood on the background of the conceptual pattern *interrelation of tools and materials* (see Figure 7.8).

Solution

We model an explicit work environment that can be personalized, and where tools, materials, and other things required to complete the tasks on hand have their place. In this work environment, we represent a space concept motivated by the application domain, which is also a conceptual unit within the system context.

Requirements of a work environment

A work environment defines the spatial and logical dimensions used to organize tools, materials, and other pieces of equipment. It should meet the following requirements:

- A work environment should implement spatial and logical dimensions similarly to how people perceive spatial organizational forms. This includes a way to manipulate objects without time delay and show the effects immediately. For this purpose, a work environment should support the mechanisms required and manage the components from the moment an application starts until it terminates.
- A good work environment combines those tools, materials, and automatons that are meaningful in dealing with the tasks at hand. It should be possible for organizers or even individual users to personalize the work environment.
- A work environment should explicitly objectify logical order principles. It should maintain consistent conditions between tools and materials.
- The integrity of a work environment should be maintained by regulating access and visibility from the outside.
- If required, a work environment should support and reflect various forms of cooperation.

Example

The EMS example

Let's return to our EMS example for a moment. The work environment of a person who creates room plans appears to have a very simple structure: there is a current room plan, employees, business cards, a registry with devices currently available, and the tools used to create and edit room plans and employee and device lists. The registry holds both current and old room plans, employee lists, and procurement cases. In addition, there is a trash can. All objects are represented in the form of icons on a desktop. Figure 7.9 shows the work environment of the Device Manager in our EMS example.

This work environment is different from that of an employee who deals essentially with the development of the company's framework and just needs to occasionally have a look at the current room plan.

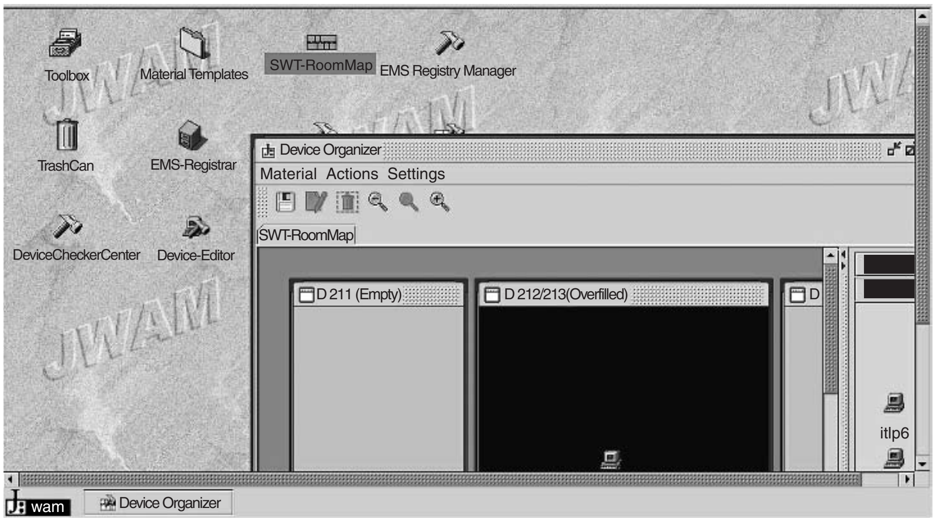


FIGURE 7.9 Work environment of the Device Manager.

Note that elementary order principles have been established: devices can be placed only on the desktop and in device folders, in the registry or in the trash can. Devices can be edited only in the device organizer, but not directly in the room plan. Registries cannot be nested.

The Device Organizer shows whether or not a room plan is consistent. It checks this on the basis of the room plan; however, the room plan allows conflicting occupancy. This tool merely ensures that you can't add devices that are not listed in the device folder to the room plan. However, the Device Organizer allows you to place too many devices in one room. If this happens, it changes the color and label of that room on the plan. The user can see whether or not a room plan has been archived in its current version when he or she tries to file that room plan in the registry. Since the registry must be used in a distributed environment, being an external component, we did not include it in the regular consistency checks of this work environment.

Background: The Work Environment

People use various tools to manipulate various materials to do their tasks. These tools and materials have to have a location, which is normally a personal workplace. But even when there is no individual workplace available, objects and tools needed to complete a job have to be placed and arranged and made available at some location.

The work environment as a location

Our metaphor of a *work environment* describes the location where tools and materials have their place. We have to be able to equip this location for our tasks, order principles, and preferences. In addition to a spatial arrangement, there is often a delimitation and a conceptual order within the work environment.

The work environment as a conceptual unit

This means that the work environment should represent a conceptual unit. We speak of "our" work environment, so we should be able to represent the things belonging to the work of a person or a group. A work environment is generally not a publicly accessible place, so it should be protected against free access from the outside.

Organizing work

An environment helps us organize our work. We place things we want to do first on a stack, while a second stack includes less important documents. Similarly, we arrange the tools we need. The organization of tools, materials, and other objects required to complete tasks is an important part of qualified human work, and it should allow for individual preferences. The characteristics of work organization should be transferred to interactive application systems in an appropriate form.

Location for cooperation

In addition, the work environment is a location where people cooperate. We hold things ready for others, and we keep documents in files and mail trays, pending further handling. All these elements pertaining to the support of cooperative work should be transferred to our application software.

Representation problems

The fundamental categories of location and spatial arrangement cannot be directly reproduced in software. What we have are name spaces, but they are nothing more than an abstract concept, and what's more, they are technically motivated. This is the reason why traditional application systems based on mainframes did not have a spatial concept. Users selected a command from a menu or typed a command in the command line, and the system responded with the output of data in template or tabular form. When hierarchical file systems, for example, the file system of the UNIX operating system, were introduced, it was the first time that users could create their own orders, even though they were represented as abstract trees.

Graphic application systems allow us to model work environments so that they almost correspond to an intuitive usage model. The electronic desktop with its graphic icons was an important step in this direction. Suddenly, people could use a mouse to point and click, that is, arrange and identify things. Nevertheless, graphic user interfaces (GUIs) are still extremely restricted, compared to a normal physical work environment, so that we cannot simply map a physical work environment on the electronic desktop. One of the most important drawbacks is that there is normally no third dimension. As the desktop is too small or the resolution is too low we cannot attempt to provide a "virtual" 3rd dimension. Consequently, we have to find design guidelines that allow us to implement the environment concept and, at the same time, correspond to current software on today's workstations and PCs.

We can safely say that the usual design of electronic desktops lags far behind the possibilities. For example, you will hardly find an equivalent representation of both objects and tools. Instead, you often find the classic Smalltalk interaction model, that is, "object—action," where you select an object and then activate one of a choice of actions. Also, the spatial notion is overly plain in many systems, that is, a desktop with files and other generic containers, in which these files can often be recursively nested in an arbitrary order. The T&M approach tries to overcome these limitations in application software.

TRADE-OFFS

The first design criterion of the T&M approach is: The work environment is the location where people work. Everything that a qualified person does within his or her work environment using tools and materials should immediately show a reaction and become visible to the user. This means that we have to solve a feedback problem, which was briefly mentioned in Section 7.5. As far as the usage model is concerned, only actions referring to objects within a work environment should have immediate effects. The user should be able to see what "inside" and "outside" mean.

The Bank example

Let's look at this issue in our example of an account manager workplace in a bank. The account manager can directly fill out and edit a form because it is available within the work environment. In contrast, opening a new account is something that occurs externally. The account manager will want to use appropriate tools to see whether or not an account has been opened. She should never get the impression that she has an account "on her desktop," because such a model cannot be maintained from the technical and domain aspects.

Implementing the notion of space

When implementing the spatial notion, we often think of the desktop as the best known metaphor. Tools and materials are represented as icons, and they can be directly manipulated, that is, activated and dragged and moved in spatial relationships. The desktop is represented as a workspace, offering a location for tools and materials in a representation appropriate for human perception. When designing such a desktop, we should be careful to distinguish that, although the desktop visualizes the user's work environment, it is not identical with that environment. In modern software design, an electronic desktop may be built either on the basis of widget toolkits or rebuilt from existing system software.

Maintaining the work state

Your expectation for finding your desktop in the morning exactly as you left it on the previous day should also apply to shutting down and starting your application system. The work environment is responsible for maintaining and restoring this state.

The actual maintenance and recovery of the state of tools, materials, and automatons could be delegated to other components (e.g., the registry). From the technical perspective, it would also make sense to let the environment manage the startup and termination of tools or automatons.

The electronic desktop is not the only conceivable form of representing a work environment and objects arranged on it. If there is a potential need to have many materials and different tools available, then structuring the work environment could be a good idea. For example, we have successfully implemented toolboxes and material or document folders in real-world projects.

Work contexts

Another concept that uses multiple *work contexts* takes the above solution a step further. Such work contexts are fully set up by the users and can then be changed. One possible implementation includes the virtual desktop interfaces offered by many UNIX window systems, which create the image of several exchangeable desktop interfaces. Another solution lets the user explicitly start, name, and change a work context for each new business process. In this case, the user, such as a clerk in an insurance company, defines a new work context for a new customer file under the name of that customer. If the user has to interrupt working on this customer case, as when he needs to call another customer, he can put this context—including all tools and materials used so far—aside to talk to the other customer and use the tools and materials relevant for that case. When he is finished, he can put the second context aside and return working with the first one.

Permissible actions for the objects used within a work environment have to be immediate, that is, they can be executed directly and show an immediate effect. This means that the user should be able to see which tools and materials within the work environment can be used without further "access protection." When working in an environment, the user should also be able to directly switch between different sets of tools. In addition, the effects of manipulating a material with one tool should immediately be visible via other tools working on that material.

Implementing the use context

Each work environment refers to a specific use context. This use context is more specific than the general application domain (see Section 9.2.1), because it is critical for the tasks and work situations for which this environment was designed. Accordingly, we assume that there will be different work environments within one application domain that differ mainly in their combination of tools, materials, and automatons. This means that the respective segment of an application domain determines the character of a work environment, and the selected workplace type, depending on the chosen guiding metaphor (see Section 3.3).

Personalizing a work environment

Having different work environments raises the question whether and to what extent they can be personalized. Many application domains require a user to identify and legitimate themselves to the system. In the T&M approach, this often leads to a personalized work environment, which shows users' last or desired combinations of tools and objects when they start the system. In addition, the set of functions available for a tool often depends on the user, based on the user's role rather than on the user. The environment is a suitable solution for both construction tasks. Users log in and out of the system, and they can request the tools they need, based on their authorization status. Once the basic user identification model has been implemented for an environment, then additional concepts, such as user groups for cooperative work (see Chapter 10) can be added.

Implementing order principles

In a traditional work environment, we are careful about where certain things belong and how they fit. We explicitly transfer these order principles to application software. This facilitates dealing with the environment for the user. One simple mechanism is the "cleanup" principle. When we can specify a basic order for a work environment, which can be user-defined at the same time, then we facilitate keeping an overview in complex work environments. This also requires options for the users to be able to save and restore the environment in their orders.

The elementary order principles also say that the objects of a work environment should be known. For example, when a user misplaced a material, they should be able to search the environment for it. Order principles also include lists of materials and tools used within a specific period.

Consistency checks

However, most organizational work beyond the above is done by the use of specific tools rather than direct manipulation within the work environment. For example, documents belonging to a customer file are handled by the use of a customer file tool. We can use such a tool to check complex consistency conditions and represent them so that users can easily understand them. For example, we can use the interplay between a tool and materials to ensure that, once a credit agreement has been signed, it is filed in a different folder, and that changes can be made only to copies of the agreement in another folder.

We know that tools are suitable for certain materials, and vice versa. This means that users should be able to see in their work environment which tools and materials match. Even when the environment technically delegates this to a manager or the tool in question, the usage model of the work environment should show these matches. It is customary to use different graphic icons for this purpose; for example, when a material is moved on a tool, the user should see whether there is a match or not. The same rule applies to materials and containers. We cannot put materials in containers at random. In addition, containers cannot be arbitrarily nested. If we have once chosen to distinguish between files and folders and cabinets, then it wouldn't make sense to allow arbitrary nesting of these things.

Current work object

A sophisticated order principle that can actually be seen as a consistency concept is the *current work object*. The current work object is always motivated by the application domain and local to the environment. For example, it could be a customer currently handled by a bank account manager. Although different tools can currently manipulate different materials, this current work object is known within the environment. To make the current work object known, we could call an explicit operation, such as "set current customer," or use a specific tool, such as "find customer." If there is such a current work object in a work environment, then we could implement operations, such as "transfer current data" in different tools. For example, a forms editor could use these operations to add a customer's name, number, and address to a form without explicit user action.

Case checks

Finally, there are consistency conditions for certain work situations, which cannot be formulated on the level of a single tool or material. For example, a credit agreement is not really completed when the user has filled out and electronically signed a specific software form. For the case to be legally valid, the contract information has to be transferred from the workplace system into the operative database held on a host and accepted there. The work environment can help us turn this case into an appropriate usage model. For example, the work environment could link contract editors, customer files, outgoing mailboxes, and the automaton for host communication such that the user obtains an application-specific view of the state of a contract case and understands when a fully edited contract will become effective. For example, it means that the user can see that the case is complete but not yet legally effective due to transmission problems to the host database.

General principle of consistency checks

This example shows a general principle for consistency checks: a consistency check runs "from the inside to the outside," and it is local to the respective context. For example, the first thing to check is domain information as context-free domain values (see Section 2.6.5). Next, the domain values are checked in their interplay with the material (the credit agreement, in our example) that embeds them. Subsequently, the tools check whether or not a credit agreement can be added to the customer file in the current work situation. The entire work context is then checked by the work environment, which then calls a consistency check in the host. In summary, everything that can be checked within a local context is checked, and all checks beyond this level are delegated upwards. The consequence for a secure usage model is that the user has to be able to reproduce these nested contexts to understand when and where to expect responses. We meet this requirement by representing the set of contexts in the form of tools and materials and by implementing a spatial notion in the work environment.

Implementing an encapsulation

The work environment is always the outer shell, protecting the tools and materials against the outside, that is, these objects are protected against external access and external visibility. The type of work environment determines what is outside and what is inside. The individual workplace is protected against all other workplaces and users, which means that the work environment is a domain-specific interpretation of a name space. The known protection and encapsulation mechanisms apply accordingly, that is, objects of a work environment are not visible or accessible from the outside. From the technical perspective, the consequence is that many workplaces may be implemented sequentially or even as single-process systems.

Regardless of this technical solution, there is generally no access to a material, for instance from a tool in another environment. An explicit action or operation has to be used to move an object from one work environment to another.

Encapsualtion and data exchange

Unless workplaces are designed as pure single-user systems, encapsulation cannot be complete. We then create well-defined outputs and inputs for workplaces and work environments, similar to the export and import interfaces of a module, connecting them to the outside. In many cases, we will use automatons to implement connections to different types of system components, such as hosts. This form of opening up to the outside does not reach beyond a single-user system in the usage model, because a user can cooperate with others only implicitly by exchanging data. An explicitly extended space concept or cooperation model will be discussed in Chapter 10.

Implementing a cooperation

When the exchange of information with other work environments or being able to view other workplaces is both part of the job and an explicit part of the application system, then we have to devise appropriate usage models. In most cases, as a minimum, we will have to implement simple cooperation and coordination mechanisms in application systems.

Such cooperation and coordination concepts that change our notion of the environment include mailboxes or a common registry. Together with the introduction of the spatial metaphor, we extend the quality of our environment concept; that is, we create joint spaces and common work environments. We will not discuss this issue any further at this point, because it goes beyond the primary question about the design of tools and materials of a work environment of interest here, but we will come back to it in Chapter 10.

RATIONALE

The work environment describes a conceptual unit that is directly related to the design metaphor. It represents a physical and logical place. Tools and materials can be used within the context of a work environment.

WHAT NEXT

The *environment* design pattern takes the conceptual pattern of a work environment and details it for the system construction (see Chapter 8.15). Aside from the more detailed technical design patterns, you may want to complete your impression of a work environment by reading the rest of the conceptual patterns in this chapter. In addition to tools and materials, the work environment provides the context for them as well.

7.7 THE CONTAINER PATTERN

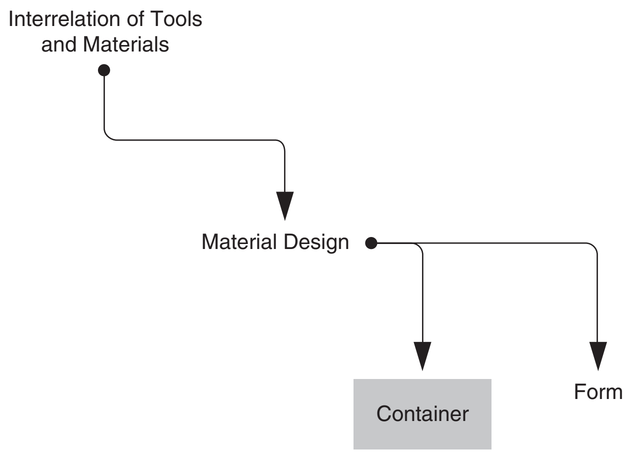


FIGURE 7.10
The Container Pattern.

INTENT

The possibility of representing tools and materials in a software system poses the requirement of storing and ordering these objects.

PROBLEM

Users often have to edit and move collections of similar or related materials. These collections are application-specific, that is, they are not only motivated by the underlying techniques. They have different use forms than traditional technical data structures (e.g., arrays). Therefore,

How can we model collections of related items and their management within an application-oriented approach?

Relate To

The conceptual pattern of a container relates to the *material design metaphor* as well as the *container design metaphor*. It can also be seen as a special variant of the conceptual pattern of a material in this section (see Figure 7.10).

Solution

We want to transfer the metaphor of a container to application software.

Storing materials and collecting things are among the elementary human activities. For this purpose, we often use containers, for which we earlier introduced a design metaphor. It is important to remember that containers are motivated by the application domain and should be thought of as independent objects that are part of the usage model. In the T&M approach, domain containers are positioned at the border between materials and tools or automatons. They are worked or processed, which makes them a material. They manage, probe, control, and create other objects, which takes them near to tools and automatons.

In continuation of our above problem, assume that we want to model domain containers as explicit objects that we want to use to store materials. Containers, similar to other materials, can be manipulated by the use of tools. They have their own interactions to manage and edit stored materials.

Services of a container

In the T&M approach, containers offer the following services:

- Containers can be used to store, collect, and order materials.
- A container can manage the materials it stores and give information about its collection.
- Containers, including their materials, can be moved to other locations, which means that containers support cooperation and coordination.

Example

The EMS example

For our EMS example, Figure 7.6 showed four containers: a toolbox, a template folder, a trash can, and a registry. Note that all four containers rudimentarily resemble traditional containers. For example, the toolbox shows a list of all available tools, which can be activated from within this list. The template folder outputs current forms again and again, never becoming empty.

The trash can is well-known from other electronic desktop applications. Note that the trash can in our example has, in addition to a storage function, a "self-cleaning" function, and that it takes specific protection and use conditions into account. The trash can does not accept protected or active objects without warning the user.

The registry can have a different character, depending on the application system. For example, if the device manager workplace is designed as a single-user workstation, it represents only the location where room plans and device information are held and managed (see Figure 7.11). The user cannot see nor is normally interested in the fact that there is a database or a file system behind it. After all, the other containers should also ensure that their materials are persistent and permanently available.

On the other hand, when the registry is used in a distributed application system, such as connected as well to a secretarial workplace, it has to have additional properties. In this case, several people, as in the device manager and a secretary, will probably access this registry. For this reason, we need a cooperation model (see Section 11.2.3),

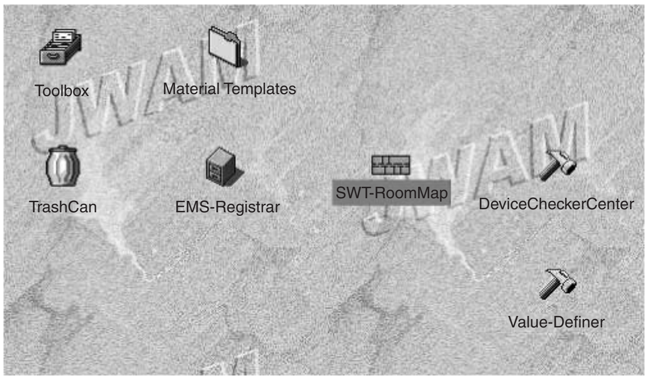


FIGURE 7.11

Containers at the Device Manager workplace.

at least in a rudimentary form, to let users interact on jointly used objects. More specifically, this means that the secretary should be able to see which room plan is current, and whether or not somebody edits another room plan. The device manager should obtain information about changes to room plans effected by the secretary. This coordination can be modeled in a rudimentary cooperation model, using registry status information.

TRADE-OFFS

Container contents

Each collection is always linked to a selection principle. We know what belongs to a collection and what doesn't. From the technical view point, this raises the question of whether or not similar or different objects should be stored in a container. An important contribution to a clean handling of containers, both from the technical and the domain views, is to let the container explicitly check the selection principle, while allowing the user to see whether or not a specific objects "fits" into a container.

The safest solution to this problem would be appropriate typing, which may also be constrained polymorphic (Section 2.1.20). More specifically, we could specify an abstract class that, acting as a supertype, would specify the expected minimal interface for all objects stored in a container. If the programming language you use is not strongly typed, then the selection principle has to be checked at runtime. Eventually, this will let the user see whether or not an object can be dragged and dropped onto a container, based on direct manipulation. (We will discuss a more complex model, using specialized containers, in a moment.)

Polymorphic containers

Polymorphic containers in strongly typed languages are often problematic in that the objects a container holds are known only through their supertype and thus the abstract superclass. This makes sense, as far as storage in containers is concerned. However, we normally want to handle such objects in their specialized forms, once we have retrieved them from a container. This means, for example, that a folder storing documents can also accept letters, if `Document` is a superclass of `Letter`. Unfortunately, the folder will only be able to return objects of the `Document` type to a tool. If that tool wants to edit letters, then we have to use type conversion, or so-called downcasting or reverse assignment.

Order principles

Containers normally represent orders. Only a few conventional containers are conceived as "rummage boxes," even when they occasionally are used as such. To avoid this risk, we explicitly build *orders* in our domain containers. This means that a container knows its order principle and ensures that it is observed. Similarly, objects are stored in their correct place when they are added and this place is maintained for other altering operations as well.

Implementing a management function

In our discussion of design metaphors for container in Section 3.5.18, we already mentioned that domain containers have a management function. In conventional systems, the user of a container manages the objects stored in that container by taking explicit actions, such as manually inserting documents at the right place or keeping index lists. We could implement this expensive and error-prone manual management work as additional internal features for domain containers. This would include tables of contents, consistency checks, and operations on collections. A good example would be a shares portfolio as a bank-specific container that calculates the portfolio value based on current stock rates.

Specialized containers

Taking this management idea a step further and combining it with consistency control, we could think of specialized containers such as a credit application file. Although the content of such a container is heterogeneous, it is defined within narrow limits. We often model such containers using pockets or tabs to clearly show in the usage model that this container manages a specific combination of documents. This definition of contents and internal consistencies allows us to answer a number of questions about the contents on the container level. For example, a credit application file can provide information about the credit contract state, that is, whether or not solvency has been checked and the application has been granted.

Implementing a transport medium

A container is also a useful metaphor where requested materials have to be collected and forwarded for further processing. This is often the case in database applications. For example, a user has to collect a number of customers that meet a specific selection criterion. A conventional database query outputs a set of results. In the object-oriented world, there is no such thing as a “set of results.” Instead, it has to be represented in a domain-specific way. It is often useful to implement a set of results as a collection of similar materials in a known container, such as a folder. This raises another question: How can we model non–object-oriented databases in T&M systems? If we want to use such a database essentially to store objects, we could use an automaton or a service provider for encapsulation (see Section 11.2).

Material transport

So far, we have just touched upon the transport aspect. In fact, it plays a subordinate role in a single workplace. On the other hand, it is convenient to move around an entire material collection in a container. The transport aspect comes into play if we want to move entire collections between different rooms within one environment or between different environments. In this case, a container could be the “natural” transport medium, simulating the way we move folders or containers in the real world (see Sections 10.3.2). These transport containers could also be useful to allow coordination in a collaborative work environment. When you put a file onto your colleague’s desk, then this normally indicates what has to be done and who is in charge.

BACKGROUND: CONTAINERS

From the conceptual and constructive perspectives, you could think of a container as a material. In fact, we generally assume that most complex tools operate on one or more containers to facilitate users to select the desired materials.

On the other hand, containers occupy a special position versus "normal" materials, because containers are always designed to contain other materials, as well as to organize these materials by calling operations on these materials. In real-world projects, this has often led to the problem that containers have been modeled as a kind of tool with corresponding interfaces and an observer mechanism (see Section 8.11.6).

Containers and data structures

In workplace systems developed by the T&M approach, domain containers (e.g., a collection of bank accounts) assume management and aggregate functions, such as managing account lists or summing daily sales figures. This means that they have moved away from the classical data structures and collection classes.

From the developer's view, we have to distinguish containers motivated by the application domain from the technical data structures. For developers, data structures are typical software containers that they handle all the time when programming. Therefore, from the developer's view, containers like lists, trees, and hash tables are "domain-specific" containers used to manage data. In traditional software engineering, data structures have been specified as abstract data types, but they are also thought of as imperative composite data types of a programming language. This situation changed when object-oriented languages introduced active technical containers, offering their own navigational concept (with operations like `first` and `next`) by the use of so-called markers, in addition to encapsulated states by means or access operations. These technical containers are called *collection classes* and were used in many object-oriented applications.

Domain-specific interactions and specializations for containers

In contrast, the T&M approach distinguishes technical containers from domain containers, the latter being independent work objects. For domain containers, we identify the interactions motivated by the tasks of the application domain. In addition, we design *variants* of domain containers, where we are not limited to a generic concept of folders and files.

There is another reason why we have to explicitly model containers. In object-oriented languages, a class doesn't automatically give you a collection of all of its instances. In this respect, object-oriented languages are totally different from a relational database. In fact, one of the main features of the relational model is that you can use a relation, implemented as a table, to also address all pertaining tuples or records. This is something we hardly ever need in an application-oriented model. Instead, we often want to store the different instances of a class in different locations, because this lets us elegantly model different stacks for closed and pending cases or applications and represent them in a usage model.

RATIONALE

Containers are an important concept of our daily work. We often have to deal with a number of materials. We keep them together, order them in separate folders, or move around briefcases containing materials. Therefore, domain containers became an elementary part of application-oriented software development.

WHAT NEXT

In addition to the container pattern, which might be regarded as a special kind of material, another special kind of materials may be found in forms, as discussed in the next section. Also, Chapter 8.11 presents the next step in implementing containers using design pattern *domain container* for this purpose.

7.8 THE FORM PATTERN

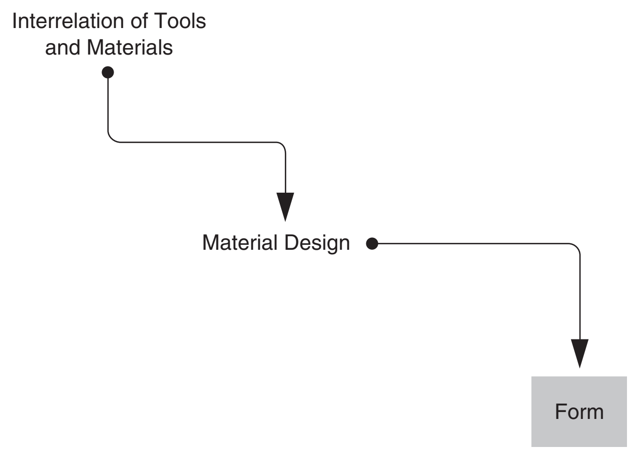


FIGURE 7.12 The *Form* pattern.

Intent

Paper forms and electronic forms play an important role in many application domains, particularly in office environments. In the T&M approach, these forms are a special kind of material.

Problem

When analyzing an application domain, we often find materials for which it is rather difficult to identify an individual domain-specific usage. We normally find that users fill out the fields of a form and read its contents, for example, customer information forms, purchase order forms, or application forms. An attempt to model such objects often ends with a number of generic operations to set and get data. It seems, therefore, to be superfluous to design and implement them as individual materials. In addition to this generic domain-specific usage, forms are also characterized by their layout, which is usually distinguished between fixed and editable fields.

When designing appropriate tools for these forms, we also often find that their domain operations are rather limited or nonexisting, because we look primarily for an electronic version of a pen. In this attempt, we often end up with editors that can be used to set and get information more or less generically. Therefore,

How can we integrate the concept of paper forms into the overall modeling approach of application orientation? Is it sufficient to model forms as materials or do they show characteristics of their own?

Relate To

Forms are a special kind of material. Therefore, they are closely related to the conceptual pattern of material design as well as the material design metaphor (see Figure 7.12).

Solution

We deal with forms as independent conceptual patterns, because they represent special materials across all development phases, from design to construction.

Therefore, we model forms as a special type of material with generic interfaces. Modeling forms as a type of material with generic usage solves many analytical and construction problems. This usage of forms is motivated by the application domain. It corresponds to the conventional way that forms are handled; for example, users fill in application forms or enter information in customer forms. When we transfer this usage to software construction, we can model operations to enter and read data for forms. Our construction becomes more elegant if we implement the write and read operations generically.

In addition to the generic interface that we promote for form-type materials, we keep our design open to ensure that we will be able to further develop forms into more complex domain materials as we gain experience.

Layout of forms

In many application domains, forms are designed in a specific way. When designing electronic forms, we also have to use layout information, such as to present forms on the screen. Also, we normally want to print electronic forms in specific formats, for example, printing form information on printed form templates.

Consistency conditions

Many forms have to meet consistency conditions to ensure that they can be properly edited and filed. Consistency conditions can be defined and checked on two different levels. Many fields of a form take specific value types, such as a date or date ranges (e.g., dates of birth), money, and so on. For this reason, each field of a form should contain a domain value that can then be checked for validity.

In addition, there are consistency conditions that cannot be checked for a single field, that is, without context or a customer form. For example, if we say that a customer's billing address is different from the shipping address, we have to enter both addresses. This consistency condition can be checked only over the entire form.

Form tools

If forms can be modeled generically, we can also implement generic tools. The following tools are frequently used for forms:

- A forms editor can be used for all types of forms.
- A forms viewer displays forms on the screen.
- A forms copier lets you copy and paste the contents of a form in another form.
- A forms printer lets you output forms on a printer, using either plain paper or preprinted form templates.

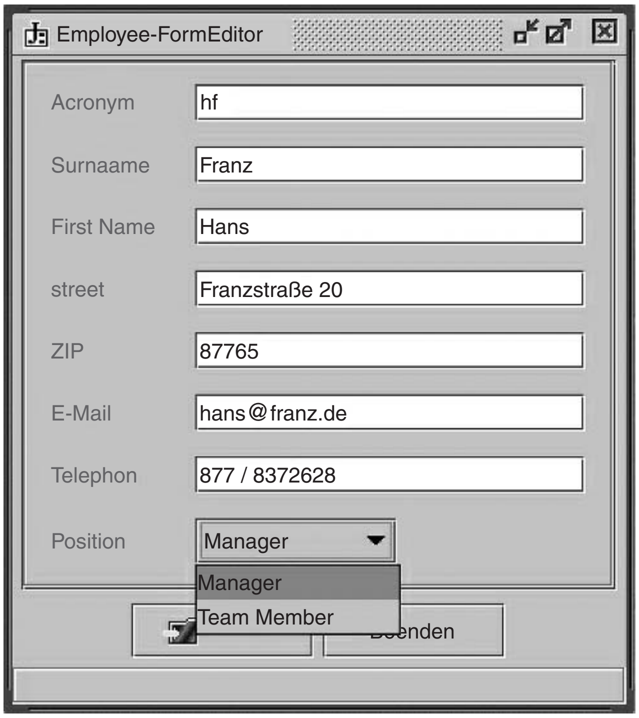
When working with forms, we have to solve a problem related to form versions. More specifically, we have to ensure that we always work with the most recent version. To this end, we can use a forms service. For example, when a user needs to fill out a form, he or she can get a new copy of the form from a central forms server, which always maintains the most recent version.

EXAMPLE

The EMS example

In our EMS example, when a new employee is hired, a form with this employee's personal information is created. In this context, the form has no individual interactions; it merely serves to enter staff information.

Assuming that the personnel department has this information, these employee forms may be edited by a simple forms editor. It would be sufficient to represent and fill the required fields according to their labels. Figure 7.13 shows a generic forms editor that we designed for our EMS example.



Acronym	hf
Surnaame	Franz
First Name	Hans
street	Franzstraße 20
ZIP	87765
E-Mail	hans@franz.de
Telephon	877 / 8372628
Position	Manager

FIGURE 7.13
A generic forms editor.

TRADE-OFFS

Forms and "other" materials

The forms concept can facilitate our analysis and design work considerably. However, we have to beware of the hammer syndrome: "To a man with a hammer, everything looks like a nail" (accredited to Mark Twain). In many cases, the idea of forms will seduce us to think of materials as pure data containers. We often observe this tendency at the beginning of a project, when developers do not have a full understanding of the domain contexts. We have to recognize this risk and avoid it. An application domain consist of forms and nothing else only in very rare cases. We have to constantly ask whether or not the material we study is really used as a form, or whether other domain interactions may be meaningful. We have to carefully study which materials can be modeled as forms. The more domain functionality we allocate to a material, the easier it will be for us to develop it further. Eventually, we will be able to deal consistently with domain changes in one place.

We could begin with a form-based design and gradually develop it further. If, during the course of our project, we find that forms offer additional domain interactions, then we could first model additional domain operations for these forms. For example, we could model the calculation of the total price for a purchase order form. If we can add many such operations to a form, then it is often useful to turn it into a "normal" material. However, it does not necessarily have to replace the form. Both the form and the "normal" material can coexist in the usage model. For example, we could have a purchase order form and a more comprehensive material, such as the purchase order itself.

RATIONALE
Forms are a special kind of material. Forms contain mostly named fields that can be filled out or read. This very simple usage of forms (that we observed in the application world in a similar way) allows us to implement those materials in a generic way.

WHAT NEXT
A specialized design pattern for forms may be found in Chapter 8.12.

7.9 THE AUTOMATON PATTERN

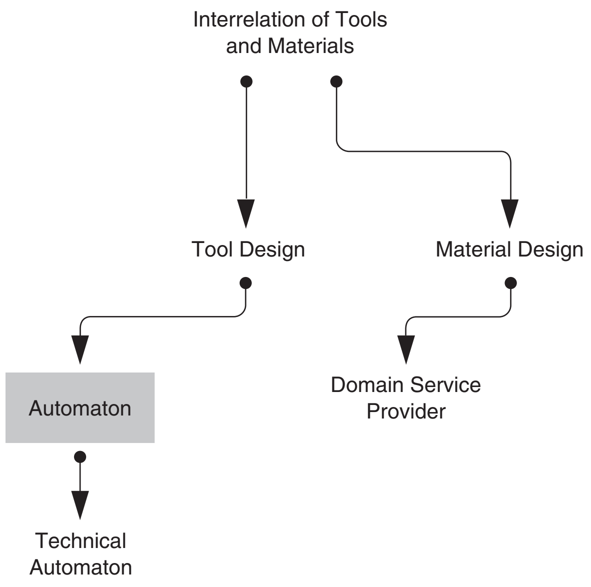


FIGURE 7.14
The *Automaton* pattern.

INTENT
Not every task within an application domain can be completed by the sole use of tools. The *automaton* pattern describes how tedious routine activities or routine work can be supported by means of the T&M approach.

PROBLEM
Tools are used to support human work. When recurring activities have to be done, we call them routing activities or routine work. If such routine activities always produce the same results, we can automate them. If there is only a tool available for the user, then the same tool operations have to be repeated interactively. Such routine work is an ideal candidate for automation. Therefore,

We are looking for a concept to support recurring routine activities with fixed results in an unobstrusive way. The question is what the conditions are and how such routine work could be automated by use of T&M metaphors.

RELATE TO

This pattern relates to the design metaphor of an automaton and can be seen as complementary to the conceptual pattern *tool design*.

SOLUTION

We implement the recurring parts of a job as an automaton. From the user's view, an automaton encapsulates a specific sequence of work steps without relating to any context to produce a well-defined result, without interruption.

We build an automaton so that it operates similarly to tools on materials, where the input that the automaton requires is limited to a small number of settings or initial interactions.

Small automatons

We refer to such automatons as "small automatons." They are normally implemented to take over tedious routine work and can be used at the user's discretion within a work environment that includes tools, materials, and containers. If such automatons are suitable for the materials involved, then they can be reconfigured optionally like tools for use within larger work contexts. In this respect, they cover standard cases. For this purpose, a small number of parameters may be set, and the automatons can then run without manual intervention. This limits their use. In a well-ordered work environment, there should be special tools available for special cases, to maintain some flexibility.

Such automatons can be used for various purposes in an application system, without destroying the basic character of a workplace designed for autonomous activities.

Automatons for functional workplaces

Automatons play an important role for functional workplaces (see Section 3.6.3). If we want to optimally support well-known sequences of activities, we design an automaton that integrates the components required. This integration is possible in different granularities. In a loosely coupled environment, single tools are arranged within the "realm" of an automaton. In this case, the tools maintain their own interaction component, but their interconnection and exchange with materials is controlled by the automaton. In a more strongly integrated environment, the automaton uses the functional component to directly access single tools, itself representing an interaction component. Finally, independent tools can totally disappear while the automaton uses selected subsets of tools for a specific task, but it directly implements important parts of the domain functionality. This means that the workplace has the character of a *control panel*.

The higher the integration, the more the automaton of a functional workplace will become a "large automaton," that is, it actively decides about the sequence of activities involved to complete a well-defined task.

EXAMPLE

In our EMS example, we can see only one domain automaton, which inspects the device park regularly for outdated devices, to then mark them for disposal or upgrading. This automaton could interact, for example, with the in-tray of the device manager's mailing system. Then, whenever the device manager activates her electronic desktop, the automaton will check the device inventory for outdated devices in the background. If appropriate, it will produce a short note and place it in the device manager's inbox.

All other activities are completed by using tools on specific materials.

DISCUSSION: AUTOMATON VERSUS TOOL

One question we frequently ask is what makes the implemented routine of an automaton different from the algorithm of a tool operation. The answer is nothing, at least from a purely technical viewpoint. In fact, both the routine of an automaton and the algorithm of a tool operation are implemented by instructions, having a well-defined effect on an object or returning a well-defined result.

Differences between tool and automaton

The most important difference between a tool and an automaton is the use context and the extent of the implemented domain operation, in particular the following:

- Focused on the result to be achieved, a *tool* is only a means to an end for the user, and its use depends on the work progress and the situation. The tool user has control over each step in the entire work process. In this sense, we expect a software tool not to be "self-operating" but manually directed by a user. For this reason, the tool's functions should be *atomic*, from the user's perspective. Like an elementary action with a manual tool, it wouldn't make sense for a user to divide a tool's operation any further. For example, a "sort" operation for a tool that is used to work with business cards is atomic. The user is not interested in the fact that a sorting algorithm has to be executed to implement the steps of that operation.
- In contrast, the use of an *automaton* means that the control over a defined sequence of activities is passed from a user to an automaton. The user is not interested in actively executing the steps involved in a work process. As long as the standard situation, for which the automaton was built, and the defined result meet the user's expectations, then the user has no interest in intervening in the process.

In the T&M approach, automatons correspond to our notion of technical processes as an automatism. From the user's perspective, a computer runs many automated processes to support the expected functionality, where the user is normally not interested in the details. We use this notion of processes in computers to represent domain processes in a suitable way.

Encapsulating technical interfaces and components

In this sense, we have used automatons to encapsulate non–object-oriented components of a system platform, for example, a relational database (see Section 11.2.5) in early T&M projects. We have used such automatons for two reasons. First, they encapsulate the concrete protocol and the interface. Second, they implement a domain interpretation of a technical component within the T&M usage model. For example, an automaton for host communication can be used both for the concrete transport of operative data from an object-oriented application system, and to use an appropriate tool to inform users whether or not their connection to the host is established and their data are being transferred.

Similarly, we have encapsulated transport mechanisms over a local area network (LAN) for client-server connection, or to implement a mail system in automatons. In these cases, rather than representing the automaton itself, the automaton was "hidden" behind a transport medium, such as an outgoing mailbox or a jointly used registry.

More recently, we have been modeling automatons where an essentially parameterizable domain result is to be produced. Today, we would think of bundling a set of services or business transactions as *domain service providers* (see Section 7.10).

TRADE-OFFS

The implementation of routines and processes in automatons has proven useful in the following domains:

- To *automate tedious human routine activities*, where the automaton replaces recurring steps otherwise performed by the use of tools to manipulate materials.
- To *control processes* at functional workplaces, where the automaton requires minimal user input to produce a specific result using specific materials.
- To *map technical processes* running relatively independently of the application system in embedded systems. In this case, an automaton represents external technical processes or components that have to be controlled or represented by the application system based on hard or soft real-time constraints.

Automating routine activities

The automation of tedious routine activities is an important design notion in the development of interactive application systems based on the T&M approach. From the user's perspective, we first have to identify the routine activities that could be potential candidates for automation. In this respect, many developers act too quickly once they have identified apparently "tedious" recurring steps. To answer this question on solid ground, we should ensure that the following prerequisites are met for designing a small automaton:

Criteria for the automation of routines

- The activities to be automated are in a well-defined and schematic sequence, and there are only a few alternatives that can be determined in advance.
- The set of activities produces a well-defined result, requiring certain identical or similar materials.
- The process can be completed by an automaton once the materials to be handled and the activities have been selected.
- The result can be seamlessly integrated in larger units of activities.

Large automatons

We have introduced the concept of a large automaton in combination with functional workplaces. The autonomy that is characteristic for all our guiding metaphors tells us that we have to observe a certain limit as to how much autonomy such a large automaton should have. The reason is that there will always be a problem when the concrete situation requires a deviation from an automaton's standard run. For a large automaton, we should also combine the benefits of an optimally supported work process with the flexibility required for special cases. For example, a functional workplace could offer other tools, in addition to an automaton, which would then be used for special cases. Unfortunately, this solution does not always fit the usage model because, for example, the user does not have the skills or possibilities to use these tools. In that case, we could at least offer some cooperation options to allow a user to delegate special cases to other workplaces.

There is no general answer to whether or not automatons should have their own interactive user interfaces for resetting or restoring them in case of failure, or whether they are themselves manipulated by adjusting tools. Either solution should be motivated by the underlying technology or the application domain.

RATIONALE

Tedious routine activities or routine work should not be supported by a tool. If the single steps of the tedious routine work can be clearly identified, the user can be supported by means of an automaton.

WHAT NEXT

Matching design patterns for automatons can be found in the next chapter. Additional conceptual patterns for automatons in the embedded world can be found at the end of this chapter. Also the *domain service provider* pattern should be considered to take the relationship between automatons and services into account (cf. Figure 7.14).

7.10 THE DOMAIN SERVICE PROVIDER PATTERN

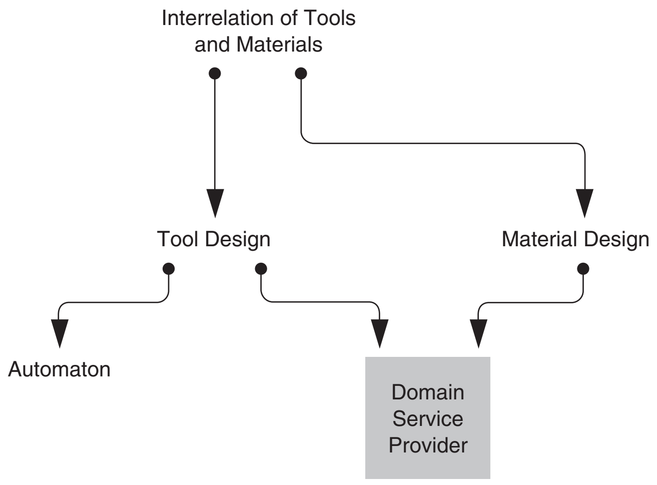


FIGURE 7.15

The *Domain Service Provider* pattern.

INTENT

Modern technologies support new forms for an organization to present its services and products. In addition, they allow an organization to open up new sales and service channels to better support both customers and suppliers.

PROBLEM

We want to use modern technologies, including the Internet, to realize open systems that integrate various applications and legacy systems within a uniform development and architectural concept.

We need a design concept that encapsulates business logic into units independent of interaction mechanisms or front ends, making them available for different channels, technologies, and workplace types.

RELATE TO

This pattern is related to the concepts discussed in the patterns *tool design* and *automaton*. For distributed systems it shows how to extract and encapsulate the business functionality of these two elements (see Figure 7.15).

SOLUTION

We integrate different sales channels and workplace types. We use domain service providers to offer bundled services and allow both users and customers and suppliers to easily handle related products and services.

By our definition, a *domain service provider* is a domain-specific conceptual unit within a large distributed application system. A domain service provider represents business logic in a way that encapsulates the reproducible and interrelated interactions of an application context with the pertaining materials.

Domain service providers are addressed by application components or other domain service providers and respond by supplying their services. To provide a service, they use the materials they manage.

Domain service providers are implemented so that the specific way these services are rendered and which interaction model is used to present the results at a user interface remains open.

If domain service providers are not oriented to a specific presentation and handling or interface technology, then a number of different sales channels could package the services and present them differently, depending on the customer type and technology. Different service providers could each support various workplace types and other services.

EXAMPLE

The Bank example

In our bank example, assume that there is a central service provider, `AccountManagement`. From his or her desktop system, an account manager could request portfolio information for a customer from this service provider. An account manager working in the field, for example, selling a new insurance package offered by the bank, could request account information for the same customer on a laptop application in an ftp (file transfer) session. In addition, this customer uses the `AccountManagement` service provider in a Web browser over the Internet, for example, to view account statements and make payments from the account.

In this example, the domain service provider, `AccountManagement`, supports a domain-specific functionality jointly used by different workplace types and frontend technologies.

BACKGROUND: DOMAIN SERVICE PROVIDERS

Modern technologies have had a major impact on how organizations handle their business as they open up different sales channels to reach customers. Many companies offer their products online, and electronic commerce is regaining its pace, especially in the B2B (business to business) sector. A major change can be observed, particularly in the service industry, where companies expand their activities by adding services on demand, call centers, or mobile field services. Currently, many of these service forms are still found in isolation, somewhat blocking integrated services at customer sites, because the underlying applications are primarily linked on a data level rather than over conceptually higher business transactions. For example, many database applications are based on isolated data for accounts, deposits, contracts, and other data pertaining in some way to a customer, but a general concept *customer* with links to all related entities is missing.

The Internet allows people to compare products and services directly. Potential customers can obtain information about prices and services of different vendors quickly from their homes. The Internet also allows competing vendors, including vendors from different industries, to penetrate the core businesses of organizations. This means that these organizations feel a need to be present in this medium. The reason is the increasing trend that open markets and new technologies dissolve narrow industry boundaries.

For example, many insurance companies have started offering bank services, while banks have expanded into the insurance business, and do-it-yourself companies offer travel packages.

Though this change challenges many traditional organizations, it offers a chance to bundle different services in a few concentrated sales or service points.

The downside is that most currently deployed application systems are too limited to support these new business trends. In addition, many host-based applications have reached the point where they can no longer be maintained or upgraded. While many workplace applications have been replaced by other technologies or implementations, we can see the same happening in an attempt to support new sales channels.

Unfortunately, there are not many integrated applications to support these organizations in their efforts to grab these business opportunities. Domain functionality is developed in many and different ways. If you find some degree of domain-specific integration, it is mostly found on the level of the host database and data-exchange.

Multiple implementations of application logic is the central problem that new technologies must overcome. This problem is further aggravated by a technical problem, because most new applications are implemented as client-server systems. These systems are mostly structured on the basis of the so-called three-layer architecture (see Section 9.3.6). This architecture suggests an integration of the functions or applications involved on the data level. In summary, there is a lack of domain-specific integration.

This chapter introduces domain service providers as a relatively new concept within the T&M approach. Note that the concept of domain service providers complements the other concepts rather than replaces it.

Discussion: Domain Service Provider

To ensure a uniform image toward the customer, coordinated sales activities, and a continuous customer service, we have to integrate all systems used in an application domain on a high domain-specific level. Regardless of their workplaces and tasks, all users should be able to handle their activities, such as business transactions, services, customer files, or product sales and support, in one system. Otherwise, we won't be able to open up application systems for both users and business partners. Domain services represent this high level of cooperation and common access to the system. Based on the principle of structural similarity, they encapsulate domain-specific tasks and objects and their distribution as independent processes and elements.

Domain service providers are "behind" all application system frontends and channels, allowing us to represent a uniform service concept and a uniform customer-oriented image. Note that all these different sales channels complement each other. Regardless of whether customers contact their bank over online banking or in person, they will always see their accounts reflecting past and current bank relationships and obtain end-to-end service.

Discussion: Common Work Object Collections

We often find jointly used collections of objects, such as customer files, in central archives or filing systems. In most application domains, objects are constantly needed at different workplaces. On the functional level, the only important characteristic of this jointly used collection is that it can accept and return materials.

Example

Let's study this issue in the context of an example, using the *JWAM registry*. The JWAM registry represents a central location to manage and register documents as they are filed or retrieved by users, without taking any further specialization or additions into account.

Design guideline

We define the following T&M *design guideline*:

A more or less generic container interface does not qualify a domain-specific collection as a domain service provider, because there is no domain-specific handling of materials.

Discussion: Domain Service Providers and Resources

Materials are often handled and checked in a standardized form. These domain-specific functions are independent of work relationships, where they are used in a specific manner. The important thing to understand is that this is where domain functionality and a collection of materials meet. Such basic services can be integrated either in existing workplace concepts or by accessing other service components.

Even if a central task of a domain service provider consists in permanently storing materials, it is not a pure material collection. In addition, the domain service provider monitors the entire life cycle of a material. This means that we can add application logic to creating, modifying, and deleting operations. The application developer can then concentrate on the domain-specific use of materials, without having to bother about a database interface.

Domain service providers of this type normally offer services like "use material B to do activity A," "check whether material A meets property B," "use properties A, B, C to create a material," or "delete material A."

It appears meaningful for this type of service provider to prevent the direct dispense of copies of managed materials to clients or to accept modified materials from clients. Instead, the service provider offers all changes to materials as operations at its interface, in particular when these changes are highly atomic, so that several clients can change materials concurrently. The reason is that all these dispensed copies may cause a locking problem, or updates may get overwritten before they can be saved.

A domain service provider is *stateful*, because it serves as a storage location for objects representing the domain services of the application. Toward the outside, the service provider changes its state as soon as a material it manages is modified. Independent of the internal construction, a domain service provider appears stateful, because a change to a material causes probing operations on the service to produce different results.

Example

Consider this example. A car rental company manages its car pool. When the market situation for used cars changes, then the depreciation rates for individual car types have to be revised.

Design guideline

We define the following T&M design guideline:

The central domain-related management of material collections is a good argument in favor of a domain service provider, because it would encapsulate a material collection with the domain-specific interactions.

DISCUSSION: DOMAIN SERVICE PROVIDERS AND BUSINESS TRANSACTIONS

Many companies supply their products and services in the form of business cases or transactions, where such a business transaction often "visits" several departments. The drawback of traditional transaction processing is that it is hard to follow up on each of the transactions as they flow between workplaces. Centralized electronic transaction management is normally used to solve this problem. This means that each active transaction is either processed at a defined workplace or stored in a central repository. Finished business transactions can be archived or dissolved in subsets stored in different locations.

A domain service provider can manage information about complex relationships and the implications of business transactions for clients, facilitating their work in the application domain. It encapsulates potential sequences of work steps and responsibilities, representing a case or transaction by a material, such as in the form of routing slips.

Business transactions can be efficiently represented by domain service providers, especially if we add the model of a user session. To activate a transaction, the user can then select the domain service, which will "remember" the user until the transaction is complete. In this case, we could do without representation of a transaction by a material in the usage model.

Service providers that encapsulate business transactions could also be used to store and retrieve materials. This means that the service provider's task would partly overlap with a domain-specific resources management. However, a transaction service provider often manages materials while a transaction is pending, passing control to the resources management service provider when the transaction is closed.

Example

Let's look at a practical example. A new health insurance contract flows across several departments or stations. More specifically, it arrives in the organization in the form of an application. Later, the company requests specific documents from the insured and his or her physician. Eventually, the contract is signed, posted for regular payment of the premium, and the payments by the insured are posted to an account. During the life cycle of this contract, the insured can call and request information about a current business transaction.

Design guideline

We define the following design T&M design guideline:

Business transactions handled by several actors in a standardized from, and which have to be located while they are active, can be easily managed by a domain service provider.

These transactions are often represented as a domain-specific collection of materials, such as folders with routing slips, and they can be managed and processed in a specified manner.

DISCUSSION: DOMAIN SERVICE PROVIDERS AND COLLECTIONS OF FUNCTIONS

Some domain-specific functions or calculations are recurring or repeatedly required in the activities of an organization. Normally, there is an entire collection of interrelated functions. These functions are independent of a workplace. They correspond to the mathematical notion of a function, because they receive all data required as values for call parameters, and return a value as a result of that function.

Domain service providers that are used as collections of functions are normally *stateless* and not based on a session model. Ideally, they have pure value semantics. However, if the calculations produced by a service provider build one on the other, then we often have to store intermediate results or subtotals. In this case, a collection of functions could tend to have a session character that is similar to the service providers used for transaction processing.

Example

Let's look briefly at a practical *example*. Assume that the financial mathematics of a bank can be implemented as a collection of functions, where all functions have to be verified. It appears meaningful to implement a centralized collection of functions across the organization, which can then be used in tools or automatons at different workplaces.

We define the following T&M design guideline:

Design guideline

A centralized collection of functions, to which we pass all parameters as values, and which calculates result values without side effects, does not represent a domain service provider. Rather, it is a mathematical collection of functions that do not encapsulate interactions with materials.

Rationale

The introduced concept of domain service providers has become a central element of our software systems. They provide the means to encapsulate domain-specific logic from the specific frontend technology or workplace type. They complete the picture outlined by tools, materials, and automatons.

What Next

The construction of domain services is described as a design pattern in the next chapter.

7.11 THE TECHNICAL AUTOMATON PATTERN

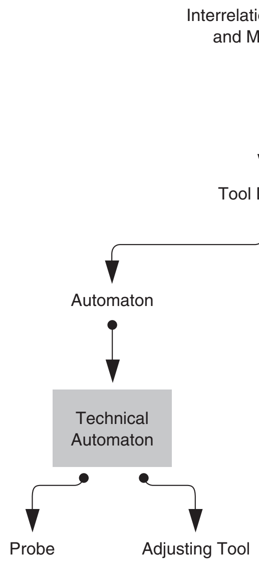


FIGURE 7.16
The *Technical Automaton* pattern.

INTENT

The patterns of automatons and tools described in the previous sections are not sufficient for use in embedded systems because they do not meet all technical requirements. This section describes the additional concepts that are relevant—technical automaton and related elements—in the context of embedded systems.

PROBLEM

So far, we have limited the concept of an automaton to reactive devices that are activated by user interaction, such as the device park checker automaton described earlier. Technical automatons, on the other hand, in our application system model must include components that can be active without user intervention. This special property of embedded systems has to be considered both in design and construction.

What we need is a concept to control an embedded software system. This application component has to be able to control or represent technical components under hard or soft real-time conditions.

RELATE TO

This pattern is directly related to the concepts discussed in the *automaton* pattern (see Figure 7.16).

SOLUTION

We design a *technical automaton* that maps domain-specific states of (real) technical devices to integrate them into the application system model. It informs other components of the application system about state changes and represents an additional source for events.

Technical automatons represent a conceptual pattern for real, physical devices, such as a lab system, a telephony system, or any other machine. Naturally, technical automatons are also real, but they are a model implemented in software that replaces physical devices. Technical automatons are a more specific variant of the generic notion of an *automaton* used in the T&M approach for use in embedded application systems.

We encapsulate the characteristics of embedded systems in technical automatons. In an embedded application system, a technical automaton represents an additional source of events, which can trigger actions on an equal rank with user-enabled actions.

DISCUSSION

The state of the technical context is extremely important for embedded application systems, so it must be explicitly modeled. This state model has a purely conceptual nature. It reflects the domain view that a software developer has of the technical configuration, that is, the embedded hardware components. The state model includes all aspects of the technical context that the developer deemed relevant for the design and construction of the application system. This means that this state model can be compared with the domain model when we develop interactive application systems (see Section 6.4).

From the purely technical perspective, the state model of a context contains mainly the interfaces available between hardware and software (e.g., readers or barcode scanners), and the events and state changes of that technical context, which are visible at these interfaces. In a medical lab, for example, this could include sample trays or sample flasks (see Section 11.4.2). In contrast, in the state model we normally abstract from the technical details of hardware components.

Embedded application systems are used to operate and monitor or control technical equipment. The state model of such an equipment is the *equipment model*.

The *equipment model* of an embedded application system is the software developer's abstract view of the embedded hardware configuration. At runtime, the equipment model maps the state of the application system's actual technical context, that is, the state of the equipment to be operated. From the technical perspective, the equipment model includes the interfaces between hardware and software, and the events and state changes of the equipment visible at these interfaces. The equipment model is a conceptual model and is not implemented in software.

Obviously, we cannot analyze and create the equipment model and the application system model independently of one another, because they interact at runtime.

BACKGROUND

Representing technical processes and components

Technical devices in combination with application software are increasingly used outside the manufacturing industry, for example, analytical automatons in medical laboratories or call centers in telephony systems. These processes or technical components are usually characterized by the fact that they are used under continuous operation and real-time requirements, widely independent of individual workplaces. For example, the telephony system behind a call center or automatons in a medical lab should ideally be available all the time and at an optimum throughput of materials, such as calls or lab samples.

At the same time, there are normally workplaces where these devices are controlled and that request services or information about these devices. For example, a call center may have functional workplaces for the marketing staff and an expert workplace for the group manager who supervises and controls the call workplaces. Or the lab workplaces in a medical lab may be used to monitor analytical automatons and run special analyses.

Automatons handle technical processes

Real-world projects have shown that it is often meaningful to encapsulate the technical processes or components in independent automatons. These automatons differ from the automatons described earlier in that they basically have to be modeled as distributed or concurrent systems. They are independently active in a high availability rate, and the only way to address them is through asynchronous communication mechanisms. At first, this concept of a so-called "technical automaton" appears to conflict with the reactive character of workplaces developed by the T&M approach. To refresh your memory, these workplaces depend on their users' activities. However, the simple registry presented as an example in Section 7.10 represents a component that can be modified by other users. At this point, we have to consider a second characteristic. The technical components of an embedded system exist "outside" of our application software.

In fact, they are generally independent hardware and software systems, using sensors and actuators to interact within their environment. This effect on the environment is not within reach of the "direct access" of our application software. This is reflected in the fact that the application software would not be able to detect or influence that a lab sample flask tipped over or that a phone handset is off the hook.

Telephony example

Let's look at an example of a complex technical automaton for a telephony system. Obviously, a device of this telephony system changes its state when it receives an incoming call. We can describe this status change by implementing a "ringing" state for a phone extension. Accordingly, the "telephony system" automaton, that is, the control model we implemented in the software, also has to change its state. To better understand this process, imagine this telephony system in an interactive application software. The "ringing" state would have to be displayed by an interaction component that, in turn, is represented to the user by a suitable icon. This means that we have two equal-ranking sources of events: the user and the telephony system automaton.

WHAT NEXT

The subsequent patterns *Probe* and *Adjusting Tool* complement this pattern.

7.12 THE PROBE PATTERN

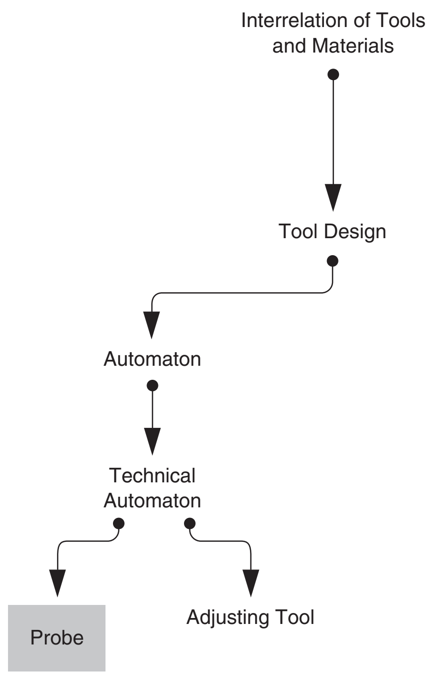


FIGURE 7.17 The Probe Pattern.

PROBLEM

Clients of a technical automaton are not always interested in the entire state of an automaton. In many application cases, it is sufficient to know a partial state of the technical automaton. This is also reflected in our design principle that the model of an application system should include only states motivated by the application domain. In addition, we should bear the software-specific rule in mind that unnecessary communication traffic between the technical automaton and its clients would slow down the system's runtime or performance.

We want to design a unit related to a technical automaton where the client of an automaton can select an interesting section of the state space mapped by the automaton. We want to integrate this unit so that it will interfere as little as possible with the automaton.

Relate To

This pattern is directly related to the concepts discussed in the *technical automaton* pattern (see Figure 7.17).

Background

The domain-specific states of a technical automaton can be grouped into two categories. Characteristic properties of an automaton that never change fall into the first category. These properties may be read directly at the automaton. The domain-specific states of an automaton that change in regular or irregular intervals fall into the second category; these are variable reading values.

Unfortunately, it contradicts the very idea of a technical automaton to interfere with its operation to read variable reading values. Instead, we want to see it operating without interruption.

We are familiar with this kind of determining reading value from technical systems, where we normally use probes. Like sensors, probes sense or detect physical conditions for reading purposes, without interfering in such conditions. We take this concept and, using it as a metaphor, transfer it to the T&M approach. In doing this, we show clearly that probes exist only on the application software level, and that, unlike sensors, they are never part of a physical equipment.

Solution

We design a *probe* as a component that can determine a measurable value of a technical automaton with high accuracy. A probe can be set so that it reads and updates this value in specific time intervals and with specified tolerances.

We attach probes to a technical automaton to read domain values (see Section 2.6.5) based on typed reading values specified in the automaton, and make these values available in the application system. In the design of our technical automaton, we will specify the types of probes that can be attached. The automaton can be queried for available probes.

Discussion

In the T&M approach, a probe is an object that reads a defined reading value type from an automaton and returns these values in specific intervals, for example, in domain-specific intervals, or when specific values change.

A probe allows us to represent single reading values, relevant for the application domain, from an automaton. A probe can aggregate the states of a physical device and use them to calculate domain values. The automaton knows the basic requirements of the probe. Accordingly, it returns a result to the specified probe according to its state changes. The probe is loosely coupled to its clients, and informs these clients when new values are available.

Rationale

Technical automatons with probes are a useful conceptual pattern to design embedded application systems. Technical automatons and probes form a conceptual unit. This unit represents a source of events in the application system.

What Next

The subsequent pattern *Adjusting Tool* complements this pattern. Technical constructions can be found in the related design pattern of Section 8.13.

7.13 THE ADJUSTING TOOL PATTERN

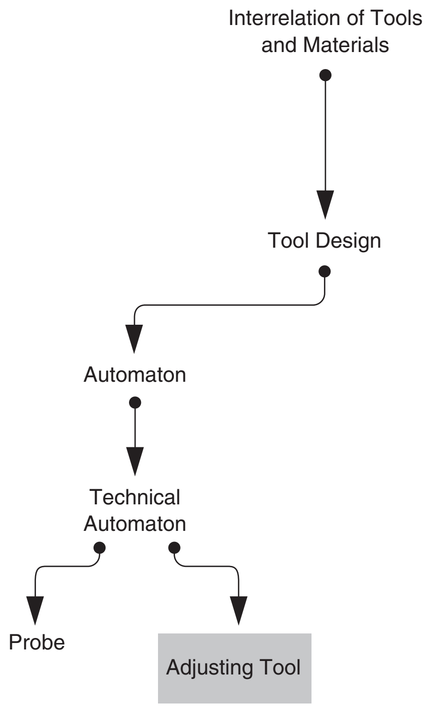


FIGURE 7.18
The *Adjusting Tool* pattern.

Problem

With technical automatons and probes, we have useful concepts to port data from embedded systems to a T&M environment. What we also need is a tool to be able to modify a technical system.

The components of an embedded system have to be set in regular intervals, in addition to other tasks such as control and maintenance work. However, these technical components are normally not installed in the immediate neighborhood of the workplaces supported by the underlying application system. What we need are interactive tools in the work environment to be able to do these setting, control, and maintenance jobs within an embedded application system.

Relate To

This pattern is directly related to the concepts discussed in the *technical automaton* and the *probe* patterns (see Figure 7.18).

Background

Though automatons *can* have an interactive component, it is not always *possible* in technical automatons. The technical device or equipment, represented by an automaton, may be in a location outside the work environment in which this automaton is used. In many cases, security reasons may require that the technical automaton, which controls a technical device and maps it to the application model, runs on a separate computer, independently of the application system. Nevertheless, we need a way to modify it from within the application system.

We observed in many real-world projects that all of the automatons used in these projects had to be set in more or less frequent intervals. For example, the technical automatons behind a telephony system supporting workplaces in a call center have to be continually maintained. A user has to be able to configure this automaton to the number and distribution of incoming calls over the call center staff.

In out search for a suitable concept, we oriented ourselves to the daily work with "real" equipment and technical devices, which are normally maintained and set by qualified engineers in regular intervals. The maintenance staff normally uses special tools to set various parameters for these technical devices.

Solution

We design an *adjusting tool*, which is a special tool used to maintain and control technical automatons. Based on domain-specific motivation, it shows a segment of the state of one or more technical automatons, allowing us to set parameters for these automatons.

Basically, a technical automaton can have a domain interface for that purpose. The adjusting tools interact between the user and the technical automaton. The user can use these tools to visualize and change the parameters of a technical automaton.

Discussion

To be able to represent the states of an automaton, we connect the adjusting tool to the automaton, for instance, by the use of a probe, so that it can be fed with state information. We can use such an adjusting tool to specify both the type of information and the interval in which these state descriptions should be returned. Depending on the application domain, we can display a subset of the states of an automaton or the entire range of domain states. In addition, we cannot generally specify how state information should be updated in the adjusting tool. Based on typical requirements, state information is refreshed upon each change of the automaton's state or in fixed time intervals.

Thus far in our real-world projects, we have rarely seen a case where a user operates the adjusting tool explicitly to request state information.

Modifying the parameters of a technical automaton requires a slightly different concept than what was described earlier for the design of tools and materials. In interactive application systems, the user normally has full control over everything that he or she does with tools and materials. In contrast, technical automatons have a certain "life of their own," because they map independently running technical devices.

For this reason, adjusting tools cannot directly access a technical device; instead, they send requests to the technical automaton. How or to what extent the automaton will then be able to change state as a result of such a request, or whether it rejects a requested state change, should be based on domain and software requirements.

Rationale

Technical automatons with probes and adjusting tools are a useful conceptual pattern to design embedded application systems. Adjusting tools frequently will be used together with probes; they manage this source of events in the application system.

What Next

Technical constructions may be found in the related design patterns of Section 8.13.

This page intentionally left *blank*

8

T&M Design Patterns

8.1 INTRODUCTION

Chapter 7 described T&M conceptual patterns. In this chapter, we introduce their matching T&M design patterns. These build on the conceptual patterns and are a major constructive step toward actual software implementation. The major portion of these design patterns deals with technically designing and implementing tools and materials. In addition, we introduce patterns that complement the core elements of our approach. This chapter discusses the following concepts, which have been introduced in previous chapters:

- tools and materials;
- tool construction and composition;
- domain values;
- containers;
- forms;
- automatons;
- domain services; and
- work environments.

Each concept is described as one or more separate patterns (cf. Figure 8.1). We take up the pattern structure from Chapter 7 and complement it with a few items based on the character of design patterns:

- Pattern name:
- *Intent*: What is this pattern good for?
- *Relate to*: Which other conceptual or design patterns precede this pattern?
- *Problem*: Which problem does the pattern solve?
- *Solution*: The central solution concepts.
- *Schema*: The elements of the pattern and their interrelation.
- *Background*: What has led us to this pattern? (A section that can be skipped on first reading.)
- *Trade-offs*: Pros and cons using this pattern

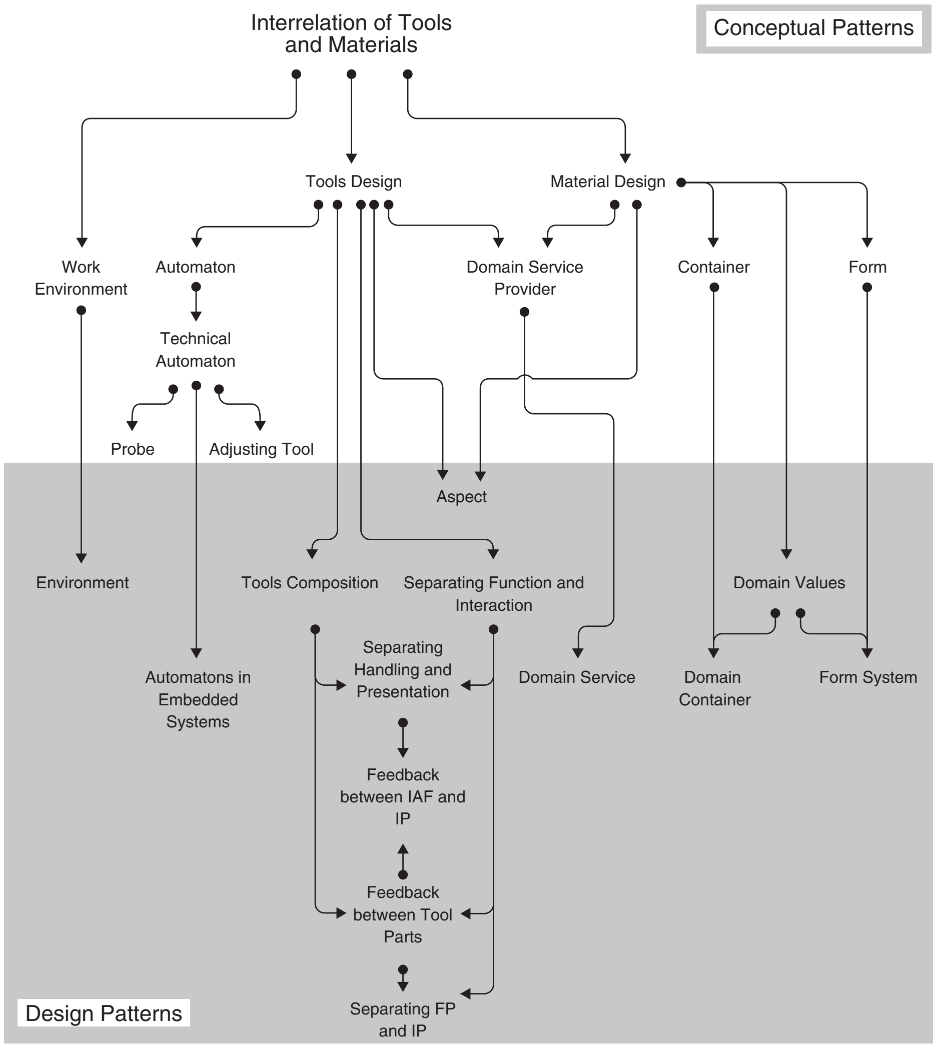


FIGURE 8.1

Hierarchy of T&M design patterns.

- *Example*: An example, usually with a short discussion.
- *Construction part*: How can this pattern be implemented? Here we use short examples from different programming languages (Java, C++, Smalltalk).
- *What next*: Which patterns will be useful next?

We begin this chapter with an overview of the T&M design patterns as a guided tour. The reader will thus understand how the patterns of this chapter belong together. In the final section we close with a discussion of the details of implementing the patterns discussed in our JWAM framework. This section should serve both as another example showing the interplay of the patterns and as a sketch of how to build a framework that supports the constructive ideas of the T&M approach.

8.2 A GUIDED TOUR OF THE T&M DESIGN PATTERNS

Based on the conceptual patterns discussed in Chapter 7, this section describes T&M design patterns. We will now show how conceptual patterns and design patterns are related. We then discuss which design patterns belong together. In this way we present something like a roadmap, with comments for the reader, that continues the guided tour to the T&M patterns.

The following guided tour is based on the T&M design patterns illustrated in Figure 8.1.

The Aspect Pattern (Section 8.3)

The central conceptual pattern is called *interrelation of tools and materials*. It was introduced in Chapter 7.3 and shows how tools and materials complement each other in the human work process. This chapter starts with a design pattern called *aspect* that shows how the two components match.[1] The aspect pattern explicitly represents the interplay between tools and materials. We will discuss two construction parts, aspect classes and interfaces, in some detail, because they are suitable for most cases. In addition, we will briefly describe a number of alternatives for special cases.

The Separating Function and Interaction Pattern (Section 8.4)

This pattern is directly related to the conceptual pattern *tool design* (Section 7.5). Designing and implementing tools is a major challenge to software developers as here everything comes together: domain functionality, handling and presentation that is based on a clear usage model, and the implemention of interactive graphic components.

In order to deal with the complexity of this task, the separation of concerns is the prime aim. This is addressed at a general design level by the pattern *separating function and interaction*. It shows a proven method to implement the two fundamental characteristics of a tool as two separate modeling and construction units. This pattern is based on the idea of dividing tools internally into a function and an interactive part (IP). A similar form of this separation appears in the Model View Controller concept well-known in Smalltalk systems.

The Tool Composition Pattern (Section 8.5)

Similar to the pattern *separating function and interaction*, the pattern *tool composition* directly relates to the conceptual pattern *tool design* (Section 7.5). It addresses the problem of scaling up tools by showing how we can build simple tools and then how we can combine them to build complex tools. In this connection, we will think of tools as components that can be implemented internally in different ways. Three construction parts give detailed guidelines on how to solve various construction problems. Several other patterns introduced in this chapter ensure that tools may be combined

1. In this context, we use "aspect" as a separate concept, not related to "aspect-oriented programming."

into complex tools even when we implement the single tools differently. This includes the following patterns:

- Feedback between Tool Parts
- Separating Handling and Presentation
- Feedback between Interaction Forms and IP

THE FEEDBACK BETWEEN TOOL PARTS PATTERN (SECTION 8.6)

This pattern directly follows the patterns *tool composition* and *separating function and interaction*. With respect to tool composition and separation of function and interaction, we have to answer the question how the single parts involved should interact. The so-called reaction patterns play an important role in answering this question. They control part of the communication between dependent components and are described in the pattern *feedback between tool parts*, which is used, for example, in the pattern *separating FP and IP* (Section 8.7). The central idea is again the separation of concerns: this time we make sure that two or more construction units, which interact, know as little of each other as possible.

THE SEPARATING HANDLING AND PRESENTATION PATTERN (SECTION 8.8)

Regardless of whether or not tools are divided into functional and interactive parts, we can use another pattern called *separating handling and presentation*. Together with the concept of interaction forms, we will introduce a proven and easy-to-implement concept that allows us to abstract GUI design from the concrete user interface and GUI toolkit. Some readers may think that this abstraction is superfluous in the days of Java and Swing, but our project experience shows that independence from a specific GUI library will pay in the long run. Of course, the trade-offs between direct GUI access and another abstraction layer must be discussed.

THE FEEDBACK BETWEEN INTERACTION FORMS AND INTERACTION PART PATTERN (SECTION 8.9)

This pattern is the logical consequence of the pattern called *separating handling and presentation*. Once we have separated the concrete GUI classes from the general interaction of a tool, we need to bring together these two construction units. Showing another application for the reaction pattern, this pattern describes a *feedback between IAF and IP*. It concludes the patterns explicitly addressing tool construction.

THE DOMAIN VALUES PATTERN (SECTION 8.10)

As materials objectify pure domain functionality, little can be said about the concrete construction of materials. So, although there is a conceptual pattern called *material design*, there is no directly matched design pattern called *material construction*. On the construction level, there is, however, a fundamental T&M concept called *domain values*. We addressed that concept in Section 2.6.5, when we discussed the fundamental elements and characteristics of the object-oriented programming model.

Domain values form the basic components that we use to compose materials. The pattern *domain values* introduces different implementation techniques as construction parts. In addition, we will show how special domain-value interaction forms (see also *separating handling and presentation* in Section 8.8) or special domain-value widgets can be used to simplify the development of sophisticate user interfaces.

The Domain Container Pattern (Section 8.11)

We have introduced *containers* as a conceptual pattern directly related to materials. On the constructive level the pattern *domain containers* shows how containers can be implemented. Although we have said that it is of little value to have a dedicated design pattern for materials in general, we must solve special design problems for domain containers. Today, developing technical containers, sometimes still called dynamic data structures, is no longer a problem. There are good standard libraries for all common programming languages. For this reason, we deal with this issue only briefly, simply explain the relationship between domain and technical containers.

The Form System Pattern (Section 8.12)

On the conceptual level, we have established forms as another materials category, in addition to containers. Forms are suitable to implement simple materials with only few domain-specific operations. The directly related design pattern *form system* shows how we can implement forms and how they can be combined with a small set of standard tools to speed up our application development process. Forms cannot exist “independently” from materials. For this reason, we will also explain how forms can evolve toward more complex materials with their own domain functionality.

The Automatons in Embedded Systems Pattern (Section 8.13)

The concept pattern *automatons* discusses the main characteristics of automatons. We have already said that an automaton usually runs in the background. This reduces an automaton almost entirely to the implementation of a domain or technical algorithm, and there is little left for a general design pattern.

The matter is different with technical automatons in a distributed environment. Therefore we discuss the design and construction of software within embedded application systems as the design pattern *automatons in embedded systems*. More specifically, we will explain how automatons can be built to fit into the T&M approach, taking asynchronism and process distribution into account.

The Domain Services Pattern (Section 8.14)

The concept pattern *domain service provider* deals with collections of materials and the domain interactions with these collections. As a direct logical consequence, the design pattern *domain service* offers a way to represent the notion of service providers as software units, resulting particularly in a possibility for flexible configuration of client-server systems. For example, we could support totally different workplace types with frontends (rich clients or thin clients on PC, Webtop clients, laptop clients, etc.) and easily connect back-end systems (e.g., host, SAP). We will use the *domain service* pattern to show how domain service providers may be implemented to realize these ideas.

The Environment Pattern (Section 8.15)

The concept pattern *work environment* discusses the characteristics of the workplaces and their environments according to the T&M approach. It defines the boundaries of a workplace and embeds tools and materials in this workplace. In addition, it allows the workplaces of this environment to communicate with other environments. The design pattern *environment* explains how we can implement this generic concept and what features we would normally need.

Zull-08.qxd 31/8/04 2:35 PM Page 190

USING THE T&M DESIGN PATTERNS FOR THE JWAM FRAMEWORK (SECTION 8.16)

The last section of this chapter describes how design patterns interact. Based on our EMS example, we show how the design problems discussed in this chapter can be solved by a framework approach. We show the general elements of an interactive T&M application that we have moved into our JWAM framework in Java. Besides seeing the interrelation of the T&M design patterns, the reader can get an idea about designing a generic application framework that is domain-independent but still provides a good basis for implementing T&M applications.

The References at the end of this chapter includes useful sources for further study of the material discussed in this chapter.

8.3 THE ASPECT PATTERN

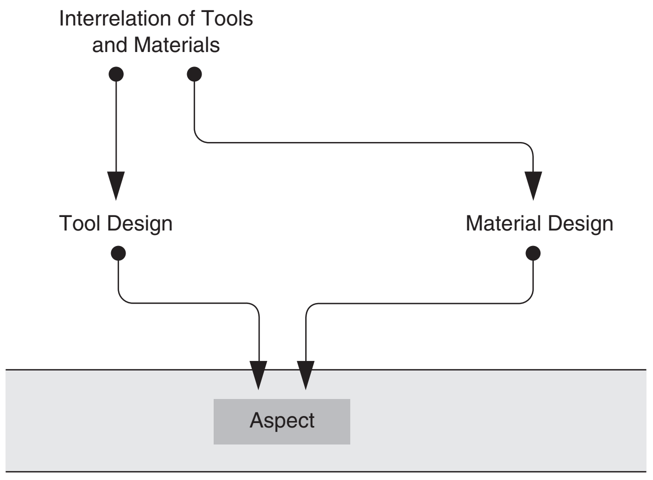


FIGURE 8.2
The Aspect pattern.

INTENT

Users handle tools to work with materials in completing their tasks. Tools and materials should not be combined arbitrarily, because a tool matches some materials but not others, and vice versa. Aspects link matching tools and materials.

PROBLEM

To identify a construction principle, ensure that tools and appropriate materials match.

This principle should apply to the design, before we get to writing code. It appears to be meaningful to identify more than a single material for a tool (and vice versa). The domain-specific relations between tools and materials should emerge in the technical design.

Relate To

The *aspect* pattern is related to the conceptual pattern *interrelation between tools and materials* (see Figure 8.2).

Solution

We represent the interplay between tools and materials by the use of aspects. Such an *aspect* objectifies the interrelated use of a tool and the materials it operates on.

The characteristics of an aspect are:

- It represents both the syntactic interface and as much of the semantics of the relation as possible.
- It ensures that a material meets the requirements specified in the aspect for its use by a tool.
- In our construction, we pay careful attention that the tool uses its allocated materials exclusively over the interface specified in the aspect.
- An aspect explicitly solves the technical problem of combining different protocols or interfaces in our design.

To better understand the solution, consider the example shown in Figure 8.3. In this example, a `Tool` uses an `Aspect`. This `Aspect` specifies the services that a

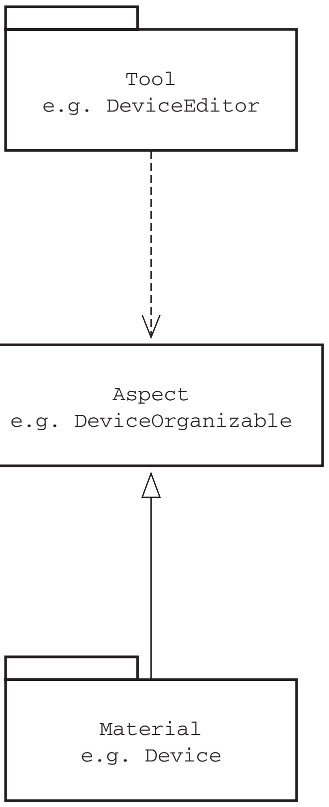


FIGURE 8.3
Pattern schema to bind tools and materials.

`Material` has to provide for a `Tool`. The `Material` meets the interface promised by the `Aspect` and the behavior defined in the interface. Note that we are not saying at this point that the material always must inherit from the aspect, because several construction approaches for the pattern will use inheritance between `Material` and `Aspect` while others will not, depend on the programming language we use.

Example

Let's return to our EMS example for a moment to better understand how we can use an aspect to bind tools and materials (see Figure 8.4).

In the EMS example, `DeviceOrganizer` is a tool that uses materials, including the `RoomPlan` material. However, our `DeviceOrganizer` needs only part of the features offered by `RoomPlan`. The `DeviceOrganizer` is responsible for allocating `Devices` and `Employees` to rooms, giving the user an overview of the distribution of devices and persons over the rooms. Figure 8.5 shows the interface of a material, `RoomPlan`, and the aspect `DeviceOrganizable`, representing that part of the interface the `DeviceOrganizer` is interested in.

Background: Matching Tools and Materials

The way tools and materials interrelate within a work process is extremely important for understanding human work. Handicraft traditions and underlying standards ensure that tools match materials. The correct handling of tools and materials is part of every crafts person's training. Standards (e.g., the ISO standards) specify the appropriate match of many tools and materials, such as screwdrivers, screws, and nuts. Unfortunately, there are very few such solid traditions and standards for software tools and materials. One of these few examples is the CORBA-IDL interface definition for services. For this reason, we have to ensure in our construction that software tools and materials match and that they can be combined.

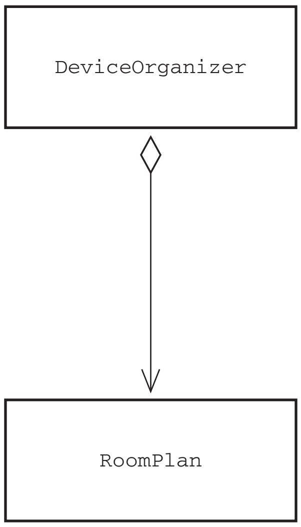


FIGURE 8.4 Using the Device Organizer tool to edit the Room Plan Material.

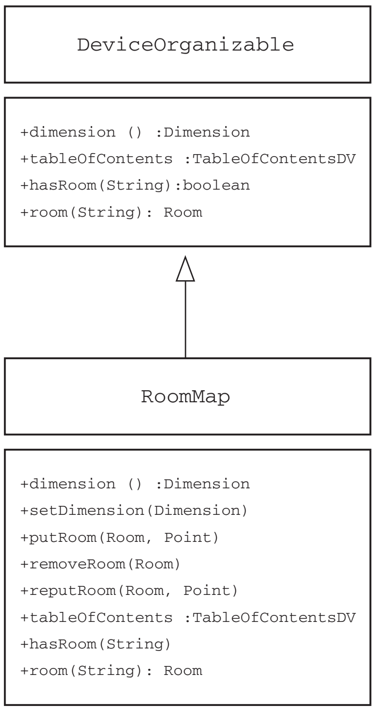


FIGURE 8.5

Material and aspect interfaces.

If we think of tools and materials as components for our application systems, then this match means that we need appropriate interfaces. Based on the usage model, users are supposed to use tools to work with materials; we obviously have to add an interface to a material, so that it can be used by a tool.

Component models and their interfaces

Most component models currently available define generic component interfaces, including a set of operations that can be used to request actual domain-specific operations at runtime. These approaches are too generic for the purpose of matching tools and materials. What we want is an explicit domain-specific abstraction that describes the match of tools and materials. This also tells us explicitly which conditions a material has to meet to be suitable for a tool.

A single tool for different materials

Our conceptual pattern *interrelation of tools and materials* (see Section 7.3) shows that a tool should ideally be suitable for different materials. This means that a tool's interface to a material is narrower or more abstract than the full interface of a specific material. If no explicit interface between a tool and a material is specified, then the question is what segment of a material's interface will be used by a specific tool. This information cannot be easily derived from the design, because the interface that a tool expects cannot simply be extracted from the material; it is determined by that tool's requirements. Unless we introduce additional components to the design, then analyzing the program text of that tool is the only way to determine the interface between a tool and a material.

Different tools for a single material

This basic problem becomes more serious if several tools use a single material. The concrete interface of a material becomes "wider," while the reference between tools and materials drifts further apart. If we want to introduce new materials to an existing application system, we must solve the following problem: To be able to handle the new material using existing tools, we have to identify the interface that materials should have. Unfortunately, we won't find this information either in the tool nor in other materials used by that tool.

Trade-Offs

A material is responsible for providing a set of coordinated operations that a tool can use to fulfill a specific task. For this purpose, an aspect should specify features beyond the purely syntactic interface and describe the behavior of materials.

Thanks to these features, aspects are also suitable to define the relationship between automatons and the materials these automatons require. This means that automatons and tools can be equally combined with materials.

Specifying the behavior of aspects

The means available in object-oriented programming languages to specify the behavior of aspects are limited. We want to use an aspect at least to *specify* an interface. We want to define the interface as a type to ensure that its requirements are met, which is basically supported by a static type concept available in all languages. In statically typed languages, you can use an abstract class to implement an aspect. Our use of inheritance to link tools and materials is based on our experience with practical projects, where we used C++ and Eiffel. Considering that Java and similar languages offer a named interface concept, these languages are even better for this kind of construction.

Aspects and single inheritance

In contrast, we will have to deal with certain problems when trying to use a type to implement aspects in languages that know only single inheritance and no interfaces. If we want to use classes to implement aspects, and if there is no one-to-one relationship between tools and materials, then a material has to inherit from several aspect classes. For example, Smalltalk knows only single inheritance, and the fact that Smalltalk does not support static type checking represents an additional problem. In that case, we have to test the interfaces of materials for compliance either at runtime, or we check and change the program text in the course of our development process.

Aspects and materials

We can also use aspects to specify the behavior of materials in different "strengths." For example, one solution uses aspect classes with abstract implementations and appropriate hook operations. If we do not want abstract implementations for the specification of behavior, then we could use the contract model described in Section 2.3, at least in part, to define the conditions for operation calls and legal states for a material. Note that these solutions require the use of class types to implement aspects; otherwise, all that remains is the pure interface test.

Construction approaches for aspects

When discussing Java and its way of defining interfaces, we have to deal with the following questions. Should we use classes to define aspects at all? Or would it be better to define aspects as named interfaces?

Rationale

If you have to design and implement complex or generic tools using more than one type of material, then you should explore the different constructions realizing the aspect pattern. For simple tools or those working on one material, subtyping of the used material will do.

What Next

There are no specific T&M design patterns for materials as they implement "pure" application logic.

8.3.1 Construction Part: Using Inheritance or Interfaces to Implement Aspects

If you use a class to model an aspect, then a material can inherit this aspect directly. This approach normally uses inheritance such that all operations of an aspect class are visible at the material's interface.

In contrast, languages like Java let you use a language construct directly to define named interfaces. In fact, this is necessary in Java, because the language does not support multiple inheritance between classes.

Note that there is a real type-subtype relationship between an aspect and a material, because a material inherits and implements the full interface of an aspect. In all places where you need an aspect type, you can use any object of a material class that is conforming to the type of the aspect.

Example

The EMS example

Let's look at an aspect interface in the context of our EMS example (see Figure 8.6). The aspect interface shows the operations of a material expected by the `DeviceOrganizer`.

Trade-Offs

Aspect classes allow you to specify the aspect interface between a tool and its material in an abstract class. This means that, although aspect classes can be used as types, you cannot use them to directly create objects, as when using interfaces in Java. This construction should be used so that tool classes never operate directly on a material class; instead, they use abstract aspect classes (see Figure 8.7).

If classes implement aspects, then the material classes inherit from these aspect classes and implement the features they inherited. Using polymorphism, tool classes call operations implemented in materials.

Aspects and multiple inheritance

Materials are normally not used by a single tool. Figure 8.7 shows that doing so would cause a material to inherit from several aspect classes. However, a tool can also operate on several materials. This means that the entire interface of a material includes the individual interfaces of each aspect class. In summary, this case requires multiple inheritance.

Note that the schema shown in Figure 8.7 is simplified, as each tool uses one aspect class to access one material. In the real world, complex tools would normally use more than one material, and thus more than one aspect. Also, it is customary to combine elementary aspect classes with more complex ones by the use of multiple inheritance.

Aspect classes and standard implementations

By modeling aspects in classes, we can define standard implementations for operations, which complies with the basic idea of aspects. In fact, the more accurately the behavior of materials is described, the more probable it is that a tool will be able to

```
public interface DeviceOrganizable
{
public Dimension dimension ();
public TableOfContentsDV tableOfContents();
public boolean hasRoom(String name);
public Room room (String name);
}
```

FIGURE 8.6
Example of an aspect interface.

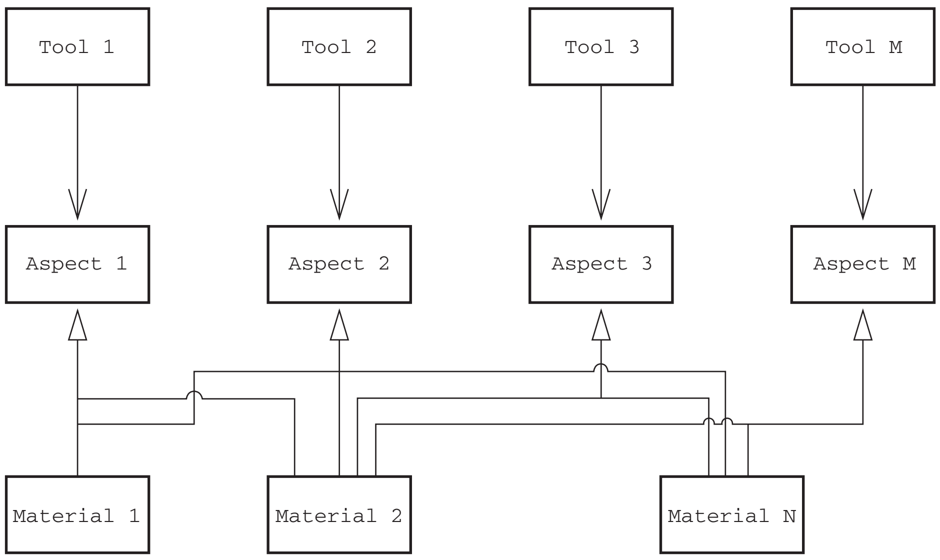


FIGURE 8.7
Tools use aspect classes to operate on materials.

work on a material, both in terms of syntax and semantics. In addition, these standard implementations allow us to model the material state that a tool requires early on, that is, in the aspect class. Material classes can override the defined operations or add attributes to expand a state.

However, a construction that also defines attributes in aspect classes should be handled carefully. The reason is that such a construction conflicts with the idea that aspects are interfaces, as well as with the object-oriented design principle that abstract classes should be lightweight. As a consequence, all material classes we derive are "burdened" with these attributes. Even if this may be justified in the original design of an application system it does not necessarily mean that future material classes under an aspect class should actually be defined with these attributes.

If we use interfaces to implement aspects, as in Java, then the pattern schema would look like the one in Figure 8.8.

In this case, however, we could not provide for a standard implementation in the aspects. This limitation caused by the use of interfaces is offset by the fact that we will not erroneously define attributes for our aspects that would burden all derived material classes. In addition, the problem of many tools using many materials is solved, because Java supports multiple inheritance for interfaces.

CONSEQUENCES OF STATIC ASPECT TYPING

One major benefit of static typing by means of an aspect class or an aspect interface is that the compliance of a material with an aspect is checked at translation compile time, which is like asking: "Does the material implement the type defined in the aspect?" In addition, it ensures that a tool can use "its" materials only under that aspect, which is like asking: "Does a tool exclusively use materials of that aspect type?" In a class-based solution, we can use standard implementations or template methods to specify behavior. And finally, if we are careful, we can implement a state for materials at that early stage.

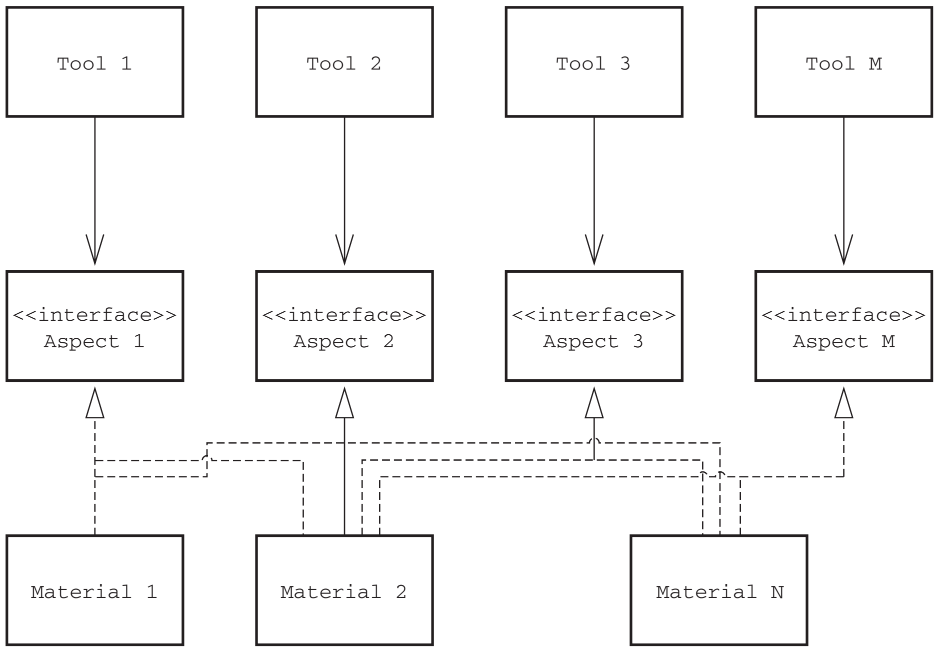


FIGURE 8.8
Using interfaces to implement aspects.

Generalizing aspects

The idea of aspects does not have to be limited to coupling tools and materials. In fact, we have extended it to the relationship between automatons and materials. If we think of aspects as protocols that a class should meet, then we can find additional applications for this modeling type. Examples include a protocol for the use of container classes, or between containers and managed objects, or to distribute objects. In all cases, objects expect a specific protocol, or generally more than one operation. At the same time, we have to answer the question whether separate classes for these protocols would eventually be too expensive and confusing.

Problems with the construction approach

Unfortunately, using abstract superclasses to model aspects for materials has some drawbacks.

When aspects are bound to materials, the *inheritance mechanism is used differently* than in domain modeling. To better understand the relation between domain and software design, we use inheritance primarily to model domain concept hierarchies, where single inheritance is an essential composition rule. When using inheritance to implement aspect classes, we describe the match of tools and materials in one single work context. This is a totally different notion, because an aspect can unite a set of materials that have no similarity from the domain view. They are all totally different, apart from the fact that they can be manipulated by the same tool. For example, a printer could print the set of different materials and a trash could delete them.

Different ways to use inheritance can make your design harder to understand. One solution for solving this problem is name conventions. For example, we name all classes that model domain objects with nouns (e.g., `Folder`, `Form`, `Account`), while using adjectives for aspect classes (e.g., `Editable`, `Storable`).

Interfaces used for aspects

This problem does not initially occur when using interfaces to implement aspects. However, it is meaningful to use interfaces for other things, in addition to aspects, in languages that don't support multiple inheritance for classes, such as Java. For this reason, it is always a good idea to use the name convention proposed previously.

Another question is of a more general nature: Will inherited aspects cause materials to be too closely coupled to tools? After all, inheritence means that a material has to implement static interfaces that may be required only temporarily or in specific use.

Another important thing to remember when working with aspect classes and multiple inheritance in large systems are the costs for compiling and linking, especially in languages like C++. In addition, we may have to deal with an enormous amount of superclasses, resulting in confusing code.

Aspect classes and subsystems

If we develop a complex framework based on the T&M approach and use appropriate mechanisms to structure these frameworks (see Section 8.15), we will have to deal with problems when attempting to allocate aspect classes to subsystems.

Considering that aspect classes describe the interface of a tool to one or more materials, they actually belong to the tool classes that use them. On the other hand, material classes must inherit from aspect classes. This means that material subsystems would depend on tool subsystems, because a material subsystem inherits from classes belonging to the tool subsystem.

Another thing is that we cannot simply allocate an aspect to a material class, because we want the aspect to be basically valid for many materials.

On the other hand, if we build an independent subsystem for aspect classes, then we will have to go through a tiresome amount of work to reconstruct the domain context; and the material subsystems would depend on the aspect subsystems. Despite all these problems, this is the road we will take, for example to avoid relationships between material packages and tool packages in a Java environment. In summary, we explicitly create aspect packages.

RATIONALE

We would use this solution of adapter classes for large and complex systems implemented in languages that provide multiple inheritance.

8.3.2 Construction Part: Using Object Adapters to Implement Aspects

Languages based on single inheritance do not let you use superclasses of materials to build aspects. We propose the use of an adapter, as described by Gamma et al., as an alternative solution.

You can use the *adapter pattern* to let classes collaborate, which they could not do otherwise due to incompatible interfaces. In general, an adapter matches a specific interface (the interface of the object to be adapted) to an interface expected by a client. The adapter pattern comes in two flavors: the *class adapter* is based on multiple inheritance (and corresponds to our aspect class), and the *object adapter*, which is the one we want to use here (see Figure 8.9).

TRADE-OFFS

The obvious benefit of using object adapters is that the aspects can be clearly identified as independent classes in the design. Another important benefit is that this solution also works for languages with single inheritance.

Zull-08.qxd 31/8/04 2:35 PM Page 199

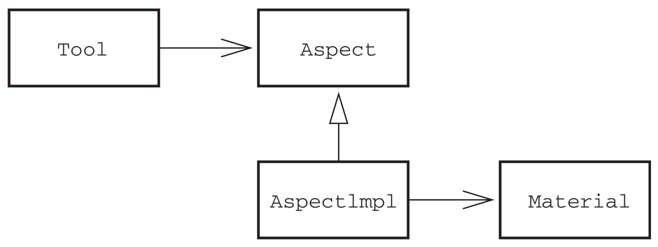


FIGURE 8.9
Using object adapters to implement aspects.

In addition, you can also use object adapters if the interface of a material does not comply with the interface required by a tool. In this case, you can use operations of that material when implementing the aspect so that it emulates the required interface. This method of implementing aspect classes so that the material interface is adapted to a tool can also be useful where aspect inheritance is supported. The operations declared in the aspect class will not be visible in the materials. And the materials remain limited to their pure domain interface, while tool interactions are defined in special adapters.

Adapting materials to different use contexts

Another important benefit of the object adapter transpires when you need materials with different functionality in different use contexts. In such a case, the object adapter will prevent a material object from being loaded with functionality that is required in only one place.

When using adapters, you can statically check for aspect compliance, and the aspects will be visible as independent classes in the design. The material interface has to be suitable basically for an aspect and can be adapted in the aspect adapter class, so that the material remains “slim” (i.e., with a minimum of operations and attributes). Consequently, another benefit of this approach is that you can incrementally improve or expand your project.

Finally, you can link aspects to tools without destroying the overall architecture, as discussed in Section 8.3.1.

Despite all these benefits, the adapter pattern also has drawbacks. One major drawback is that you have to implement a concrete adapter subclass for each material. When implementing generic tools, such as editors or list generators to edit many different materials, you will soon observe an *inflation* of concrete adapter classes. This adds complexity to your design, reducing the benefit of having explicit aspect classes.

More serious problems arise in many contexts resulting from the two objects, such as an adapter object and a material object, that actually make up the material. One of these serious problems is a loss of identity: from a tool’s perspective, a material has a different technical identity than its adapter. However, we need the domain identity of materials in many situations. For this reason, we have to ensure that the domain identity of a material is maintained after we have attached an adapter, despite a different technical identity. A feasible though rather complex solution uses the role pattern described in Section 9.4.1. In any event, we cannot directly compare objects, but have to define an additional identifier for comparison purposes. For example, in C++ you can elegantly solve this problem by overloading the comparison (relational) operator. The additional identifier for objects is required anyway in complex application systems to ensure unique identifiers in connection with persistence and cooperation support.

Finally, we have to solve a construction problem. How and from where should we create aspect objects? If we use direct aspect inheritance to build them, we can simply pass a material to a tool, but the additional adapter objects for each tool-material binding have to be available. The tool should not be able to know and create a special adapter object. It should merely use the abstract aspect class. One possible solution would use a "class object" of the abstract aspect class for the late creation of the concrete adapters. To this aspect class object we would then pass the respective material as a specification for the concrete adapter (see also the pattern *product trader* in Section 9.4.2).

Another more general solution would be the use of the *factory pattern* (see Gamma et al.) for creating adapter objects.

RATIONALE

We recommend that the adapter object solution for languages that use single inheritance without explicit interface construct and for large applications with different workplace types, where material objects migrate between these workplaces. This solution may also be combined with the aspect inheritance solution.

8.3.3 Construction Part: Using Development Tools to Realize Aspects

The previous sections introduce solutions that use aspect objects to build aspects. These solutions check at runtime whether or not there is an adapter object for a material in compliance with the underlying aspect. More specifically, each material offers operations for use by its aspects, and each tool uses only the material operations declared in the aspect. The part of this test that is really critical at runtime has to be done only once, namely when a tool or material is created. Once we can rest assured that a material offers all required aspect operations, or that a tool uses only aspect interfaces, then this situation will never change during that component's life cycle. All we have to do in addition is to ensure that tools and materials are bound correctly.

When using a language like Smalltalk, we can move expensive checks from the time an object is created forward to its class definition. Aspect compliance is then tested during the programming phase rather than at the program's runtime. This solution requires appropriate development tools, which shouldn't be a problem in Smalltalk, because all popular Smalltalk versions support an open development environment. Such an environment lets you modify browsers and program text editors so that aspects appear as independent categories that can be included in parsing.

A development tool for aspect editing purposes, say `AspectBrowser`, can be used to copy the operations of an aspect to material classes. This means that you can both use abstract superclasses to implement aspects and to define standard implementations for operations.

You can also use the `AspectBrowser` to define new aspect classes. Subsequently, you can implement the operations you copied, if they are available only in textual form, or replace a standard implementation from the aspect class by a material-specific one.

Finally, you can use the `AspectBrowser` to check that all operations have been implemented.

Trade-Offs

For this solution to work, the language you use should allow you to expand it during runtime. To take full benefit of the solution, the language should offer a powerful metaobject protocol (see Section 2.7). Notice that, even with the appropriate support by the programming environment, this solution is insecure, because the standard tools of the development environment do not know aspects, which can lead to an inconsistent system.

Rationale

We recommend this solution only for Smalltalk or comparable languages that can be extended at runtime as an alternative to interface objects.

8.3.4 Construction Part: Alternatives to Using Aspects

Sometimes, tools use their materials directly instead of using aspects. In this case, a tool always knows the full interface of a material. Polymorphism is used only if the materials themselves form an inheritance hierarchy.

Trade-Offs

One major benefit of using this alternative to aspects is naturally that working without aspects will simplify the coupling of tools and materials, because no extra work is required conceptually or constructively to let a *single* tool use a *single* material. In addition, the coupling will be typesafe.

One drawback of this approach emerges when we try to let one tool use more than one material. In this case, we have to make some constructive effort and maintain several interfaces. If several tools are to operate on one material, it might become difficult to see which parts of the material's interface are suitable for which tool. Notice that the coupling of tools and materials is not made explicit in the design. Consequently, working without aspects will make it more difficult for us to change or expand our design.

Nevertheless, there is enough justification for small and "young" projects to do without aspects. Designs that have not yet reached their maturity often show a one-to-one relationship between tools and materials. Languages that support dynamic typing, such as Smalltalk, often develop prototypes or initial pilot systems without aspects. In addition, extremely specialized materials may require a close tool coupling, that is the tool must know the full material interface. This applies also to special tools used for a single material.

And finally, aspects represent an abstraction of the work relationships between tools and materials, so that often they can be implemented only once we have a clear picture of these work relationships.

Rationale

We recommend using this solution when starting a new project and then creating aspects successively as required in the course of the project. This solution works equally well for simple tools and materials or very specialized ones.

8.4 THE SEPARATING FUNCTION AND INTERACTION PATTERN

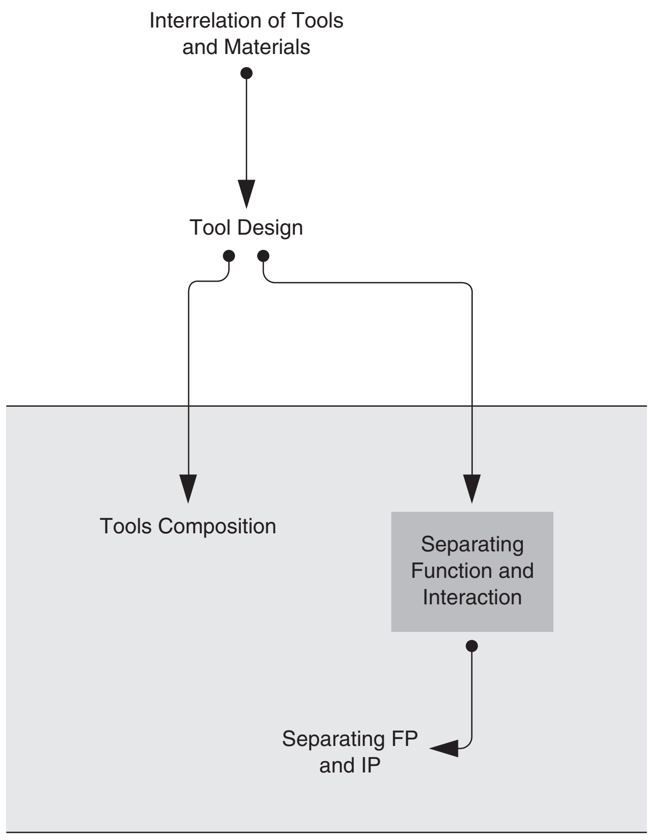

Interrelation of Tools and Materials
Tool Design
Tools Composition
Separating Function and Interaction
Separating FP and IP

FIGURE 8.10
Separating function and interaction.

INTENT

This pattern describes how to utilize an important architectural principle of interactive software systems, that is, the separation of interaction and function as one essential design principle for interactive tools.

PROBLEM

We want to maintain the separation of concerns between a tool's handling and presentation on the one hand and on the other hand its functionality in software. We want to be able to modify the interactive parts of the tool without having also to adapt its functionality.

RELATE TO

The conceptual patterns *tool design* and *interrelation between tools and materials* show how to design tools and materials on a conceptual level (see Figure 8.10). When we have understood what a tool is good for and how users should be able to interact with it, we can deal with the problems addressed in the current pattern.

Solution

When implementing tools, we observe the following conceptual division:

Design guidelines for tools

- **Define the characteristic domain functionality for each tool independently of its shape. We call this its *function*.**
- **Define a specific shape and a characteristic handling and presentation for each tool based on its function. We call this its *interaction*.**

Background: Tool Construction

We said that we define software tools to be interactive. For this purpose, we assign a domain functionality as well as handling and presentation to each tool. To elaborate on these guidelines, we search for answers to the following questions:

- *Function*: What domain purpose does a software tool have?
- *Handling*: How can a software tool be used?
- *Presentation*: How can a software tool and the material it works on be represented?

Obviously, a tool must be good for *something*. It lets us handle tasks in that we can use it to work on suitable materials. This means that a tool has a domain functionality. The conceptual pattern *relating tools and materials* indicate how to design the functionality of a tool.

A tool never becomes active on its own; it is always handled by a user, so that handling is essential for a tool. The domain functionality of a tool is accessible only by handling it. The conceptual pattern *relating tools and materials* shows that the domain functionality can be utilized by different usage forms or interactions. This means that, although handling refers to functionality, it can also be implemented independently of the functionality.

For a tool to be usable, it has to have a representation of its own. Considering that a tool shows the state of the material it manipulates, the tool has to represent the material. This representation is the only way that a tool can give feedback for the user needed for a reliable working with tools and materials. Handling and presentation are closely related.

Trade-Offs

There are several ways to separate the function and interaction of tools. First, we can implement both concepts in a single construction unit by representing these two concepts as distinct interfaces (see design pattern *tool composition* in Section 8.5).

Responsibilities of an interactive tool

An alternative would be to further divide tools along the basic concepts. This will initially produce two construction units; we call them *functional part* (FP) and *interactive part* (IP). We have described the appropriate design pattern *separating FP and IP* in Section 8.7.

When further dividing a tool, we take the three fundamental *responsibilities* of an interactive tool into account. As mentioned earlier, each tool has a GUI representation that can be manipulated by users, in addition to its domain functionality. The original model-view-controller (MVC) paradigm found in Smalltalk systems uses separate classes for each of the three responsibilities. Considering that presentation and user input are closely coupled in modern GUIs, the *separating FP and IP* pattern combines the presentation and manipulation of user inputs, originally separated as *view* and *controller*, in a class—the interactive part.

We can identify a number of arguments in favor of each of the solutions introduced here for internal tool construction. For prototypes, simple tools, and small or young software projects, a separation based on the MVC paradigm or FP-IP model can introduce too much overhead. In these cases, we suggest the use of monolithic tools instead (see Section 8.16.2).

However, the latter solution should be handled with care, because there is a tendency to miss the right point when a tool with its different responsibilities should be divided into separate classes. If this job is done late, it could become expensive.

For complex tools or large software systems, tools that were divided from the outset are much easier to maintain and reuse, compared to monolithic tools.

RATIONALE

Always conceptually separate a tool's interaction and function. For complex tools there should be distinct construction units; also, if the interaction is likely to change frequently. Otherwise, you could just represent interaction and function as different interfaces.

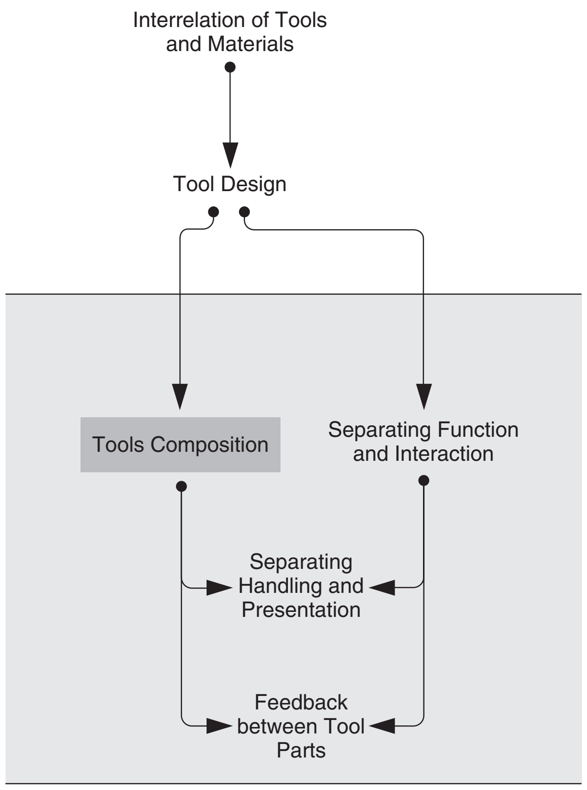
WHAT NEXT

The construction part using *components to build tools* of the design pattern *tools composition* shows how to represent interaction and function as two interfaces.

The design pattern *separating FP and IP* shows how to implement interaction and function as two construction units.

8.5 THE TOOLS COMPOSITION PATTERN

FIGURE 8.11


Interrelation of Tools and Materials
Tool Design
Tools Composition
Separating Function and Interaction
Separating Handling and Presentation
Feedback between Tool Parts

The Tools Composition pattern.

INTENT

This pattern shows how you can build complex tools by composing subtools that are responsible for handling well-defined tasks.

PROBLEM

A tool, including its manipulation and presentation and its domain functionality, can be very complex, depending on the work and tasks. Using a few construction units to map this complexity to one single tool is normally not a feasible software solution. In addition, each tool has to be developed from scratch, even when we implement similar subsets of functions. For this reason, we should try to build tools on the basis of existing tool components to reduce the complexity of each component and make them reusable. Therefore

How can we divide a tool into subtools, and how should these subtools be integrated into one tool?

RELATE TO

The conceptual patterns *tool design* and *interrelation between tools and materials* show how to design tools and materials on a conceptual level (see Figure 8.11). Once we have understood how a tool is related to the individual tasks in the application domain, we can address the problem of how complex tasks or entire workflows should be handled by a composition of tools.

SOLUTION

We develop tools so that they are responsible for a defined task or set of activities. We integrate the subtasks of these simple tools to form a combination tool. We build a context tool that integrates several subtools. Each tool can basically be embedded in the context of an enveloping tool, thus becoming a subtool. For this purpose, we design a generic component interface.

The schema in Figure 8.12 shows the pattern and technical relations.

In our tools composition, we will ensure that context tools know their subtools and can call their operations directly.

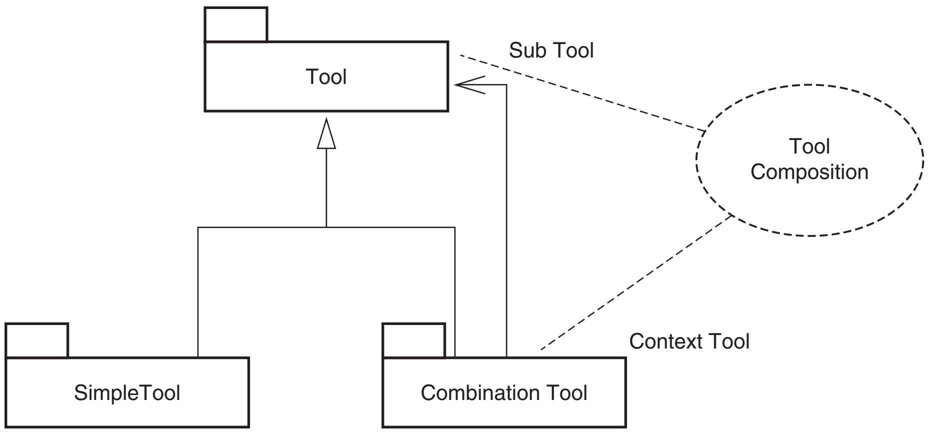


FIGURE 8.12
Building tools by composition.

Background: Building Subtools

We design tools for the domain tasks on hand, that is, we normally design one tool for each task. We want to be able to combine tools for simple tasks to complete more complex interrelated tasks. Therefore, new tools should be built on the basis of existing ones. This allows us to reduce the complexity of single construction units and reuse existing components.

Tools designed to support complex activities are normally built so that they can manipulate one material and even material containers. The work that a tool is designed for can normally be divided into subtasks, that is, working with single materials and containers.

A taxonomy of tools

To combine tools from components, we first need to define several terms.

A tool used as a technical construction unit within another tool is referred to as a *subtool*.

A *context tool* embeds subtools, both from the technical and conceptual views, that is, it implements a domain combination tool.

A *combination tool* combines different domain services to complete a complex task. It is composed of subtools in software, and it is a domain element of the usage model.

A *simple tool* is domain-specific and represents one elementary task or functionality. On the software side, it has no subtools. Simple tools are also domain elements of the usage model.

The term functionality of simple tools has to be strictly separated from the term function. The reason is that simple tools do not implement functions the way we know from the structured design (e.g., create a record). Simple tools normally also have states.

Since we are talking about simple and combination tools in our domain design, we are now interested in how subtools and context tools relate. This composition can be recursive. For this reason, a tool can occur as a context tool for its subtools and be itself a subtool. In addition, we look for a view of tools as components to facilitate the composition.

Example

The EMS example

Figure 8.13 shows an example for a complex tool—the `DeviceOrganizer` we know from previous sections. This example shows each room by a subtool. The actual room plan embeds these tools. The right-hand part of this figure shows a special room—the storage room, which is also represented by a subtool.

We could build this tool in different ways:

- We design separate tools for each task (a `DeviceHandler`), allocating employees to rooms (a `RoomHandler`) and managing the storage room (a `StorageHandler`).
- We build a complex tool that can display rooms and the storage room and run operations on these rooms and devices.
- We build one complex tool by combining the single components.

Zull-08.qxd 31/8/04 2:35 PM Page 207

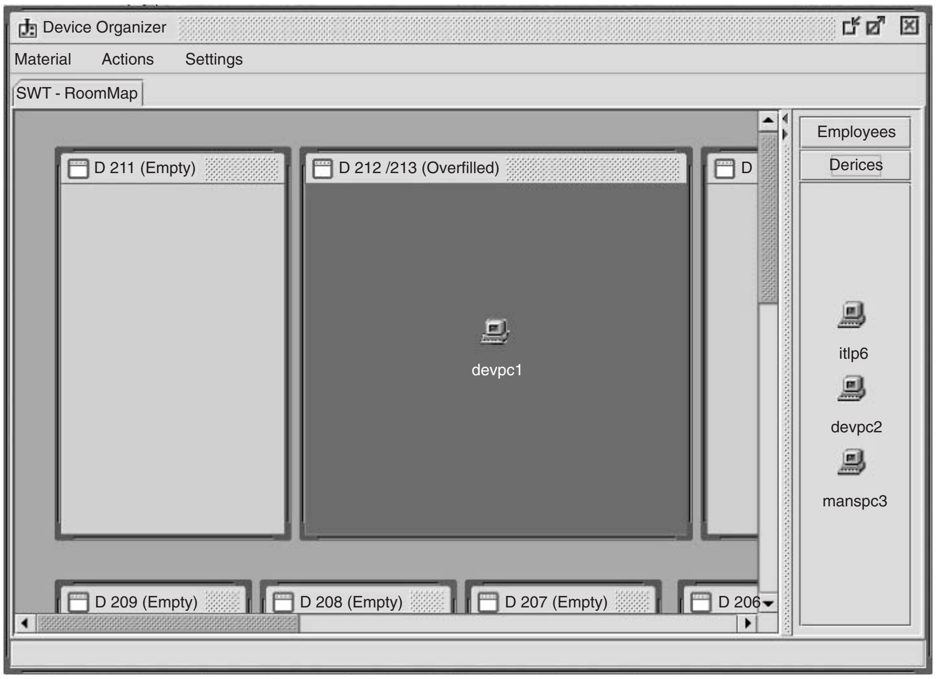


FIGURE 8.13 Example of a complex tool: the `DeviceOrganizer`.

In our `DeviceOrganizer` example, we have chosen the following solution. The `DeviceOrganizer` is a combination tool implemented by one context tool and several subtools, where these subtools know nothing about each other. They are controlled by the context tool so that the entire task of the combination tool, such as organizing devices in rooms, is fulfilled.

TRADE-OFFS

Design alternatives for related tools

When building separate tools, we often find it difficult to establish a relationship between these tools. For example, when we move a device from storage to a room, then this involves both the room and the storage. To produce this relationship, we would either have to link the two tools or use their material to link them.

Linking the two tools directly means that one of the two tools (e.g., the storage handler) has to know the other tool. This is not a good solution, because the storage handler would lose its independence; it could be used only in combination with the `DeviceOrganizer`.

Using the material to link the tools is also ruled out by our construction principles, because the material would then know about the existence of the tools, or at least have to have some notification mechanism.

Though we could build one single complex tool, this solution is not appropriate from the software view, because we would not be able to use the single parts, that is the `DeviceOrganizer`, room handler, and storage handler, separately. For example, we would have to write a new room handler for the room planner to create and arrange rooms.

For these reasons, it appears meaningful to encapsulate all subtasks in independent tool components. We could then compose tools based on the building block principle. This means that existing tools should

- be usable as *components* in new tools, that is, we can reuse their functionality for subtools in extended contexts; and
- whenever sensible, be able to act as *combination tools*, using the functionality of new components.

We are looking for a construction part where simple tools can be composed into combination tools based on a uniform schema.

Rationale

Whenever you have to design and implement a tool you should think about reusing existing ones. Thus, for every tool design and construction, this pattern should be checked for applicability.

What Next

The design pattern *separating handling and presentation* shows how to further subdivide tools with an elaborated user interface.

The design pattern *feedback between tool parts* shows the basic principles and constructions of how to couple tool components in tool hierarchies.

8.5.1 Construction Part: Using Components to Build Tools

To enable an integration of tools as components, all tools have to support a common tool interface. Figure 8.14 shows a minimal tool interface.

In this example, the tool interface has to let us initialize the tool (`equip`). Subsequently, the tool can be activated (`activate`) and deactivated (`deactivate`) or closed (`close`). The tool's `Interaction` interface allows us to show or hide the tool representation. The `Functionality` interface supports all domain-specific operations on the tool and in addition, those operations required for tools and materials to interact—mainly setting and probing the material. Specific tools specialize this interface.

Trade-Offs

Building tools like components

We build tools so that they behave like components. This means in particular that their internal representation is hidden. This approach allows us to combine tools with different internal construction to one single system. It even allows us to combine complex tools from different other tools, which do not have to be built by the same internal schema.

On the other hand, this construction approach leads to a generic interface that may not fit perfectly in all situations. If, for example, we want to work with multiple materials, then we have to use `setMaterial()` for changing the materials. This is not very elegant and leaves the tool in an undefined state during the changes.

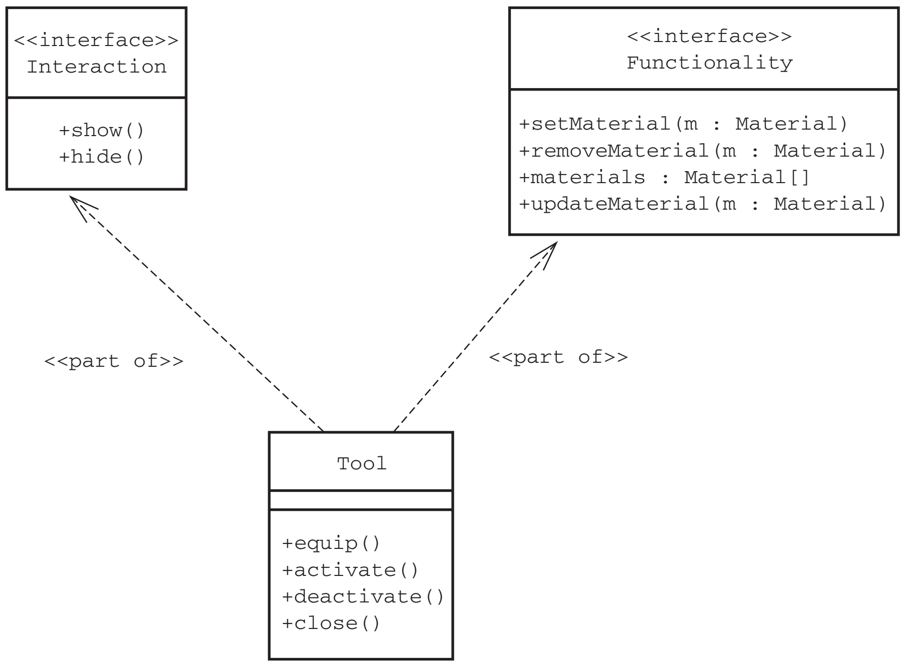


FIGURE 8.14

Minimal tool interface.

RATIONALE

Use this pattern whenever you have to combine subtools with different internal structures or reuse existing tools that are built without a clear separation of function and interaction.

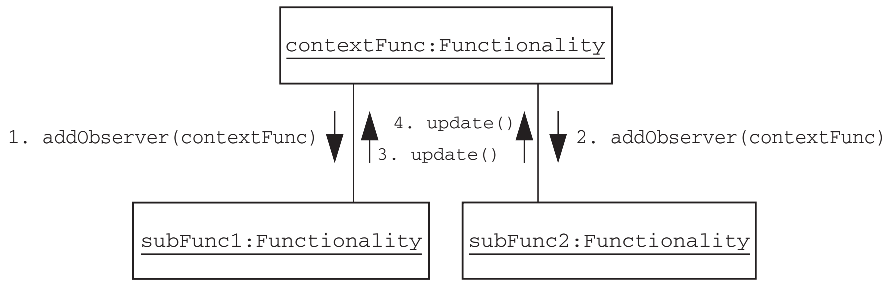
8.5.2 Construction Part: Using Components to Build Combination Tools

When building subtools and context tools that, together, form a combination tool, we have to answer the question of how the functionality of each of the above simple tools should behave within that combination tool.

Constructing combination tools

To implement a combination tool, we use simple tools that provide the functionality we need. Accordingly, the functionality of each simple tool is used as a subfunctionality. To combine several subfunctionalities, we build an additional context tool. This context tool implements the interaction of the subfunctionalities and delegates subtasks to the responsible subfunctionality. Remember that we are here talking of the conceptual functionality of a tool, which does not necessarily mean that there has to be an independent class for this functionality. The responsibility defined by a functionality can also be assumed by the tool class. In this case, the tool class would directly provide this functionality (see the discussion of monolithic tools in Section 8.16.2).

Delegating subtasks to subfunctionalities means that we clearly distribute tasks over components. This means that the context tool does not have to handle all subtasks itself. In one direction, from the context functionality to the subfunctionalities,



contextFunc:Functionality
1. addObserver(contextFunc)
4. update()
3. update()
2. addObserver(contextFunc)
subFunc1:Functionality
subFunc2:Functionality

FIGURE 8.15 Feedback between a context functionally and its subfunctionalities.

there should be tight coupling, because the context functionality delegates specific tasks to the subfunctionalities, so it must know their interfaces.

Designing a context tool

For the context tool to be able to assume its coordinating function, it needs information about relevant changes in the subfunctionalities. On the other hand, we don't want subtools to know their context tools, which would cause cyclic dependencies and eventually make the system more difficult to understand and maintain. For this reason, we use loose coupling in this direction.

In general, subfunctionalities should be built to be independent of their context functionality, so that we can use them in different contexts. Since as we want to build reusable tool components, it will eventually not be clear when building a tool that it will be used as a subtool and integrated in a specific context tool.

The feedback problem

We develop single tools so that they can also be used as subtools. For this purpose, the tool functionality has to support a defined subtask. We use a context functionality to integrate subfunctionalities into a combination tool. The context functionality will then delegate subtasks to its subfunctionalities and coordinate them. In doing this, we use the event pattern or observer mechanism to solve feedback problems.

Figure 8.15 shows how the observer mechanism is used. In this example, the context functionality is the observer and the subfunctionalities are the observed.

EXAMPLE

The EMS example

In our EMS example with the `DeviceOrganizer`, the context functionality `DeviceOrganizerFunctionality` coordinates a subfunctionality, `RoomEditorFunctionality`.

When a user drags and drops a device from one room onto another room, then the `DragDropManager` removes the device from the source room and adds it to the target room, as shown in Figure 8.16. This causes the source room editor and the target room editor to send an event (using the `announce` operation) saying that their device sets have changed. The `DeviceOrganizer` has registered for this event and obtains the new device sets from the source and target rooms. The `DeviceOrganizer` can now have the screen representation updated.

RATIONALE

This pattern should be used to combine well-designed subtools into combination tools.

8.5.3 Construction Part: Identifying Tool Boundaries

We have seen how tools can be recursively combined to form complex tools. However, when looking at a tool component, this construction does not tell us automatically

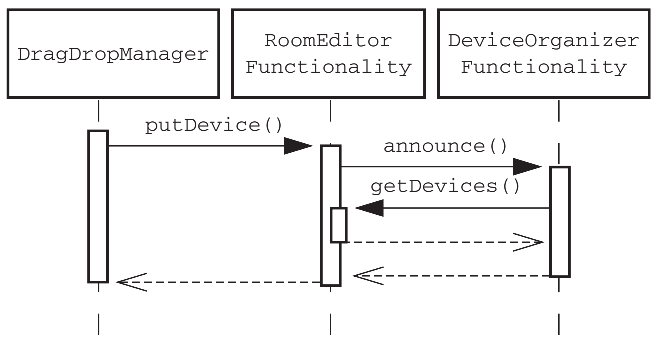


FIGURE 8.16
Control flow when adding a device.

whether or not it is the top-level context tool or an embedded subtool. But we can always decide at runtime whether or not a tool component marks the tool boundary. This means that this is the only tool component in this combination tool that has no higher-level context tool.

A tool has to fulfill certain software and domain responsibilities, which we want to manage in a dedicated instance. A good example for such responsibilities would be information about tool names, presentation icons, vendor names, and tool versions, or a list of aspects that the tool can manipulate. This information could be accommodated in the context functionality of a tool. On the other hand, we could argue that this information rather refers to the tool as a whole.

EXAMPLE

The EMS example

Figure 8.17 shows a class diagram for our combination tool, the `DeviceOrganizer`. The `DeviceOrganizer` has a functionality, `DeviceOrganizerFunctionality`, which formulates the domain handling of that tool. The `DeviceOrganizer` can have an arbitrary number of subtools of type `RoomEditor`. In turn, each `RoomEditor` has a functionality, `RoomEditorFunctionality`. The `DeviceOrganizerFunctionality` knows the `RoomEditorFunctionality` objects, while the context tool, `DeviceOrganizer`, does not know the other parts of the `RoomEditor's` tool interface.

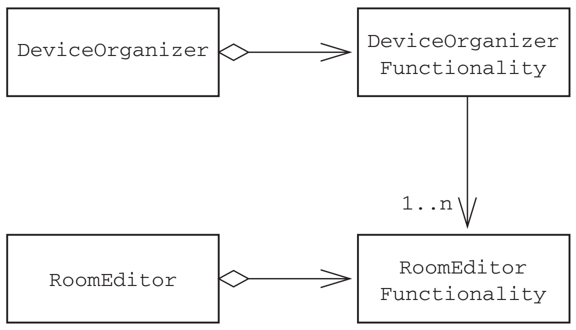


FIGURE 8.17
Functionalities in the Device Organizer.

RATIONALE

Use this pattern whenever tools have to be represented by a single instance in your system and when a tool as a whole has to offer specific services that cannot be allocated to a tool part.

8.6 THE FEEDBACK BETWEEN TOOL PARTS PATTERN

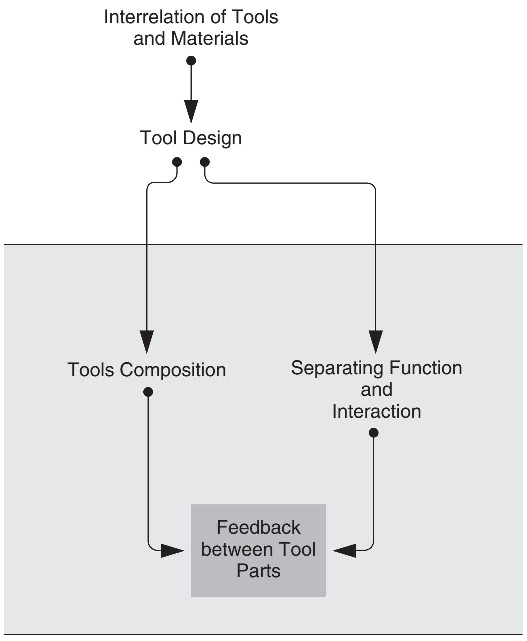

Interrelation of Tools and Materials
Tool Design
Tools Composition
Separating Function and Interaction
Feedback between Tool Parts

FIGURE 8.18 The Feedback between Tool Parts pattern.

INTENT

This pattern tells you how to realize a feedback mechanism between tool parts, so that one part knows as little as possible about the other.

PROBLEM

When composing tools from single components, we often find that there is an asymmetric relationship between these components. A component, such as the context tool, knows its subtools from their interfaces. On the other hand, a subtool should know as little as possible about its context tool when state change messages are exchanged.

We can formulate the general problem that this pattern solves as follows:

Two components should be linked together so that one component is the service provider and the other is the client. The client is responsible for reacting to the

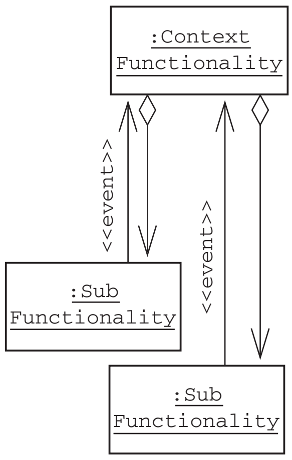


FIGURE 8.19

Feedback problem of a combination tool.

results of service requests. The client then needs to know how the provider's state has changed once its services have been used.

Relate To

This pattern solves a problem that emerges when you use the patterns *separation of function and interaction* or *tool composition* (see Figure 8.18).

Background: Feedback Mechanisms

Let us look at a subfunctionality that should have a way to inform its context functionality about state changes. As mentioned earlier, we want to use loose coupling in this direction to avoid cyclic dependencies. This inversion of the control flow is called *feedback problem* (see Figure 8.19).

The literature describes several construction approaches and patterns to implement the required feedback mechanism. All of these construction approaches are either based on their use contexts or are bound to specific language mechanisms.

Feedback mechanisms can be distinguished by whether the observer is generally informed about the changes of the subject, or whether the subject informs its observer about state changes in specific ways. In addition, a feedback mechanism may be able to address a specific observer or an unknown number of observers arranged in a hierarchy.

Event patterns initially appear suitable for our purpose of using and combining tool components. A similar solution would be the *observer pattern* proposed by Gamma et al., but if we have to deal with increasingly complex dependencies, it is better to use a variant of the event pattern, which includes explicit event objects. In contrast, we use a *chain of responsibilities* (Gamma et al.) for hierarchies of potential handlers. A component can then send a request to these unknown handlers.

Solution

We build a feedback mechanism, which informs observers about changes to a subject in an abstract way so that these observers can respond to such a change.

The following construction parts spell out different concepts and related construction approaches to solve this problem on a more concrete level.

8.6.1 Construction Part: Event Pattern

This construction part uses combination tools, composed of tool components. The *event* pattern will serve us as feedback mechanism (see Figure 8.20).

A subfunctionality informs its context functionality anonymously about each relevant state change. In this example, the context functionality assumes the role of an observer, while the subfunctionality acts as subject. If a tool is internally built from separate functional and interactive parts, then the event pattern can also be used by functional parts to inform interactive parts (see *separating FP and IP* in Section 8.7). In this case, the interactive part is the observer and the functional part is the subject.

An observer knows its subject and calls operations that change or probe it. The feedback mechanism works in the opposite direction, that is, the subject informs the observers about events when a relevant state change has occurred. In order to avoid an uncontrolled "firing" of events, we divide the operations at the interface of a class into *statements*, *requests*, and *tests* (see Section 2.1.8). We apply this rule to the interaction between context functionality and subfunctionalities:

Structuring the interface and interaction between functionality and sub-functionalities

- *Requests* and *tests* are probing operations (functions) with no side-effects. Requests inform the calling context functionality, for example, about the state of a material or about the configuration of a subtool. Tests are often used to show whether or not other operations of a subfunctionality may be called, such as to test pre-conditions for other operations. Probing operations must not cause a visible side-effect at the interface of a subfunctionality. In particular, they must not lead to informing the context functionality in its role as observer. For this reason, requests and tests can be called at any time as a response to a notification of the subfunctionality.
- *Statements* are operations (procedures) that change a state. They are used by the context functionality to have a subfunctionality manipulate a material or change the work state of a subtool. After a statement, the context functionality can call a probing function to check whether or not the last subfunctionality call was executed successfully. Most statements lead to a notification. For this reason, a statement should never be used by a subfunctionality as a response to an event it received.

Trade-Offs

Interaction between observer and subject

The example of the FP-IP coupling by means of the *observer* pattern shows a clear benefit of this anonymous notification: It is very easy to have arbitrary observers observe a subject at any later point in time.

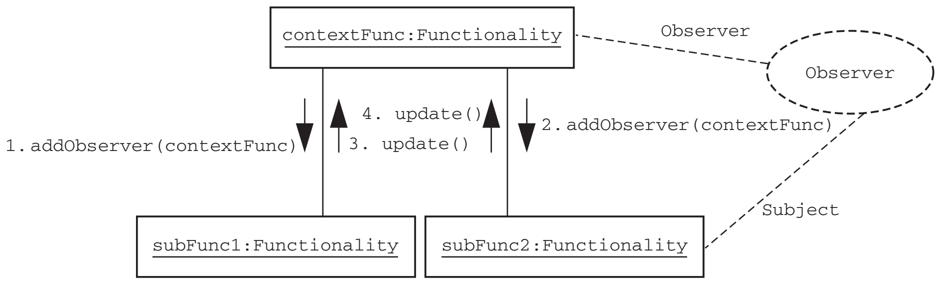


FIGURE 8.20 Using the Event pattern for combination tools.

The subject (here the functionality) must not know how its observers respond to changes. All the subject should know is that the observers can potentially respond to changes. The idea is to have the subject inform the observers with minimum knowledge of the interface when their own states or the material state has changed. Considering that we don't deal with languages that have an anonymous signaling mechanism built in (e.g., Events in HyperTalk), we have to use the regular call mechanism for this notification.

Avoiding the "oscillation" problem

We have to take action to prevent an undesired "oscillation" between a subject and observers. The reason is that, when observers cause another state change to the subject as a response to being notified by the subject, then this leads to a notification. Obviously, this process could continue infinitely and never end. Another problem can occur when several observers register for the same event. In this case, observers that get notified later will not find the subject in the state that was signaled to them. They would then take wrong assumptions, such as calling operations on the subject that can be called in the signaled subject state, but no longer in the current subject state. We call these problems collectively "reactive change" and request that reactive changes be constructively avoided.

If, however, these rules are observed, then the event pattern can be used safely. But note that there are cases where we will combine a procedure with a function, such as for reasons of better performance or understandability. Then, the operation looks like a function but semantically is a procedure with a return value. In order to avoid confusion and reactive changes, we should carefully document this type of procedure and, perhaps, use naming conventions.

RATIONALE

This is the standard mechanism for a feedback mechanism with loose coupling of clients and related service providers. It works well for all simple tools, that is, tools with few different events.

8.6.2 Construction Part: Event Objects

We build an event class and create an independent event object for each relevant state change in the subfunctionality. We extend the subfunctionality's interface so that each event can be polled at the interface. The context functionality can register for events available at the subfunctionality's interface. It can optionally pass an operation together with the event.

The context functionality normally registers directly with each event. In many languages (e.g., C++ and Java), an operation called by an event can be passed together with an event more or less elegantly, that is, typesafe.

The EMS example

EXAMPLE

This section describes how we can implement the event pattern, extended to include event objects, for the `DeviceOrganizerFunctionality` and `RoomEditorFunctionality` in our EMS example (see Figure 8.21).

The class `RoomEditorFunctionality` implements the interface `EventSubject`. `DeviceOrganizerFunctionality` implements two interfaces, that is, `EventSubject` and `EventObserver`, where the latter indicates that `DeviceOrganizerFunctionality` is also an observer. None of the two classes has to implement operations. Note that `EventSubject` and `EventObserver` serve merely for typing and hiding the mechanism used to signal events.

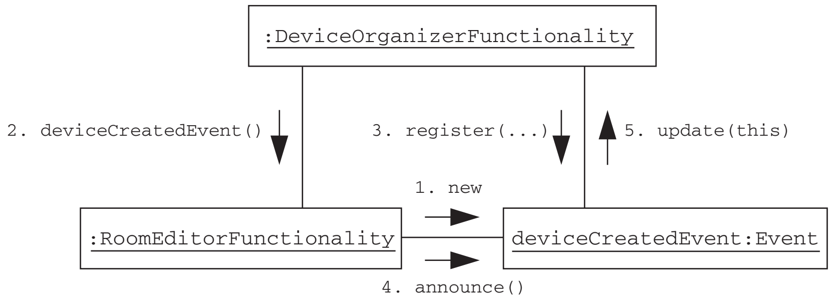


FIGURE 8.21 Even pattern with event objects.

At its interface, the `RoomEditorFunctionality` offers one operation for each event that it announces; in our example we only have the operation `deviceCreated()`. This operation returns a corresponding event object, which is created during the initialization of the `RoomEditorFunctionality` (step 1). The `DeviceOrganizerFunctionality` registers directly with this special event, `deviceCreatedEvent`.

The `DeviceOrganizerFunctionality` requests an event object for a specific change that it is interested in from `RoomEditorFunctionality` (step 2). The `DeviceOrganizerFunctionality` uses `register()` to register one of its methods with this event object, `deviceCreatedEvent` (step 3). As this is a Java example, the `DeviceOrganizerFunctionality` registers with the event using an anonymous inner class.

If `RoomEditorFunctionality` calls the operation `putDeviceProxy`, then the appropriate state change (e.g., adding a device to a room) will be executed. Next, `RoomEditorFunctionality` calls the `announce()` operation at the event object, `deviceCreatedEvent` (step 4). This means that, rather than informing all observers of `RoomEditorFunctionality` about an effected change, only the observers registered for this specific event object are informed (step 5).

Once the control flow has reached the `DeviceOrganizerFunctionality` after calling `reactOnDeviceCreated`, `DeviceOrganizerFunctionality` can then respond to this event. It uses the operation that has been passed as a call back. This means that it knows the state change effected in `RoomEditorFunctionality`, so that it can poll the new values from `devices` and update its representation.

TRADE-OFFS

Complex tools and the observer mechanism

Complex tools show particularly well that the simple signaling of an observer can cause noticeable runtime effects. A considerable number of possible state changes need to be requested from a subfunctionality, or detected in the context functionality. This is the reason why different state changes to the subfunctionality should lead to different events, which can be distinguished early on, that is, in the subfunctionality. The context functionality can then register for specific events.

The different states of a subfunctionality need to be observable at its interface. We could develop a dedicated observer that monitors changes specified at a subfunctionality's interface.

Zull-08.qxd 31/8/04 2:35 PM Page 217

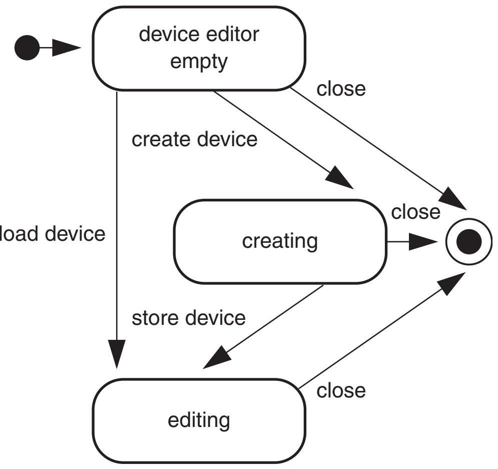


FIGURE 8.22

State model for the Device Editor in *the* EMS.

In practice, we often find that subfunctionalities and their events are strongly tailored to existing observers. This means that subfunctionalities offer only events needed by existing observers. On the other hand, it also means that a subfunctionality is more strongly coupled to observers, so that it may be difficult to replace observers later on. To avoid this implicit dependence, we can model the states and events of a subfunctionality solely from the domain view. In this respect, the design of state automatons for subfunctionalities has proven to be a useful technique (see Figure 8.22). For this purpose, we define the domain states that a subfunctionality can take, as well as the changing operations that initiate state transitions. Each state transition will then be an event. For example, we can deduce the events called `deviceCreated`, `deviceLoaded`, and `deviceStored` from the state diagram shown in Figure 8.22. This is a nice way to design subfunctionalities that are independent of specific observers.

RATIONALE

This is a rather elaborate feedback mechanism for complex tools with several events and a broad probing interface of the observed subject.

8.6.3 Construction Part: Chain of Responsibility

When using the event pattern between subfunctionalities and their context functionality, we often find that there are messages sent from a subfunctionality to its context functionality, which are not caused by a state change of that subfunctionality. These are messages by which the subfunctionality signals that it cannot supply a requested service. In addition, it is often difficult to identify the entity that can actually supply a requested service. To allow sending such messages, we can build a *chain of responsibility* between a subfunctionality and its context functionality; this chain can then be used to send such messages.

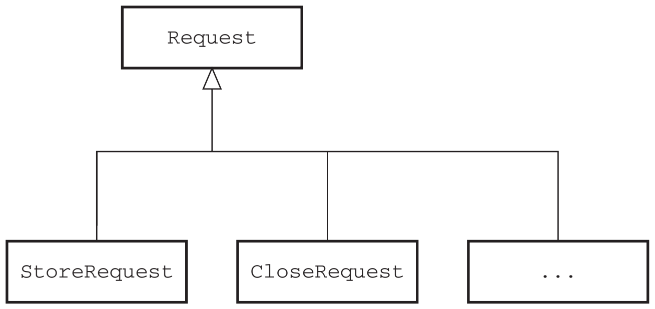


FIGURE 8.23
Often used *Requests*.

A chain of responsibility adds functionalities to a tool tree, in addition to the event pattern. Requests can then be sent along the chain of responsibility. We represent requests in a separate class, say `Request`, and we can then derive more specialized request classes. For example, we could introduce a special request for a closing procedure (see Figure 8.23).

BACKGROUND: CHAIN OF RESPONSIBILITY VERSUS EVENT PATTERN

For example, if a user wants to close a tool, he or she will most likely click a button representing that subtool on the screen. However, the subtool cannot terminate itself or its context tool. Also, it does not know whether or not the context may reject such a closing attempt. Therefore, the subtool has to delegate this task to its context. Depending on the construction, the context can consist of a context functionality, a tool object, or the work environment.

If we select the event pattern to let the subtool's context send a close request, then we will violate several concepts of this pattern:

- An event is sent, although the functionality's state has not changed.
- The event pattern should be used exclusively for signaling, but not to request services from the context, in the sense of an operation call.
- If a subtool wants to close, then the observing object will not simply delete the subtool from the memory. It will normally delete the subtool *and* call several cleanup operations. For example, in C++ we would automatically call the destructor. So we would have a reactive change leading to a system behavior that is hard to control.

EXAMPLE

The EMS example

Let's see the above idea in our EMS example. The `DeviceEditor` can edit several devices concurrently. If a user closes a card tab, then the corresponding subtool sends a request announcing that it wants to be closed (see Figure 8.24, step 1). The context tool responds to this request by closing the subtool (step 2). If the user closes the context tool itself, then the context tool will send a request to the environment (step 3), causing the work environment to close the entire tool (step 4).

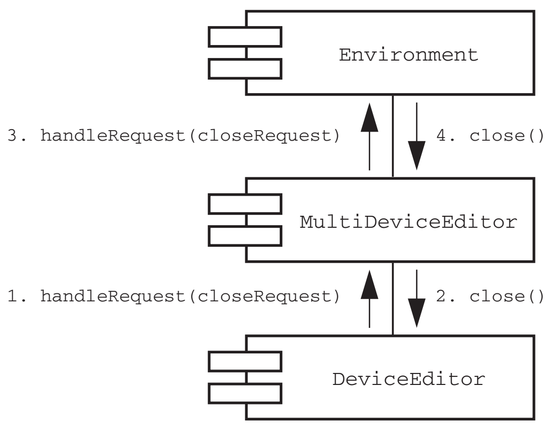

Environment
3. handleRequest(closeRequest)
4. close()
MultiDeviceEditor
1. handleRequest(closeRequest)
2. close()
DeviceEditor

FIGURE 8.24
Chain of responsibility for our EMS example.

Trade-Offs

Regardless of whether or not you use a chain of responsibility, deleting objects can cause problems in some programming languages, in particular when there are still operation calls on objects on the callstack that are to be deleted.

Using C++

In languages like C++, where you have to delete objects explicitly, problems often arise while the callstack is being processed. To avoid having to remove the callstack of objects that have already been deleted, we could integrate a trash bin object into the system. When you then delete objects, they will be moved to that trash bin, while the callstack can be further processed, and observers can be deregistered. Objects in the trash bin will be actually deleted from the system when you empty the trash bin. We place the trash bin emptying task before a new request to the event loop of the window system, because we can assume that, at this point of the control flow, there will be no more active objects.

Using Java

In a language like Java, which has its own garbage collector, we don't need a dedicated trash bin to delete tool hierarchies. Java's garbage collector only deletes objects that can no longer be reached by a thread running in the virtual machine. This prevents the deletion of objects that are still referenced in the callstack.

Rationale

Use a chain of responsibility for passing requests to the context. Whenever a subfunctionality or a subtool or another subordinate element cannot handle a request for service on its own, this is the right construction approach.

8.6.4 Construction Part: Tool Component with Reaction Mechanisms

The component model for tools allows us to compose combination tools from simple tools. However, for a fully generic tool interface, we have to integrate reaction mechanisms. Figure 8.25 shows these interfaces, which support both the event pattern and the chain of responsibility.

Rationale

Use this construction part as a reaction mechanism when you assemble combination tools from simple tools.

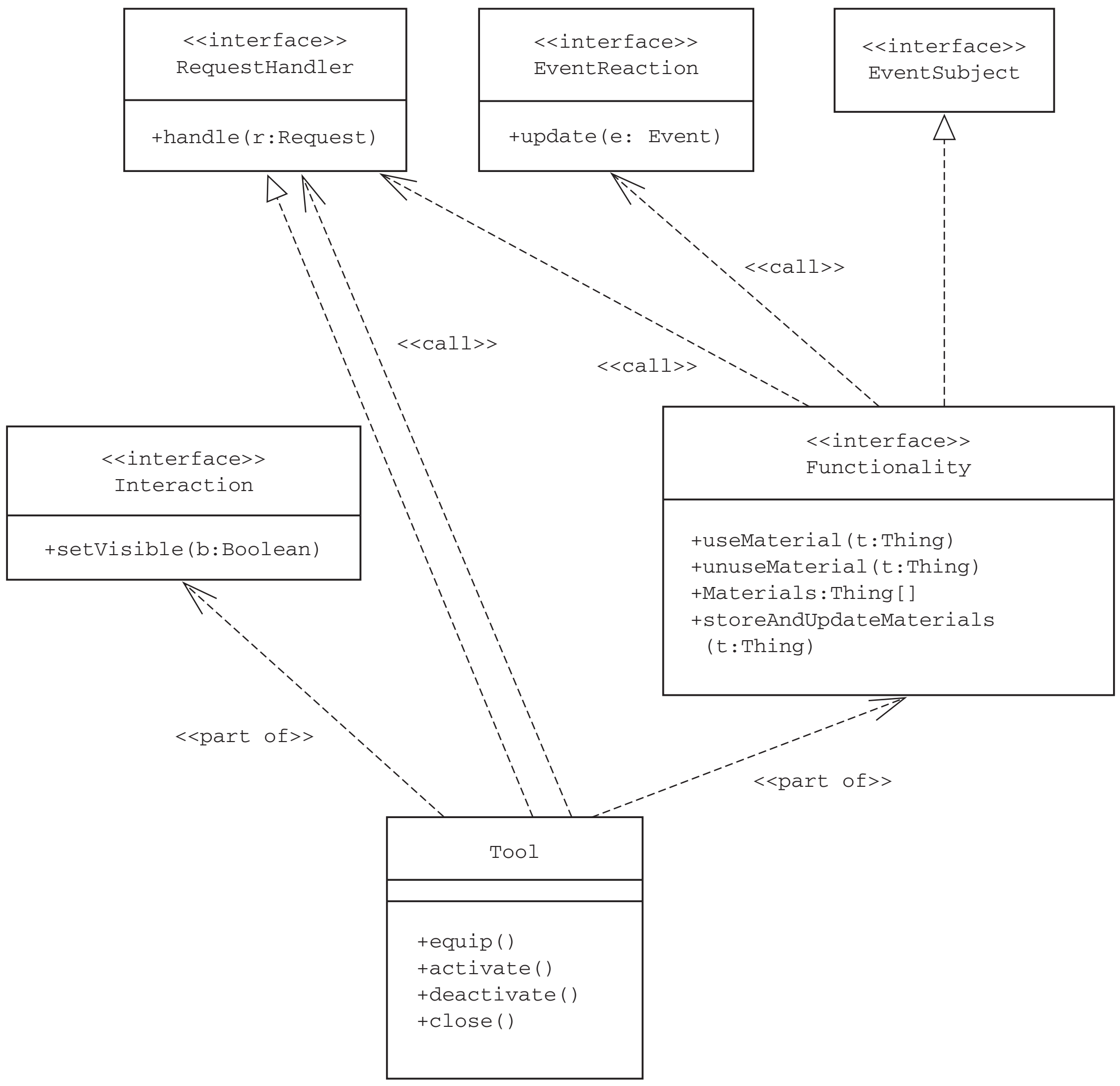


FIGURE 8.25 A tool component with reaction mechanisms.

8.7 THE SEPARATING FP AND IP PATTERN

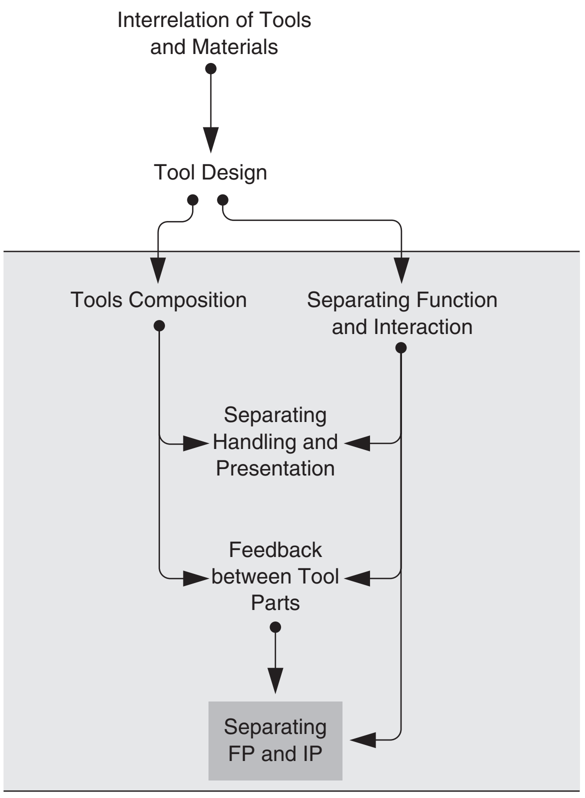

Interrelation of Tools and Materials
Tool Design
Tools Composition
Separating Function and Interaction
Separating Handling and Presentation
Feedback between Tool Parts
Separating FP and IP

FIGURE 8.26
Separating FP and IP pattern.

Intent

This pattern details the essential concepts for designing an interactive tool. Even if you decide to implement the function and interaction of a tool as one construction unit, you should understand the principle behind this pattern.

Problem

There are a number of different ways to constructively implement the conceptual division of function and interaction.

The pattern *separating function and interaction* that we previously introduced described the basic tool design principles. When developing a tool, we will not see the usage quality of its handling and presentation before it is actually used. Since requests for changing the handling and presentation of a tool will arise from its use, we want to implement the requirements *for different handling and presentation in software* without having to adapt the domain functionality.

The presentation of a tool depends largely on the GUI elements, so we also have to take the changes in the GUI into account. We want to change the interaction of a tool by exchanging window systems or GUI frameworks, without affecting the function. Therefore,

We need an internal division of interaction and function into two construction units, which allows us to exchange the tool interaction without impact on the tool functionality.

Relate To

This pattern directly follows the pattern *separation of function and interaction* or *tool composition*. It leads to the feedback problem already addressed by the pattern *feedback between tool parts* (see Figure 8.26).

Solution

We divide a tool into a *functional part* (*FP*) and an *interactive part* (*IP*). We combine FP and IP so that an IP knows and uses its FP, while the FP knows as little as possible of its IP. For this purpose, we use a suitable feedback (or reaction) mechanism.

The FP implements the functionality of a tool, while the IP implements the tool's interaction. We loosely couple the two parts, so that the primary direction of the control flow, from the user into the system, is observed, thus maintaining the FP's independence of IP.

Task division between FP and IP

The following division of tasks for FP and IP results from this general separation of function and interaction:

- The *functional part* (FP) is the acting and probing part of a tool. It defines the domain functionality of a tool. The FP handles the material and knows the work context supported by that tool. To handle or manipulate materials, the FP uses operations specified in one or more aspects. To be able to support the work context, the FP manages a work state—the *tool's memory*.
- The *interactive part* (IP) defines the tool's user interface. More specifically, it accepts events, calls the FP, and controls the GUI presentation. To allow the IP to abstract from the concrete interactions and the window system used, it normally uses so-called *interaction forms*.

FP is the starting point

The T&M approach pays much attention to discussing tools and their domain interactions with the future users. In this context, it makes sense to design the FP of a tool first. The next step then defines the way a tool should be handled and the desired presentation. This development approach ensures that all tools are motivated by the application domain, and that the responsibilities of FP and IP are separated.

Notice that this guideline does not conflict with our use of prototypes, such as when decisions are taken about how to proceed in the design based on presentation prototypes. The reason is that these prototypes, essentially showing representation and manipulation aspects, are built on the basis of a domain tool design, forming our platform for discussions with future users.

Example

The EMS example

In our EMS example, the *material* is the room plan as a collection of rooms. The user wants to add a room to this collection. To do this, the user uses the *RoomMap Designer* (see Figure 8.27). This RoomMap Designer shows the rooms, and its menu can be used to create new rooms and edit existing ones. It creates and displays a new room when the user selects the *NewRoom* option from the *Actions* menu.

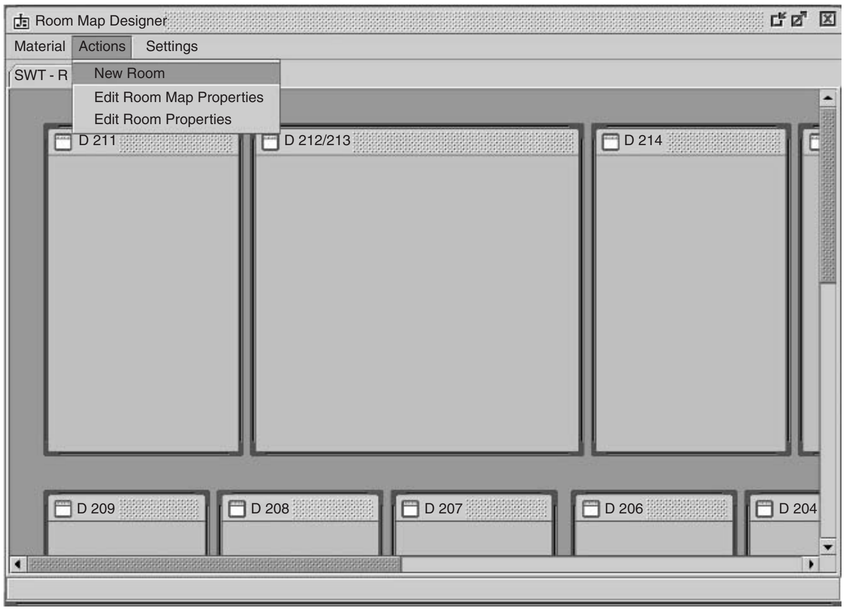


FIGURE 8.27 The RoomMap Designer from our EMS example.

The interaction form, `activator`, converts the system event (e.g., *NewRoom)* into a program event and sends it to the IP. The IP calls the FP's operation, `addRoom`. Next, the FP adds a new room to the room plan. This means that the user's action has led to the desired material manipulation. What's missing now is some feedback about the action's success, where the following construction guidelines are important.

The IP decides whether and how it wants to represent the modified room plan. The IP does not automatically add a new room to the rooms represented on the screen, and it does not represent that change immediately. It needs information that the FP actually created a new room. In addition, it does not know what standard name the FP used for the new room.

Displaying state changes

On the other hand, the FP does not automatically display the new room when it calls an IP operation, because the IP is responsible for this task. The FP does not have information about the display of a new room. It merely offers a list with information about all the rooms in the room plan. This information is available by calling a probing operation. The list is completely separate from the material, `RoomPlan`. The FP decides about the form it wants to use to create the list with room information from the room plan. Consequently, we need a feedback mechanism. The event pattern has proven useful in these cases, too (see Figure 8.28).

BACKGROUND: A TOOL AS A REACTIVE SYSTEM

If you look at it from the technical perspective, a tool is built as a so-called *reactive* system, that is, each tool activity is triggered by an explicit user action, such as clicking the mouse or pressing a key (see Figure 8.29, step 1). These actions are directed to the tool in the form of a stream of events (step 2). The tool reacts to each user action. To allow the tool to do this, these events are interpreted by the IP. The IP converts these events into FP calls or into a different presentation (step 3). The FP uses appropriate aspects to handle the material (step 4). After manipulating the material the FP anounces the state

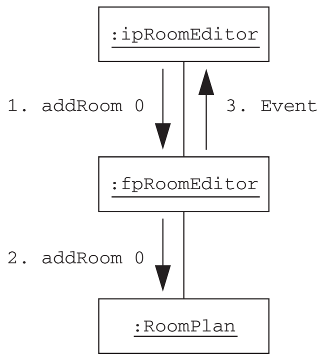


FIGURE 8.28
Feedback problem between FP and IP.

change (step 5). The IP reacts on the announced event and updates the presentation (step 6), which is recognized by the user as a program reaction (step 7).

TRADE-OFFS

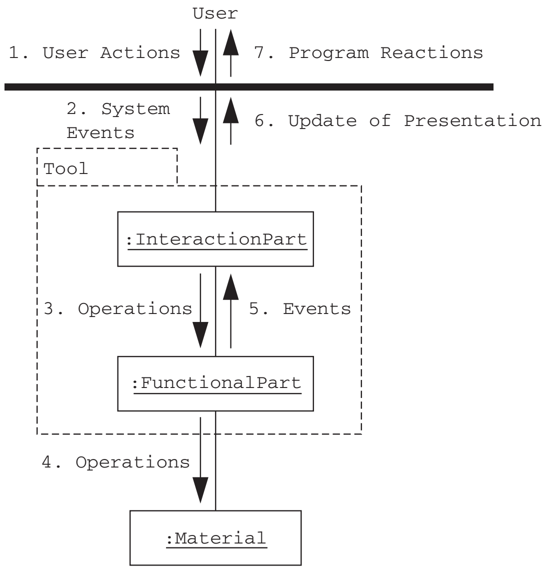
The division of FP and IP described so far specifies responsibilities within a tool that improve the legibility of your design. In addition, it supports two important design goals: flexibility and reusability.

Visual programming

A tool's user interface can normally be changed independently of its functionality. Given that the domain components of a system, that is its functional part, aspects, and materials, do not make any assumptions about the interactive environment in which they are embedded, they can easily be ported to different system platforms.

Many development environments offer attractive support for visual programming. GUI tools allow you to simply drag and draw GUI elements to your layout. You can then gradually add more functionality to these GUI elements. Remember that we have already mentioned some risks inherent to this design method (see Section 7.4).

FIGURE 8.29
A tool can be seen as a reactive system.



In connection with tools construction, this method normally leads to tools that are developed with the interaction part (IP) as its basis. Our experience has shown that this does not always take fully into account the domain contents of a tool. In fact, the tool could easily become a "window" on the material, which then appears to be manipulated directly. The tool itself often has no own elaborate domain functionality, beyond that of the material.

In addition, this method leads to a one-to-one relationship between tool and material. In such systems, all tools have the character of an editor. For example, you can use a tool to read attributes from a material, or set such attributes, and save changes to a material. Though these applications use object-oriented technologies for implementation, they do not utilize them to the user's benefit. Basically, such systems are not much easier to use than conventional software systems.

RATIONALE

Whenever you have to design and implement a medium-sized complex tool that has a good chance of needing frequent changes to its handling and interaction, this pattern provides the basic design and construction priciples.

WHAT NEXT

The design pattern *separating handling and presentation* shows how to further subdivide a tool's interaction part to encapsulate the actual GUI used.

8.7.1 Construction Part: Interactive Part (IP)

Characteristics of an IP

As already mentioned in the previous Section 8.7, we build a separate class that encapsulates all IP tasks for each specific IP. Before we continue discussing this construction approach, it will be useful to understand a few facts about IP.

An *interactive part* takes a stream of system events, triggered by user actions, as program events and interprets them. In this respect, it converts presentation-specific events (e.g., scrolling in a list or menu), while passing application-specific events to its FP.

An *interactive part* implements the manipulation and presentation of a software tool. It can call the widgets of the underlying window system or use generic input and output components, the so-called *interaction forms* (e.g., 1:n selection, activator), and call these interaction forms for representation and user input.

Returning to our discussion, the IP's tasks are specified by its responsibility for interaction within a reactive system. Each user action arrives over input channels as a system event (e.g., the user pressed the left mouse button at a specific point on the screen) at the event context of a tool. Modern systems use an event dispatcher, that is, the normal distribution mechanism for system events in window systems, to route system events to the appropriate event context.

TRADE-OFFS: SEPARATING TASKS BETWEEN IP AND INTERACTION FORMS

System events do not directly reach a tool's IP from the outside. As one option, the IP can use building blocks, the interaction forms introduced above, to react to interaction events. Interaction forms encapsulate the specific window system and convert system events into program events. *System events* are events generated by the system base, while *program events* are events generated by the application program. The general idea is to abstract the design of interactions from the specific system base. The IP can be thought of as a shell around interaction form objects. Naturally, the IP can also interact with the widgets of a window system. Even a combination of direct widget calls and the use of interaction forms is feasible.

The main task of an IP is to interpret incoming events and manage the user interface. Interpreting incoming events means that the IP decides whether or not an event is to be passed on as an FP call, or whether it entails a change to the GUI presentation.

A specific implementation of widgets or interaction forms and their arrangement based on a specific layout (horizontal, vertical, proportional, etc.) does not have to be handled by the IP in modern window systems. In fact, this part of a tool design can be specified separately by the use of a GUI builder or similar development tools or generators. The GUI resulting from this method is normally linked with the tool at runtime.

Trade-Offs: Separating Tasks between IP and FP

To maintain a fair division of tasks between IP and FP, the IP must not make any assumptions about the logical relation of FP operations or results from operation calls. This method ensures that the IP will not automatically change the GUI representation of information after an altering FP operation has been called. This is important, because it would mean that the IP knows the effect of an operation call on the FP state, thus breaking the semantic encapsulation of operations in the FP. Again, we must deal with the general feedback problem. An appropriate solution was introduced in our discussion of the event mechanism in Section 8.6.

Rationale

Whenever you have to design a complex tool with an elaborate user interface, it is useful to implement the interaction part as a separate component. When the application is supposed to work over a long period of time, you may consider abstracting interaction design from the actual GUI system by means of interaction forms.

8.7.2 Construction Part: FP

The construction approach discussed in this part uses a separate class that encapsulates FP tasks for each specific functional part (FP). First, let's look at the terminology.

A *functional part* (*FP*) implements the domain functionality of a software tool. It manipulates material via the interfaces of aspects.

An FP uses probing operations to provide information about its own working state and that of a material. It manages its own working state and the tool memory, depending on user actions and material states.

In the construction approach, the FP is the acting and probing part of a tool, where the services of a tool are implemented. The FP defines what application-specific activities can be performed by a tool.

Responsibilites of FP

The FP encapsulates the following design decisions:

- *Access a specific material*: When a material is to be manipulated, the IP always calls the FP, passing information to the latter. In turn, the IP uses a material-independent FP interface to obtain information for presenting a material. The IP does not directly access a material. In some cases (e.g., to handle complex tabular materials), the IP may be granted reading access to materials.

- *Changes to the material*: The IP can obtain information about the working state of a material to the extent that such information is provided by the FP, which means that a material never causes a representation to change.
- *Managing a work context between different materials and a tool*: The FP ensures for a work context that all participating materials are edited consistently. For this reason, the IP cannot at any time call the FP's altering or proving operations. It can call such operations only provided that the FP is in a suitable work state. Tests can be used at the interface to identify the FP's work state.

Trade-Offs: Separating FP and IP

In order to ensure the separation of interaction and function, the functional part should not make any assumptions about the handling and presentation of the interaction part. This is in line with our discussion about the responsibilities of the IP. An FP should know nothing about the ways and means by which a user interface and its interactions are realized. This, by the way, facilitates testing the functionality.

According to the general design principle, the FP never calls the IP directly to cause a change of the representation. Here we have another instance of the general feedback problem discussed earlier in Section 8.6.

Rationale

When you decide to implement the interaction part and functional part as separate components, you should consider this construction part.

8.8 THE SEPARATING HANDLING AND PRESENTATION PATTERN

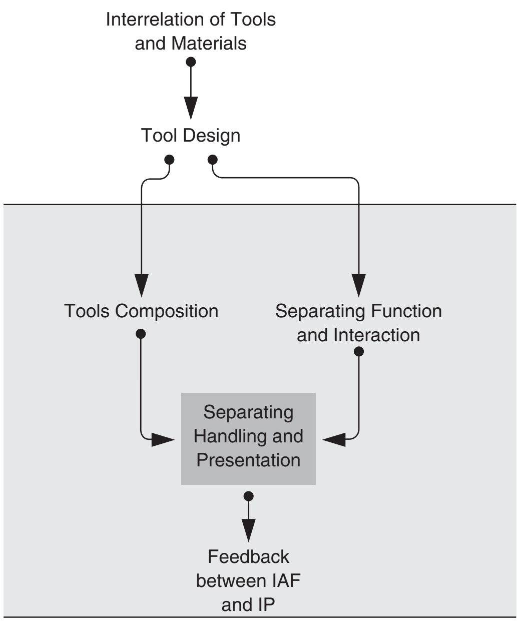

Interrelation of Tools and Materials
Tool Design
Tools Composition
Separating Function and Interaction
Separating Handling and Presentation
Feedback between IAF and IP

FIGURE 8.30

The *Separating handling and presentation* pattern.

Intent

This pattern details the essential concepts of designing the interactive part of a tool. Even if you should decide to implement the interaction of a tool as one construction unit, you should understand the general principle behind this pattern. This helps clarify the different concerns covered by an interaction.

Problem

We know from previous discussions on the *separating interaction and function* pattern that it is meaningful to separate the functionality of an interactive tool from its interaction. Since we are dealing primarily with the construction of interactive workplace systems, the interaction will normally be initiated by users and implemented on a graphical interface. There is a large number of commercial and public libraries and development components for the design of graphical user interfaces. We collectively call them *user interface toolkits*, or *toolkits* for short.

How can we encapsulate a user interface toolkit so that we reduce the dependence of our IP on a specific toolkit to a minimum?

Relate To

The design patterns *tool composition* and *separation of function and interaction* provide the conceptual background to understanding this pattern (see Figure 8.30).

Background: Separating Handling and Representation

The most important task of an application developer is to convert the domain functionality of a tool into interactions and then use the latter in combination with graphic components to implement suitable handling and presentation forms. Such graphic components are readily available in many toolkits, such as Motif, TCL/TK, or the Swing framework. These graphic components are often called *widgets*, and they are used to create graphic user interfaces. Using these turnkey components to build your user interface has several benefits. One major benefit is that you can quickly implement complex user interfaces at relatively little programming cost. In addition, turnkey components contribute to a uniform look and feel for your user interfaces, facilitating the use of your application system. For example, Java's Swing framework even let's you replace the look and feel of a GUI without having to change the toolkit.

Let's first see how the components of a toolkit may be used to build tools (see Figure 8.31).

The user interface of an IP is built by combining elements from the toolkit. In doing this, we have to observe the following important points:

Designing the GUI of an IP

- Each toolkit makes assumptions about the control flow within an application. This may conflict with your ideas about the tools to be implemented.
- The way widgets are linked with the IP is defined by the toolkit developers. This means that, depending on the toolkit, different system events may have to be linked with callback operations. These callback operations have to be implemented in the IP.

Another motivation for loose coupling of IPs to the toolkit you use is the latter's volatility. The historical development of toolkits has shown that the interfaces of toolkits have changed, often to the point where they were actually completely redesigned, while other toolkits disappeared from the market. One example is Java: the

Zull-08.qxd 31/8/04 2:35 PM Page 229

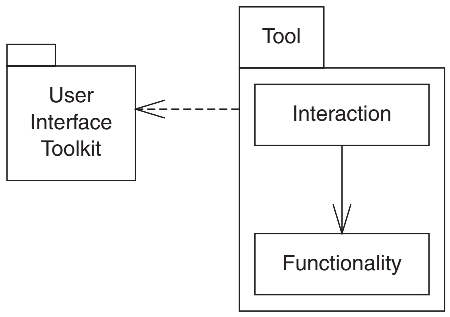


FIGURE 8.31
Linking a tool to a GUI toolkit.

introduction of the Java Foundation Classes and the related Swing toolkit almost entirely replaced the former GUI library, the Abstract Windowing Toolkit (AWT).

Exaple for a toolkit library

Figure 8.32 shows a segment of the *Swing* library's class tree in Java 2. Important features of this library include components arranged in an inheritance tree. Notice that inheritance is used mainly to be able to reuse an implementation, as in other toolkits. Such inheritance hierarchies generally violate the type-subtype hierarchy in many ways. For example, you often find operations that can be called in many but not all subclasses in the root class of the toolkit. In some cases, calling such operations can either lead to a runtime error, or an exception is thrown, or the call does not lead to any result at all.

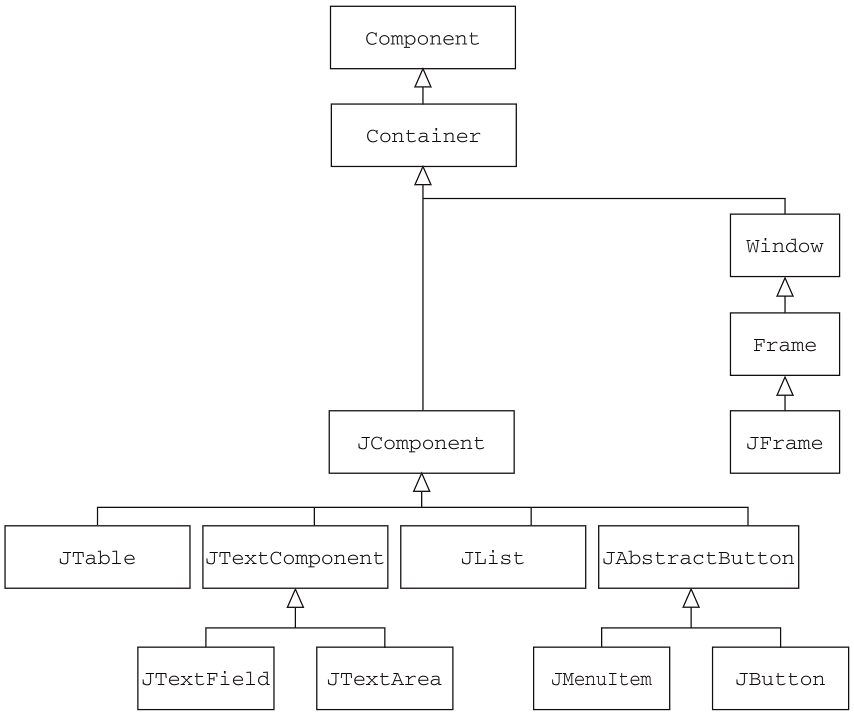


FIGURE 8.32
Components of the Swing library (excerpt).

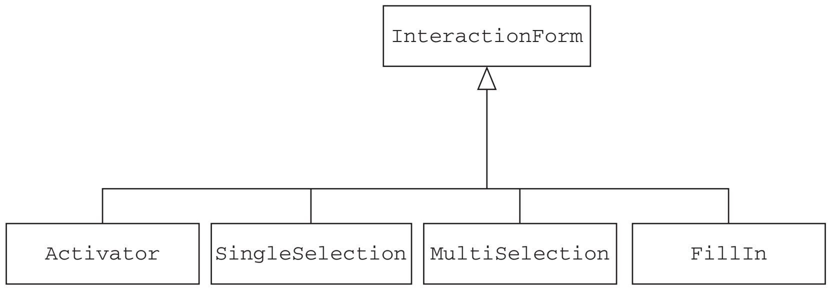


FIGURE 8.33

Examples of common interaction forms (IAFs).

SOLUTION

When building the IP, we want to ensure that it is not dependent on the toolkit we use. Rather, we want to abstract the GUI design from a toolkit. For this purpose, we introduce *interaction forms*.

The solution proposed in this section reduces the dependence of an IP on a specific toolkit.

We encapsulate different types of potential manipulations for a tool in interaction forms. Next, we compose an IP from the instances of our set of interaction forms. And finally, we ensure that the IP exchanges only domain values with its interaction forms.

Let's first look at some terminology to better understand our idea.

- An *interaction form* (IAF) is an abstract form of handling a tool. It represents a domain way of using a tool and has no side-effects on that tool.
- An IAF represents and returns only domain values, which means that is has no global effect. An IAF is used by an IP. An IAF's interface does not make assumptions about a toolkit, which means that the IP is totally independent of a selected toolkit.

Figure 8.33 shows examples of common interaction forms (IAFs).

When building an IP, developers normally select interaction forms based on their decision about how the functionality of the FP should be converted into useful user interactions. Developers then create one object out of the set of available interaction forms for each type of user interaction.

All information represented by an interaction form on the user interface, and the results this IAF returns, are domain values. These domain values are supplied and accepted by the IP. This allows an interaction form to run stateless and without side-effects.

Of course, an interaction form has to be represented at the user interface. On the other hand, the type of widget used from a toolkit to represent an IAF at the GUI is not important for an interaction. Also, the interaction form is not interested in the size and position in which it is represented. This information can be encapsulated in presentation forms. To better understand this, we discuss the following terminology:

A *presentation form* (*PF*) implements the specific representation and handling of a domain interaction form (IAF) at a tool's user interface.

Presentation forms encapsulate widgets of a GUI toolkit, implementing a protocol that allows us to link it with interaction forms.

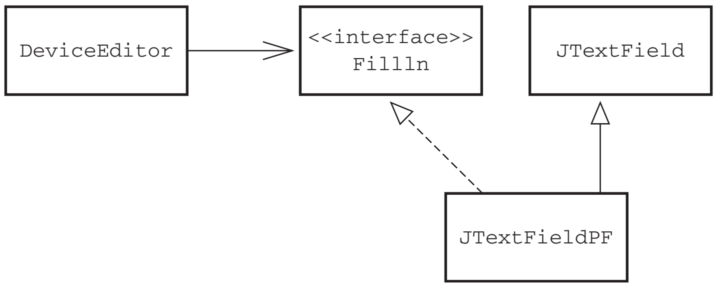


FIGURE 8.34

An interaction form, Fillin, is represented by a JTextFieldPF in the Device Editor.

Presentation forms are managed outside a tool, but they are linked with interaction forms inside a tool.

Depending on the GUI toolkit we use, the widgets of the toolkit could also assume the role of presentation forms. In this case, there is no need to program additional presentation form classes.

Example

The EMS example

Returning to our EMS example, we want to see how a `DeviceDescription` is filled out in the `DeviceEditor`. Figure 8.34 shows an interaction form used to fill in a value, `FillIn`. We used such an interaction form in the `DeviceEditor` tool, and selected a matching presentation form, `JTextFieldPF`. The value for `DeviceDescription` is passed to the interaction form and read from it after an interaction.

Trade-Offs

Interaction forms and presentation forms represent a good way to abstract from concrete toolkits. The tool is completely decoupled from the concrete GUI representation. Presentation forms can be easily adapted to toolkits as they change, without a need to change the tool interaction. In such a case, the interactive part does not have to be changed. In addition, interaction forms decouple the material from its interactive manipulation.

IAFs and domain values

To allow the use of interaction forms independent of a context, they have to exchange domain values or basic data types with the IP. This is the way to ensure that the IP will have full control of what an IAF represents and which results it returns.

The set of interaction forms that can be identified no longer depends on the options available in a specific toolkit. For example, it does not matter for interaction forms whether or not there is a special widget available on the platform used. Most toolkits have to deal with this problem, if they really want to be portable. Either they must reduce their set of widgets to the smallest common set for all supported platforms, or they must reimplement exotic widgets not directly supported by a specific platform, which is expensive.

Encapsulating the system platform

The presentation forms allow us to implement a presentation of the interaction required by a tool for the platform used. For example, Microsoft Windows systems have floating menu bars and toolbars in application windows, while the Macintosh displays menu bars and toolbars for the active application in a generic program bar in the top part of the screen. Similarly, buttons were represented differently in Windows 95 or Windows 3.1. There are, however, still some fundamental problems in implementing complex tree or table interactions independent of the specific GUI toolkit.

By separating the interaction from the presentation form, we overcome the direct coupling to single, specific GUI widgets from within a tool, so that we can respond much more flexibly to changing presentations and toolkits. Of course, the independence of a specific toolkit does not come for free. The cost is that we lose part of the control over the widgets of a GUI toolkit. For example, we can no longer use interaction forms to directly control a specific representation from within a tool (e.g., to set the color of button labels). However, experience has shown that a very detailed control over widgets is normally not necessary for most tools. Representation details, such as font colors, can normally be defined statically in the GUI builder. For tools where exact control is required, we may have to access the presentation forms or widgets directly. In summary, the interaction forms concept shows clearly which tools depend on a specific GUI toolkit and which don't.

Rationale

Whenever you need a decoupling of a tool's interaction and presentation from the concrete GUI toolkit you may use this pattern.

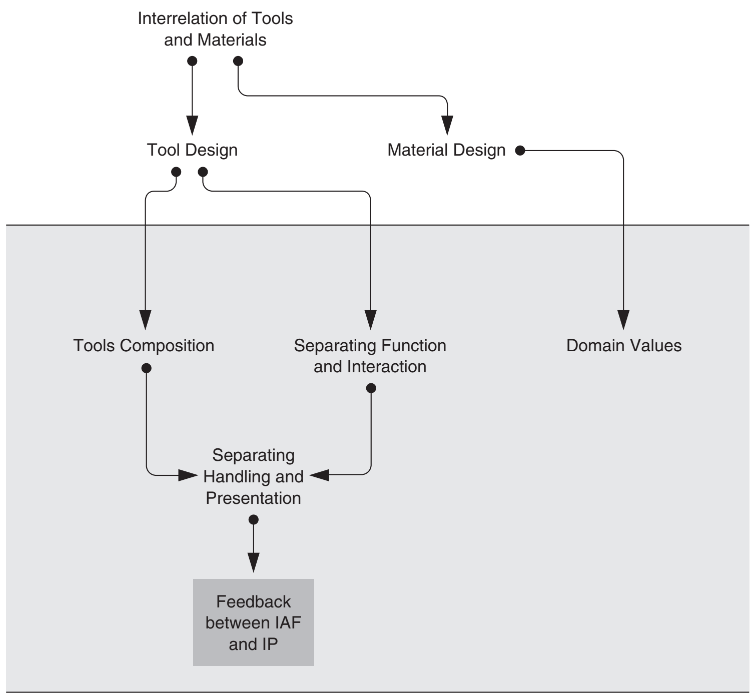
What Next

The design pattern *feedback between interaction forms and IP* shows how to implement the appropriate reaction mechanism between the interactive part and its interaction forms.

8.9 THE FEEDBACK BETWEEN INTERACTION FORMS AND IP PATTERN

FIGURE 8.35

The Feedback between interaction forms and IP pattern.


Interrelation of Tools and Materials
Tool Design
Material Design
Tools Composition
Separating Function and Interaction
Domain Values
Separating Handling and Presentation
Feedback between IAF and IP

INTENT

We introduce a solution that uses the command pattern (Gamma et al.) to solve the feedback problem between interaction forms and IP.

PROBLEM

User actions such as mouse or keyboard movements are accepted by the window system. The window system converts these actions into system events, which are initially passed on to the representation and interaction forms. For this purpose, representation forms are normally linked to the appropriate widgets of the window system over a callback (or over subclasses and the bridge pattern). Interaction forms convert system events into program events by alerting the IP. This requires an interaction form to be linked to the IP. The IP can then identify the FP operations that must be called (see Figure 8.36).

Interaction forms, however, must not know the specific type of the IP they collaborate with, otherwise we could not use interaction forms with different IPs. As a consequence, we would have to reimplement all interaction forms for each new IP. As in the feedback between FP and IP (see Section 8.7), we have to solve the problem of loose coupling between interactive components.

How can we implement a suitable feedback mechanism between interaction forms and IPs to allow loose coupling and meet the special requirements of an IP at the same time?

RELATE TO

The design pattern *separating handling and presentation precedes* this pattern conceptually (see Figure 8.35).

SOLUTION

We use the command pattern to bind the interaction forms to the IP, achieving loose coupling.

One way to solve this problem is to define suitable command classes (e.g., `Command, DropCommand`) for the program events sent by interaction forms to IPs. Another way is to have a generic command class that identifies the appropriate interaction form. We can build a command class so that the interaction form will feed it with appropriate domain values (see Sections 2.6.5 and 8.10). These domain values are required to further process the events in the IP.

If an IP then wants to obtain information about a user action from an interaction form object, it generates a command object. The IP passes a reference to the operation to be called to this command object; this operation will be called as soon as the command object is activated. Subsequently, the IP registers the command object with the corresponding interaction form object. The different interaction forms each accept a set of command objects matching their interactions.

The interaction form object activates the command object registered with it when the corresponding user action occurs. This causes the command object to call the IP's operation registered with it. In Java, a typical implementation of this pattern is based on an interface that contains an `execute` method. Next, the interaction form calls this operation from the registered command object. This allows the IP to implement this interface and register directly as a command object, using anonymous inner classes of Java.

If a user action means the input or selection of domain values (e.g., a user enters an account number), then the IP requires these values. A command class used for such user actions expects a parameter from the interaction form. The operation the IP passed to the command object has to accept arguments from the command object or probe the IP.

The EMS example

Example

Let's look at the `DeviceEditor` tool in our EMS example. This tool allows us to save changes to a device. The `DeviceEditor` tool includes an interaction form, `Activator`, for this user action. This interaction form is implemented by a button, that is, a presentation form, in the GUI. `Activator` is an interaction form that accepts only one type of command objects, namely `Command`. It is a command class that does not expect any parameter.

The sample code in Figure 8.37 was taken from a Java implementation of the `DeviceEditor`. The constructor creates the interaction form object for the button and the command object. In addition, the IP of the `DeviceEditor` tool registers the command object with the button.

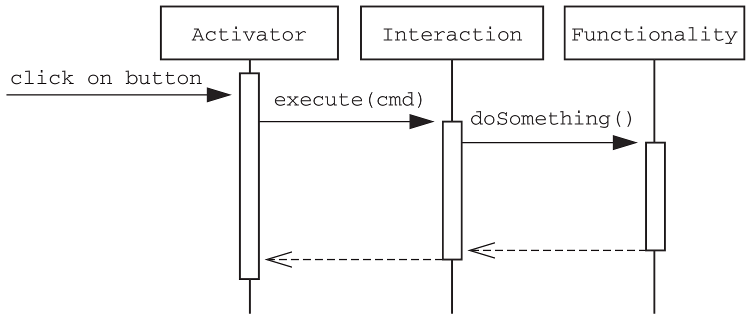
When a user clicks the button to save device information, then the button triggers the command object registered with it, and the command object calls the `storeDevice` operation in the IP of the `DeviceEditor` tool.

Trade-Offs

Implementation variant: passing operation names

One possible implementation variant to the solution introduced in this pattern would pass only the name of the called operation as a character string to the command object as callback. Java lets you use CoreReflection to call such a named operation at runtime. But this solution is not as performant and less typesafe. In other languages, such as C++, you have to use a method pointer or a "normal" class.

FIGURE 8.36



Flow of events in a tool with interaction forms.

```
public class DeviceEditorIP extends InteractionImpl
{
  //constructor
  public DeviceEditorIP(IAFContext iafsContext, ...)
  {
    // create command object
    _storeCommand=new Command()
     {
       protected void doExecute ()
       {
         storeDevice(this);
       }
     };
    // get interaction form
    _store = (ActivatorIAF) iafsContext.interactionForm
                          (ActivatorIAF.class,"store");
    // attach the command object to the interaction form
    _store.attachActivateCommand(_storeCommand);
    ...
  }
  // callback operation for the command object
  public void storeDevice (Command cmd)
  {
    ...
  }
  // attributes
  // interaction form
  private ActivatorIAF _store;
  // command object
  private Command storeCommand;
}
```

FIGURE 8.37

Using the command pattern for interaction forms.

Multiple registration of command objects

An IP often represents the same interaction form in different places of a tool's GUI. In our preceding example, device information entered in the `DeviceEditor` tool could be saved by clicking a button or by selecting a menu option or by typing a value in a popup menu. In this case, the IP can register the same command object with three interaction forms. Regardless of which of the three interaction forms activates the command object, all three cases would call the same operation of the IP to save device information.

One important benefit is that undo and redo mechanisms are prepared so that they can be implemented directly in the command object, where the relevant context for undoing or redoing the command can be stored. Previously executed command objects are stored in a list that maintains the command history. This means that we can implement unlimited sequences of undo and redo by traversing this list.

Zull-08.qxd 31/8/04 2:35 PM Page 236

RATIONALE

When you use interaction and presentation forms, you will have to solve the feedback problem that this pattern addresses.

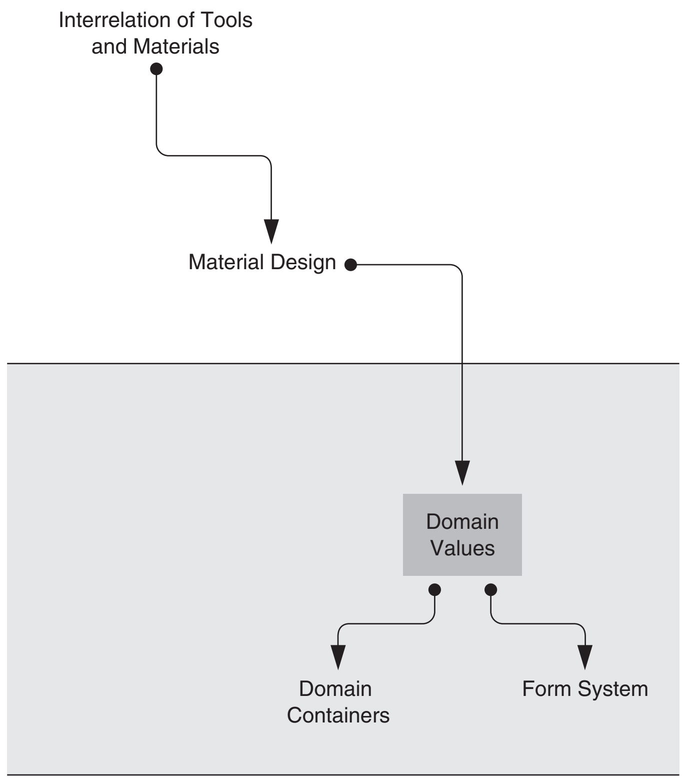
WHAT NEXT

The design pattern *domain values* discusses how to extend the basic data types provided by object-oriented programming languages. Domain types are needed for decoupling interaction forms from the interaction part, and the interaction part from the functional part.

8.10 THE DOMAIN VALUES PATTERN

FIGURE 8.38

The domain values pattern.



INTENT

For application-oriented program design it is important to use the different value types relevant in an application domain. This pattern explains how to design and implement these domain values.

Problem

When using object-oriented programming languages to develop large systems, we quickly notice a major shortcoming. The standard data types available in the languages are not sufficient to implement the full bandwidth of basic values for an application domain. Even a value as simple as a money value cannot be elegantly mapped on a standard data type. The type `REAL` available in most languages has an arbitrary number of decimal places, while we need exactly two to represent a money value. Moreover, calculations on `REAL` numbers lead to rounding errors, which are unacceptable for money values in commercial applications. In addition, the arithmetic operations defined for standard data types are generally not suitable for arithmetic operations required in an application domain. For example, there are precise calculation rules for the conversion between currency values that differ from `REAL` arithmetic.

At the same time, we cannot simply use classes and objects to extend the set of primitive data types, because instances of classes have reference semantics, while data types have to obey value semantics (see Section 2.6.5). Therefore,

How can we extend the primitive data types of object-oriented programming languages to user-defined domain values based on value semantics?

Relate To

The conceptual pattern *material design* is the conceptual context for this pattern as domain values are a consequence of our application-oriented approach to modeling (see Figure 8.38).

Solution

We implement domain values in an object-oriented language, similarly to other user-defined types, using classes. However, in the implementation we ensure that the instances of these classes behave like values.

Our new data types for domain values can be divided into two groups by general typing principles: *elementary domain values* and *composite domain values*. An elementary domain value (e.g., a time of day) represents exactly one “atomic” domain value, which is implemented internally as one or more encapsulated standard data types. At its interface, it offers only operations that are relevant for the domain to handle values of that type (e.g., adjust the time).

Composite domain values build on elementary and other composite domain values. They group different values into a new structured value (e.g., a period consisting of two time values). At their interfaces, there are operations that compose domain values from elements and access individual elements (constructors, selectors), in addition to the domain operations.

In addition, we can classify values based on their cardinality, that is, *finite domain values* and *infinite domain values*. Finite domain values correspond to an enumeration type, which means that they have a fixed number of values (e.g., days of the week). In contrast, infinite domain values cannot be limited (e.g., date), or they extend over a range that can be divided into an arbitrary number of parts (e.g., time).

BACKGROUND: DOMAIN VALUES

Programming languages normally offer a limited set of data types that follow value semantics. These standard data types, such as `INTEGER`, `REAL`, or `BOOLEAN`, are either related to mathematics or oriented to implementation aspects. An attempt to introduce domain value types to a programming language has not proven to be very helpful, as we know from some so-called fourth-generation languages. In contrast, functional programming languages, such as Miranda, are based on a totally different approach. They use powerful type constructors for user-defined types based on value semantics. Unfortunately, object-oriented programming languages have a serious shortcoming in this respect, as they offer only the class concept for user-defined types. It appears that language designers forgot the importance of domain-value data types for the application system to be developed.

RATIONALE

For systems based on the T&M approach, domain values are an essential concept for introducing domain-motivated values as the "atoms" of programming.

WHAT NEXT

The design patterns *domain-specific containers* and *form system* use domain values as an implementation concept.

8.10.1 Construction Part: Domain Value Classes

We use classes to implement different domain values. They are arranged in a class hierarchy, where the top class is the `DomainValue` class. This superclass is an abstract class, specifying the following operations for its subclasses:

- Operations that test a passed parameter for a valid external representation of a domain value. Parameters can be strings or number values as a type of the representation (e.g., `InterestRate.IsValid("2,5")` or `InterestRate.IsValid(2.5)`).

  For composite domain values, this test refers to the individual elements. These operations have to be implemented in the interface of a class or as class operations, depending on the programming language used. If classes are first-order objects, like in Smalltalk, and class methods are inherited, then these operations should be implemented as class methods. In contrast, if using C++, we would implement a prototype object. Clients would call this prototype object, which represents the class object, to do the test there. Java let's us use inner classes to statically embed factory classes in each domain value class, and to accommodate methods in these factory inner classes. Also, we could use the *Singleton* design pattern (Gamma et al.) to ensure that there is one and only one factory for each domain value type.
- Operations like `hasFiniteNumberOfValues()` or `getAllValues()` allow us to request the set of values valid for a domain value type, if this set is finite. These operations show us whether we are dealing with a finite or an infinite domain value. For example, this aspect is relevant when representing domain values by separate interaction forms. Alternatively, we could implement a special class, such as `Enumerable`.

- The operation `toString()` returns the value of a domain value in the form of a string. Most window systems accept only standard data types at their interfaces. Therefore, a domain value should be able to return its value in a string representation.
- Both C++ and Smalltalk let us overload operators, so that we could define relational operators (e.g., `<,<=,>,>=,==,and !=`) for domain values in the `DomainValue` class. Notice that `==` and `!=` are semantically required for all data types, as suggested by Hoare. In any event, we have to ensure that these operations have tests to make sure that two values can actually be compared. It can also prevent relational operators from executing in domain value classes with objects that cannot be compared (e.g., it wouldn't make much sense to use the greater-than operator on `Color`). Also, comparing with objects of the superclass is invalid.

Depending on the application domain, a class like `DomainValue` will apparently be the root of different inheritance structures of domain value classes (see Figure 8.39). Each of the domain value classes has to implement the operations specified in the superclass, `DomainValue`, for their specific values. To ensure that the value of a previously created domain value object cannot be changed, for example in C++, no public procedures that change the internal state should be offered. Instead, there should be one or more special constructors with corresponding parameters available.

To ensure that domain value objects are used according to value semantics, there are two different solution variants, more or less suitable, depending on the programming language: immutable and mutable domain value objects. The construction parts described in Section 8.10.2 to 8.10.5 explain these solutions.

DISCUSSION: USING DOMAIN VALUES

The T&M approach uses domain values as elements of materials. If you think of a material as a tree of objects, then the leaves of this tree are domain value objects that in turn, use standard data types for their implementation. Therefore, we have to build appropriate domain value classes for each special application domain, so that these domain value classes can be used as elementary components of materials.

Domain values for FP-IP interaction

When building tools on the basis of the FP-IP pattern, domain values are the only objects passed from the FP to the IP, in addition to standard data types. In any case, interaction forms are called with domain values. This explains another reason

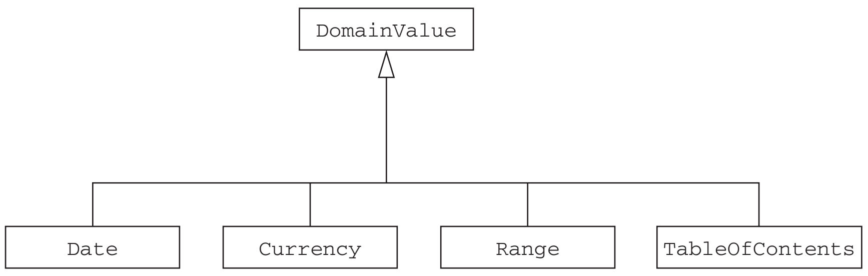


FIGURE 8.39
Domain value classes.

why domain values should be built by value semantics. Only the function and not the interaction should be able to directly manipulate a material (see Section 8.4). For this reason, an FP must not pass references to a material to the IP. If the FP passed domain values to the IP, then the IP would receive copies of the domain value objects, at least from the conceptual perspective. If a user changes a value, then the IP passes this value to the FP, and the FP will update the material to reflect this change.

Domain values for IAFs and PFs

To display domain values in the IP, we can use specialized interaction and presentation forms (see Section 8.8). These interaction forms for domain values are built to match a domain value class, and they can directly check user input for validity. This prevents that, for example, checking a date input for correct syntax is left to the FP, which then activates a feedback mechanism to alert the user. Accordingly, we let domain values do such checks that are not related to context. This means that domain values offer the means to let an interaction form test user inputs for validity. Domain values will then only have to be tested for consistency in the respective context. For example, if users have to enter a date of birth and a wedding date in forms, then the FP or the form itself could check whether the date of birth is older than the wedding date.

8.10.2 Construction Part: Immutable Domain Value Objects

This solution variant implements domain values as *invariable* objects so that changes to domain value objects are excluded. More specifically, we do not allow altering operations at the interfaces of domain value classes. This solution can be used in all object-oriented programming languages. It can be implemented both by the dynamic creation of objects (in Java or Smalltalk) and by the static creation of objects (in C++).

TRADE-OFFS

Drawbacks of solution

Unfortunately, this solution has the following drawbacks:

- New memory space has to be requested for each change to a domain value object.
- Programming with domain value objects from classes that do not offer altering operations is not elegant, because we have to create a new identifier for each change, and then assign the result of a change to that identifier.

Using flyweight *or* factory

The storage problem inherent in this solution can be solved by using the *flyweight* and *factory* design patterns. These design patterns ensure that we do not have to create a technical copy for each change. The *flyweight* pattern ensures that there will always be only one object for each individual value, while the *factory* pattern creates objects by means of appropriate operations. A factory stores all previously created domain value objects. When a new domain value is requested, then the factory verifies whether or not an object has been created for this domain value. If it finds such an object, it will return this object; otherwise, it will create a new one. This ensures that different components using the same domain value can share the same domain value object.

The *flyweight* pattern requires that all previously created domain values are stored by the factory, so that the factory obtains a reference to a domain value object, even if an object is no longer used. This means that a garbage collector cannot remove a domain value object from the memory as long as this factory exists. This leads to an expanding system. Java lets you implement an elegant solution to this problem. You can use weak references to manage objects, as in a `WeakHashMap`. Objects in this hash map will be removed by the garbage collector when nothing in the system references them, except the hash map. The *flyweight factory* can use Java's `WeakHashMap` to store domain value objects and let the garbage collector remove them, as appropriate.

Example

The EMS example

Figure 8.40 shows how we use several elementary domain values, such as `Date` and `EmployeePosition`, in our EMS example. `TableOfContents` can be thought of as a composite domain value that includes `ThingDescriptions`. Moreover, two device objects, that is, `Device: PC-1234` and `Device: PrinterMax`, share the same date from the *flyweight* pattern.

The flyweight pattern is suitable to manage domain value objects, in particular for domain values that have a finite and relatively small value range, such as days of the week or month. Domain value objects for these classes are often created when the application starts and deleted when the application is closed.

Rationale

Use this approach for domain value types with a limited number of values and a high number of occurences of these values in your programs. Check whether the programming language you use offers simpler means for "immutable" objects.

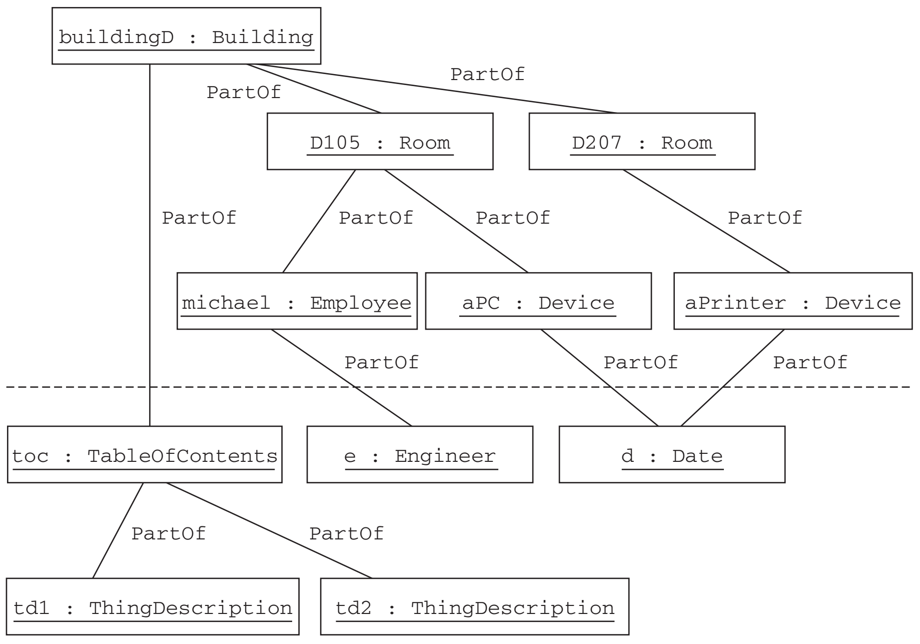


FIGURE 8.40 Using domain value objects in the EMS room plan.

8.10.3 Construction Part: Mutable Domain Value Objects

To avoid problems inherent in the program-specific handling of immutable domain value objects, we can choose a solution variant that uses *mutable* domain value objects.

In this solution, the domain value classes provide all altering operations required. If the domain value objects created by these classes should be used exclusively by value semantics, then the application programmer has to either copy the domain value object before each altering operation, or the domain value classes have to use an internal copying mechanism.

TRADE-OFFS

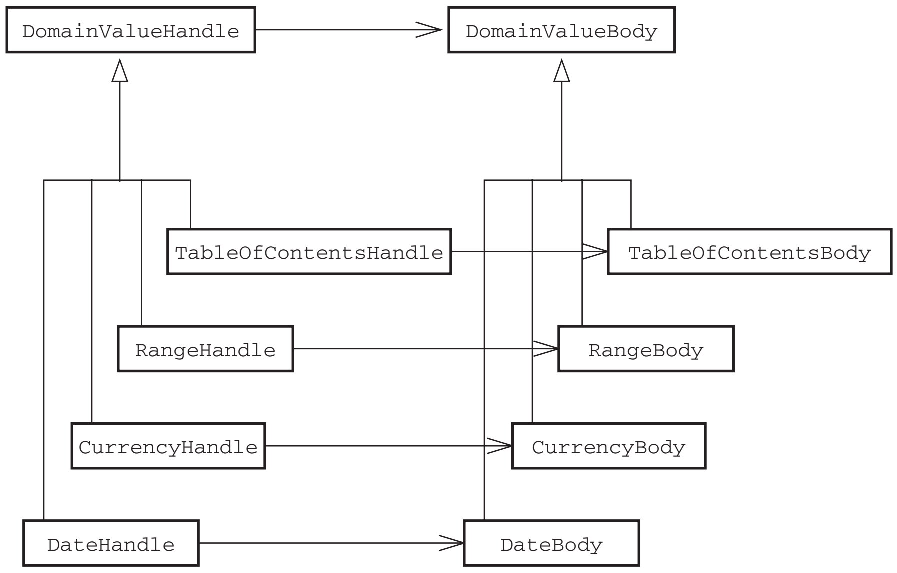
The first of these two variants is basically workable. We can use the *flyweight* pattern to implement an internal performant copying mechanism. However, we have to prepare programming guidelines to ensure that the application programmers will really create a copy before each change to a domain value object.

Using the body/handle pattern

We can use the *body/handle* pattern to implement the copying mechanism for objects within our domain value classes. The body/handle pattern is extensively described in the literature; Gamma et al. refer to it as "copy-on-write" in connection with the *proxy* pattern. The body/handle pattern is based on the idea of building a second class tree, a so-called "shadow tree," in addition to the existing class tree, with both trees having an identical structure. The class tree for domain values is the body tree, and there is an additional isomorphic handle tree. Figure 8.41 shows an example.

Notice that this solution is well balanced; it let's you implement your storage management easily by counting the references in the body object, especially in C++. The major drawback of this solution is that handle objects must not be used by reference

FIGURE 8.41 Domain value classes, using the *body/handle* pattern.



"from the outside." In addition, this rule has to be established in programming guidelines. Another drawback is that parallel inheritance hierarchies have to be maintained.

Rationale

Use this approach with a language like C++ or when you opt for a simple but potentially error-prone solution.

8.10.4 Construction Part: Implementing Domain Values as Streams

We often find simple values, similar to elementary units, used as domain values in many applications. This hides the fact that complex domain values can be large and extensive. One good example for such a domain value is a table of contents. When you build a table of contents for a selection from a database, you at least have to deal with its size and ask yourself whether it is necessary to keep the complete composite domain value in memory.

It is often a better idea to implement such a domain value as a stream. At the public interface, such streams often present themselves as usual domain values (with the difference of throwing additional exceptions). However, they are internally implemented so that they contain only the amount of data required to meet the interface. If more elements of a composite domain value are required, then these are generated or loaded "on demand."

Rationale

Use this approach with very complex and exceptionally large domain values.

8.10.5 Construction Part: Domain Value Types by Configuration

When analyzing an application domain, we often find a large number of similar domain value types, such as different types of amounts. These domain values often differ only in their value ranges and not in their operations. Such differences can basically be expressed by inheritance. For example, `CreditAmount` and `DebitAmount` would then be subclasses of `Amount`. However, subclassing can lead to a large number of domain value classes, making maintenance more difficult.

One construction approach uses dynamic configuration of domain value classes. More specifically, we develop only domain value classes for domain values with different operations. Then we express the difference in value ranges by configuration, thus saving the differences in configuration files or databases.

When creating a new domain value object, we additionally define an identifier (e.g., `String`) for the configuration to be used. We would then develop only the class `Amount` and configurations for `CreditAmount` and `DebitAmount`. These configurations will specify that both the credit amount and the debit amount must be positive values.

Rationale

Use this approach when a considerable number of domain value types have the same interface and differ only in value ranges.

8.11 THE DOMAIN CONTAINER PATTERN

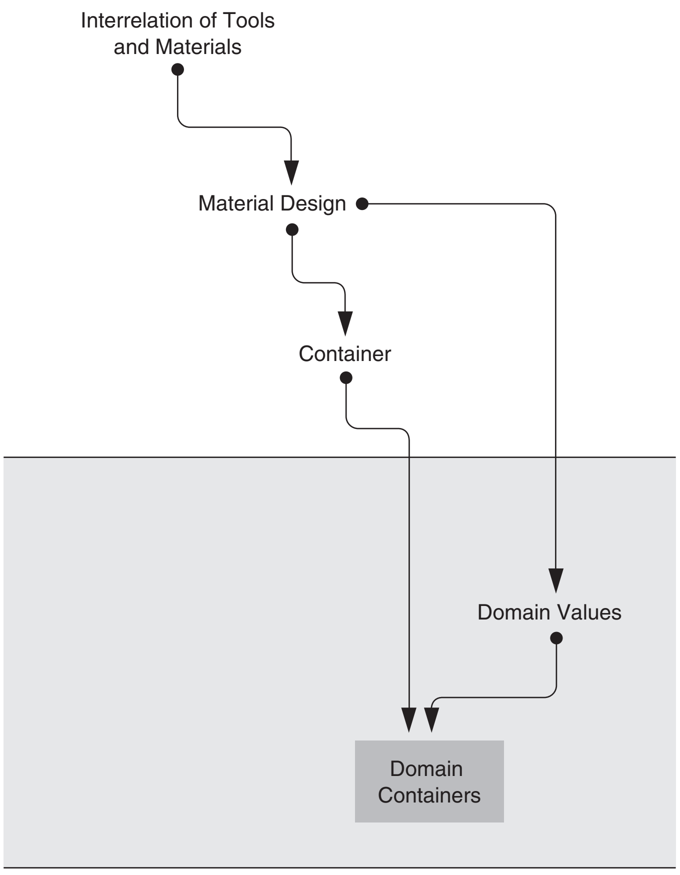


FIGURE 8.42

The domain container pattern.

INTENT

Design domain containers on the basis of the containers you have identified in the application domain and implement them using technical containers.

PROBLEM

Containers serve to hold objects. When we are modeling based on the T&M approach, we distinguish between two types of containers: domain and technical containers.

Domain containers are designed on the basis of existing containers (e.g., folders) that we identified when we analyzed the application domain (see Sections 3.5.16 and 7.7). In contrast, *technical* containers correspond to traditional data structures, such as lists, arrays, or complex technical structures like `WeakArray` in Smalltalk. They are normally used to implement domain containers.

We think that it is no longer necessary to build our own technical containers, since all common object-oriented programming languages include class libraries with powerful and standardized technical containers, so that we shouldn't expect major interface changes. Therefore,

We want to model domain containers as a special type of materials that provide a place to store other materials and define an order principle for these materials.

Relate To

The conceptual pattern *container* explains the domain-related context for this pattern. The design pattern *domain value* explains a useful implementation concept used here (see Figure 8.42).

Background: Domain Container

The object-oriented methodology discussion has not taken domain containers into account. Most programming manuals and class libraries model and implement exclusively technical containers.

In general, there is no clear view of what domain containers are and how they can be designed. In contrast, we often find that collections and orders of materials are very important in daily work. However, when we implement existing container concepts in our application system model, we often map more than the container functionality, that is, we also try to utilize the potential of interactive software.

Solution

We build a domain container as an independent domain-specific object, which has the character of a material, but shows some specific interactions.

The elementary interactions of domain containers include storing and editing a set of materials, and accessing elements of this set. Usually a domain container provides a table of contents, which it updates internally.

Containers are materials that can contain materials. This means that we can add other containers to a container, structuring the contents of a container.

Like any other material, domain containers have an identity and normally user-defined names. They are active in the sense that they have their own "navigational model," that is, a specific way that we can navigate from one element to another. Internally, they manage their elements themselves. These interactions can be generally modeled for domain containers.

Example

The EMS example

In the context of our EMS system, we could think of the room plan as a container for rooms. The room plan would be consistent when all the rooms it contains are consistent (i.e., no excessive occupancy by persons or devices). This consistency check can be elegantly implemented in the room plan. It will then be available for all tools that use room plans, so that we do not have to reimplement it as we add new tools.

Trade-Offs

Modeling real-world containers

We do not assume that we can directly transfer containers from an application domain to the software model. For example, we want to have large domain containers manage their own tables of contents. Domain containers can also encapsulate a persistence mechanism, if there is no independet persistency service.

When building domain containers, we often observe that there are certain interactions not found in containers existing in the application domain due to physical limitations. We usually will add such interactions, such as consistency checks, to domain containers.

You can think of a domain container as a means to collect a set of materials. We often find that a domain activity or operation should be executed not only on single elements but on the entire set of materials. This is easy to model in containers. Though we use technical containers to implement domain containers, we will not derive our domain containers as a specialization of technical containers.

RATIONALE

For systems based on the T&M approach, which show characteristics of a workplace, domain containers are usually an essential element.

8.11.1 Construction Part: Using Technical Containers to Implement Domain Containers

In cases where we have access to a technical container library, we may consider a way to relate domain containers to technical containers. Should our domain containers inherit from or use technical containers?

In our approach, technical containers always serve to implement domain containers. Consequently, when implementing domain containers, we can use the full choice of variants and interactions offered by technical containers, regardless of the interfaces our domain containers should have.

BACKGROUND: INHERITANCE IN THE T&M APPROACH

It should be obvious from the discussion so far in this book that we should use an inheritance relationship in our domain modeling only provided that the two classes to be linked have a subclass-superclass relationship. A domain container should not be seen as a subtype of a technical container. In addition, its interface should primarily have operations motivated by the application domain, which means that they are not always compliant with the interface of a technical container. For this reason, we use a use relationship between domain and technical containers.

EXAMPLE

The EMS example

To better understand this idea, let's look at room plans as domain containers in our EMS example (see Figure 8.43). A domain container, `RoomMap`, uses a technical container to manage the set of rooms. The class `RoomMap` knows nothing about the concrete type of the technical container (`HashMap`) used. Instead, it works with the specific technical container exclusively over the abstract interface of a `Map`. In addition, the `RoomMap` can generate *iterators* if needed, and use them to traverse the set of rooms.

RATIONALE

Whenever a suitable class library for technical containers is available, you should use it to implement domain containers.

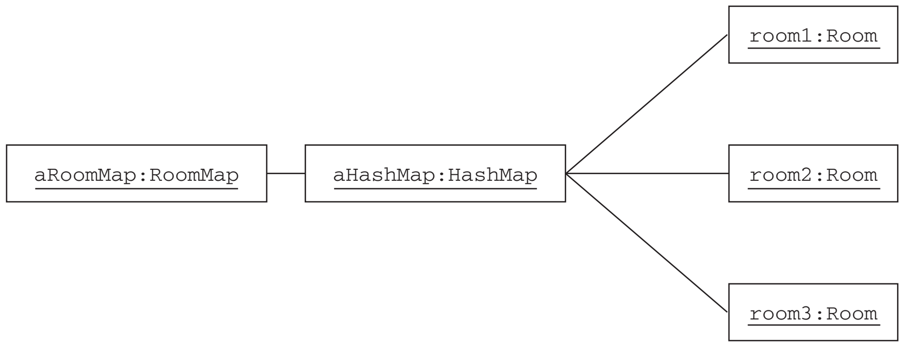


FIGURE 8.43
Domain and technical containers.

8.11.2 Construction Part: Loading Materials

If we do not want to cache all materials in the main memory, then we have to answer the question about what component should be responsible for loading a material on demand. This is the case especially when we want to encapsulate and provide persistent mass data in an object-oriented system. So far, we have implemented the following two strategies within our T&M approach.

Loading materials

The easiest way is to let the using tools load materials into containers. This solution shows at a container's interface that this container holds an identifier for each material element instead of holding all materials. In this state, a container resembles a table of contents. With a container in this state, a tool cannot use the full interface. Though it can display material names and delete or add materials, there is no way yet to edit a material. If a tool wants to edit a material, then the tool will first get that material's identifier from the container and load the material itself by use of a persistence service (see Chapter 11.2). Subsequently, the tool replaces the material's identifier by the material itself. One major drawback of this construction approach is that the tool construction is more expensive, and there may be consistency problems.

Loading containers

Another solution lets containers load materials. Using the domain values (see Sections 2.6.5 and 8.10) identified in the table of contents, a container can use a persistence service (see Chapter 11.2) and request the material. If we use this construction approach, then the container must know the interface of the persistence service it wants to use. In addition, we have to define the depth in which the material is to load. Materials held in a container can consist of complex submaterial structures, requiring a relatively large amount of storage and long loading times. Notice that the entire material has to be loaded, since it is normally not known during the loading process what parts of a material a user wants to edit.

Yet another solution uses proxy objects, where the loading process is hidden from the containers and the tables of contents. Each pointer to a material object in a container references a proxy object. This proxy object can be thought of as a placeholder, and it behaves like a smart reference. When this proxy object is called, then it loads the material object into the memory. This material object can manage more references to other proxy objects, so that materials are loaded only on demand, even deeper material levels. In this solution, the proxy objects either have to know the interface of the persistence service they want to use, or we implement a kind of generic loader to request materials.

Trade-Offs

Loading materials from containers

The following points are important when loading materials from containers:

- The container or proxy object must know the interface of the persistence service they want to use.
- The container or proxy object must know the location from which to load. This can turn into an expensive task if we cannot assume that all materials are maintained in the same persistent location. This problem becomes even more serious when materials can be maintained in different persistent services or media over the system's runtime. In this case, the knowledge about a specific persistence service cannot be coded into the container or proxy object. This information has to be provided when a container loads or a proxy object is created.
- In cases where materials are not loaded completely, we have to deal with concurrent access by several users in a special way. For example, if an object is to be loaded from different sources, then we have to identify users who may modify the object. This problem is directly related to the question how exceptions should be handled. Since loading is now taking place in a material, there is no easy way to send feedback to users, such as when a load request fails.
- In cases where containers can load their elements themselves, we obviously have to anchor some technological knowledge in the containers. This means that containers lose an important material property, that is, their independence of underlying technologies. These issues will be further discussed in connection with different types of database systems (see Chapter 11).
- Depending on the persistence service used, a container or proxy object may have to handle transactions explicitly, some operations may require the transaction context as a parameter.

Rationale

Whenever you have to store complex or numerous materials in a persistence medium, you should consider this construction approach.

8.11.3 Construction Part: Tables of Contents for Containers

When working with large collections in an application domain, we often find that we use tables of contents to access these collections. From the domain perspective, we could argue that tables of contents are independent components in the construction phase. From the technical view, we can justify the separation of a table of contents from its collection.

Most containers hold large quantities of complex materials. For efficiency reasons, we often have to avoid that all materials managed in a container are loaded in the main memory. For the user to understand this point, we normally use the concept of an independent table of contents. A table of contents provides users with an overview of the materials (container elements) available in a container. The users see the table of contents as an object, avoiding the wrong impression that they can access elements in a container directly. In addition, the users understand that there is some delay involved in retrieving elements from a container.

A table of contents gives users structured access to the contents of a container. Each container can create a table of contents, including a defined set of information about the container's elements. To list container elements, we need only a small part of the information about a container element. We normally display the name and one additional attribute, such as a domain value (see Sections 2.6.5 and 8.10). When a user selects a container element for editing purposes from the table of contents, then the complete element is loaded or instantiated at this time.

EXAMPLE

The EMS example

In our EMS example, if we think of the room plan as an example for a domain container, then its table of contents would include the room names. The room plan would return an entire room only when requested to do so.

RATIONALE

As soon as you have numerous materials in one or more domain containers, the question of a table of contents arises and you should consider this approach.

8.11.4 Construction Part: Implementing Tables of Contents as Materials

If we implement a container's table of contents as an independent material, we can treat it like any other material, for example, representing it by an icon on the desktop. The user can see that the table of contents exists independently of the container. The usage model shows clearly that the table of contents does not necessarily show the current state of a container. In summary, the table of contents shows the state of a container that prevailed when the table of contents was created, and the user can explicitly update it.

RATIONALE

If you want to represent a table of contents as a material of its own, for example, represented as an icon on the desktop, you should use this approach.

8.11.5 Construction Part: Implementing Tables of Contents as Domain Values

If we implement tables of contents as domain values, it will be clear both from the domain and the technical views that a table of contents reflects the state of a container that prevailed when it output its table of contents. In this case, being a domain value, the table of contents cannot automatically update itself to changes in its container. Such tables of contents can be handled more easily in tools. In contrast to materials, domain values can be easily treated by GUI elements, as we can easily create a separate GUI element for a list of tables of contents. This list would show all tables of contents and directly supply identifiers for selected entries.

For tables of contents that become very big (e.g., tables of contents for persistent containers), we could implement them as domain value streams (see Section 8.10.4). The example of persistent containers shows clearly that the table of contents can fetch as many elements from its container as are currently requested by the user. However, implementing tables of contents as domain values means that we cannot arrange them on the desktop as we can any other material. If we need this functionality, we can embed a domain value, such as "table of contents," in a material by the same name.

RATIONALE

If you always handle a table of contents by means of a tool, then this approach is recommended.

8.11.6 Construction Part: Coping with Changes to Containers

Users can display the table of contents of a container and concurrently operate a tool on a container element. A user could change a material so that such a change could influence the table of contents.

A simple solution to this problem can be derived from the metaphors for materials and containers: To be able to change a material, this material has to be removed from its container. The container's table of contents marks this material accordingly to reflect that it has been removed. Removing materials means that the container is being changed by the use of a tool. This tool informs all other tools operating on this container over the work environment (see Section 8.15). All tools that receive such an announcement can decide whether or not they have to update their container representation. If a tool changes a material removed from a container, then all tools interested in this particular material will be informed. Each change will become effective, and an announcement to this effect will be sent once the tool has placed the material back into the container.

EXAMPLE

The EMS example

Returning to our EMS example, let's assume that a clerk is fetched from the registry. The name of this employee has changed due to marriage. When the user finishes editing, this employee is returned to the registry, and the registry updates the table of contents. When the editing user removes the employee from the registry, then all tools (e.g., other finders) currently using the registry are informed. When the employee's data is corrected, the employee editor receives an announcement, if it currently displays this employee, and if it registered for changes to this material. Once the employee is returned to the registry, all tools operating on the same employee will be informed and can then display the updated table of contents.

TRADE-OFFS

Implementing this reactive behavior can be expensive, especially if there are process borders between containers and tables of contents. In this case, tables of contents should have a way to handle exceptional situations, such as network failures.

BACKGROUND: MATERIALS AND REACTION MECHANISMS

In the constructions based on the T&M approach, reaction or feedback mechanisms are used only between active components, that is, between tools, tool components, automatons, and the work environment. In addition, access to persistence and distribution services is limited to these components. Materials are treated as components that, in turn, are limited to their domain functionality, without interactive or reactive characteristics.

Materials usually are not reactive

The reason for this construction principle is found in the metaphor for materials. Materials are located in a place, where they can be accessed. Materials do not address their contexts or the technology used by their own initiative when they have been changed. Moreover, it is also meaningful for the technical implementation to keep materials clear of any knowledge about work contexts. A material should not know whether or not its change is relevant for other components. If it knew, then the material would have to decide whether or not a change to it is relevant for tools that directly operate on it or for other tools that operate on the underlying containers.

When generalizing a reaction mechanism, all materials in a use relationship to a changed material will eventually be informed. This would lead to an unmanageable cascade of announcements that make the entire construction too complex. We generalized a mechanism that is relevant for interactive applications only. Even when we merely divide an application into client and server systems, the pure reaction mechnism is no longer useful. For this reason, we generally discard this solution for materials held in containers.

Discussion: Complex Materials and Changes to Containers

The situation gets more difficult if we have complex composite materials with parts containing domain crossreferences, but where the elements still can be manipulated individually by tools. In such a case, the container is actually the only simple means to ensure domain consistency of the composite material. On the other hand, changes to a material component should have an immediate effect on other parts, even though these parts are currently not used by a tool. In such a case, we cannot go the simple way of using a reaction mechanism. Still, we should bear in mind that we have already expanded the container concept. A container manages the consistency of materials that are modified either in this container or outside. But rather than including the table of contents in our announcements, we let the individual material components coordinate themselves via a container that will then update the table of contents.

When containers are expanded to a coordinating context for complex materials, they often take the character of an automaton. In such a case, we check whether or not the desired functionality can be provided by a "real" automaton (see Section 8.13) or a service (see Section 8.14).

A combination of container, service, and automaton has proven useful, particularly for jointly used materials. More specifically, a container resides in a separate service and can never migrate to user workplaces. These containers are always accessed over an appropriate service, where the service may reside on a server "in front of" the container. Especially in workplace systems, we often find either technical proxies or independent automatons that access a service. Since the service is the only thing that accesses a container, it can easily ensure that container's consistency. In addition, the service can send messages as soon as a container has changed. This means that tools at other workplaces can respond elegantly to changes effected to a jointly used container.

Example

The object chart represented in Figure 8.44 shows how a `Finder` tool edits a registry. The `Finder` starts a `SnifferAutomaton` to find room plans in the registry that match a specific search term in their names. The `SnifferAutomaton` requests matching room plans from the `RegistrarService` and receives a table of contents with all matching room plans. Note that the `RegistrarService` obtained these room plans from the `RegistryContainer`. The `Finder` displays the table of contents.

The EMS example

The user can select a room plan from the displayed list and place it on his or her desktop. This action causes the `RegistrarProxy` to be called, which forwards the request to the `RegistrarService`. The `RegistrarService` requests the room plan from the `RegistryContainer`, which marks it internally as logically removed and returns it (as a technical copy) to the finder via the `RegistrarProxy`. Finally, the `Finder` places the room plan on the desktop.

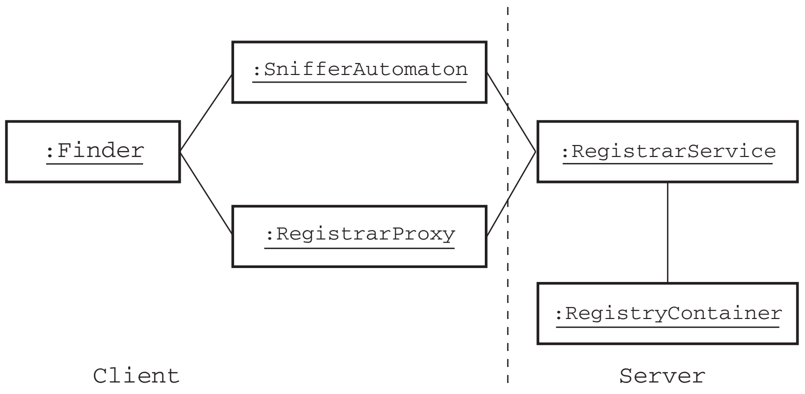


FIGURE 8.44
Object chart of an automaton for domain.

When the room plan is (logically) removed from the `RegistryContainer`, the `RegistrarService` sends a message to all tools registered for that type of message. These tools can now update their tables of contents.

RATIONALE

When you have complex or compound materials, which are changed by many users, then you should be aware of the potential pitfalls and solutions described in this approach.

8.12 THE FORM SYSTEM PATTERN

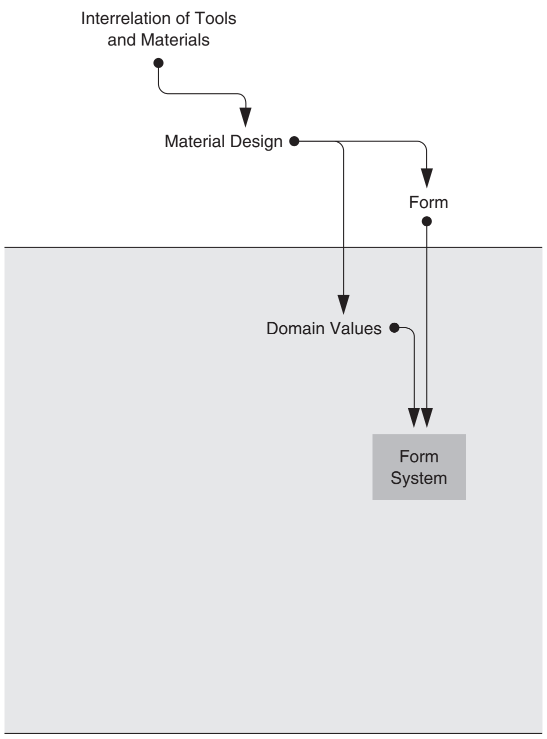


FIGURE 8.45
The *Form System* pattern.

Intent

When you have identified the frequent use of forms in an application domain, it is useful to transfer this concept into a system of software forms with appropriate tools.

Problem

Converting the conceptual pattern of forms into a construction appears to be easy at first sight. However, we often find that a large number of forms has to be implemented in a similar way. Implementing generic operations for forms is relatively expensive and does not offer tangible benefits, compared to simple "data containers." In addition, building tools to operate on such generic forms is normally just as expensive. Therefore,

How can we support the use of forms inexpensively to become part of an application?

Relate To

The conceptual pattern *forms* outlines the background of what forms are and how they relate to materials. The design pattern *domain values* provides the essential elements for handling domain-motivated data in the form system (see Figure 8.45).

Solution

We build a form system, consisting of classes for forms, form templates, form elements, element lists, and form fields, where we base the form fields on existing domain value classes. We add generic form tools for defining and editing form templates and instances of forms.

Forms are composed of form elements. The simplest structure of a form element is the *form* field, which contains a single domain value. These parts are sufficient to implement a simple form: the appropriate form fields are selected from a predefined set and parameterized by fitting domain value types.

If we build a form from individual form fields, then complex forms will soon become unclear and hard to handle. To avoid this problem, we structure form elements based on the *composite* pattern of Gamma et al. To utilize the full potential of the *composite* pattern, we treat the form itself as a subclass of a form element within the inheritance hierarchy, so that a form can, in turn, have subforms as its elements. It also allows us to reuse forms when building other forms. Figure 8.46 shows such a forms hierarchy.

If we want to use a specific form, we have to have already define a the elements that this form should contain. For normal use, we want to prevent changes to the structure of a concrete form. For this reason, we specify operations that change the structure at the interface of the class `Form`. But then, how should we initially define a form?

We can solve this problem by separating the usage of a specific form from its form definition. Consequently, we describe the form's structure in a form template,

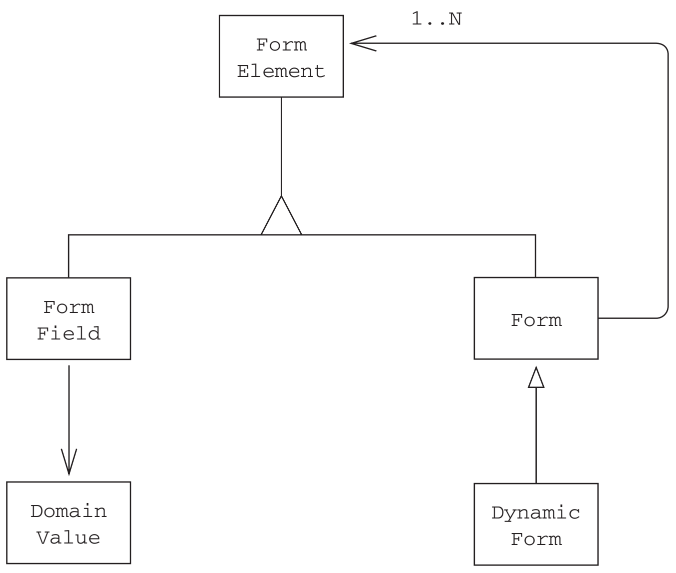


FIGURE 8.46
A forms hierarchy.

which will then be used to create similar types of forms. Let's look at the interface of the form class:

```
class Form
{
  public int elementCount ();
  public FormElement elementAt (int index);
  public boolean hasElement (String name);
  public FormElement elementByName (String name)
  public FormElement[] elementsByType (Class type)
  public boolean hasField (String name);
  public FormField fieldByName (String name);
  public boolean isValid ();
  public String[] validationHints ();
  public void setGeneralNote (String note);
  public String generalNote ();
}
```

Once we have defined a form, we cannot change its structure. A form is defined by a template called `DynamicForm`. A form can be defined only once. Changing the structure of an instantiated form would conflict with the requirement we just saw, namely that we want to prevent arbitrary changes to the structure of a form.

EXAMPLE

The EMS example

Let's see these ideas in the context of our EMS example. When developing the device manager, we initially represent employees as employee forms. The corresponding class for an employee is inherited from the `DynamicForm` class and defines its structure once in the constructor, that is, by composing an appropriate form template.

```
public class Employee extends DynamicForm
{
  public Employee()
  {
    addElement( new FormField(StringDV.Factory.instance(), "ini-
                tials", "Initials"));
    addElement( new FormField(StringDV.Factory.instance(), "name",
                "Name"));
    addElement( new FormField(StringDV.Factory.instance(), "first-
                name", "First Name"));
    addElement( new FormField(StringDV.Factory.instance(),
                "street", "street"));
    addElement( new FormField(StringDV.Factory.instance(), "zip",
                "ZIP"));
    addElement( new FormField(StringDV.Factory.instance(), "email",
                "E-Mail"));
    addElement( new FormField(StringDV.Factory.instance(), "tele-
                phone", "Telephone"));
    addElement( new FormField(EmployeePositionDV.Factory.
                instance(), "position", "Position"));
  };
}
```

That's all our sample class contains. The single fields in an employee form can generically be accessed from operations of the superclass. The forms editor uses the `FormEditable` aspect to work on the form superclass. Figure 8.47 shows the relation between generic components and the employee form used in our example. Note that the gray classes are generic parts, such as from a framework.

TRADE-OFFS

Using forms can mean a significant reduction of our development cost. We can quickly implement a basic functionality and save tedious implementations of recurring functionalities.

Unfortunately, using the forms concept can also entail some problems. The forms concept may entice developers to model materials as forms, when these materials should have various domain operations. Inexperienced developers especially will often tend to design data collections instead of materials.

Another risk is seen in a reduction of the type safety in typed programming languages due to generic forms. In many cases, this can mean that the compiler can no

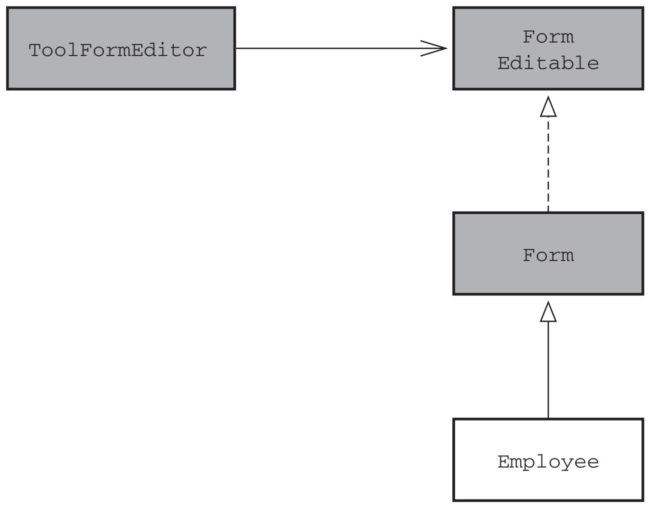


FIGURE. 8.47

An employee form used in the EMS example.

longer check whether assignments of form objects to identifiers are meaningful. In particular, there is no static way to ensure that fields accessed in a form actually exist under the specified names, and that they have the expected (domain value) type. This problem is particularly serious when forms are changed.

Changing field names and types represents a particular risk, because it may mean that the compiler cannot detect inconsistencies caused by such changes. We can reduce this problem if we make field contents accessible only via access operations. Then only these operations will be used whenever a known field needs to be accessed. Generic queries for field values will then be left exclusively to generic tools.

Another problem arises in connection with the persistence of forms. If a user loads an older version of a form, then this form may have some fields missing that were added in later versions. The only way to prevent this problem is by explicit programming. Once a form has been loaded, an extended form has to generate the new fields, that are missing in the older version and fill them with default values.

Discussion: Developing Forms

In the course of the development process, we may find that a material once identified as a form has actually more domain interactions than initially expected. In this case, we need to develop it from a generic form into a more specialized material.

Our general development strategy supports "migration." This means that we can implement additional interactions in the form class, without needing to replace the generic implementation of the superclass. For example, if we want to add the total price to a purchase order form, we could implement an appropriate operation in the purchase order form class. The purchase order form class could then request information such as number of units and cost per unit from the generic part of the form, then do a calculation, and finally return a result. From the domain view, the purchase order form no longer uses the generic form fields, for example, to fill them in or read them; it now also has a specialized interaction.

When developing a "normal" material from a form, this does not necessarily have to replace the existing form. The "normal" material can output its data like a form and update itself from a completed form.

Layout of forms

Basically, materials know nothing about their representation on the screen or on a printed hard copy. For this reason, we have to store layout information in another place. This layout information may then be read by a tool or automaton and used to process forms. This allows us to strictly separate the form from its screen representation.

For example, we could specify layout information in configuration files, independently of our program code. This allows us to define a layout for forms independently of the forms. Even programming novices could then create and adapt the layout of forms.

Consistency conditions

We saw how consistency can be checked independently of a context in connection with domain values (see Section 8.10). We may now use this consistency check for forms and ensure that each field of a form contains a valid value. Intra-form consistency checks are initially left to each single form. A form then has to provide probing operations indicating that it was properly filled in or whether there are any problems.

We also saw that it is important to allow inconsistent forms to be saved. This is a fundamental requirement for expert work with forms. Users can put an incomplete or inconsistent form aside and continue editing it later. However, some special tools or automatons may refuse further processing of a form if they find that consistency conditions are violated.

SOLUTION: TOOLS FOR FORMS

A framework can provide the generic tools we need to create generic forms. A generic forms editor can easily be used to fill in and represent fields in a form. Such a tool actually needs only to support the generic form interface. It does not have to know specific forms, such as a purchase order form.

We often find in practice that a generic tool is not sufficient, that is, it has to be adapted to a specific task. If such requirements concern only the user-interface of a tool, then it appears useful to split the tool into an FP and an IP, as discussed under the *separating FP and IP* pattern (see Section 8.7). All we have to do then is to specialize the IP, including the GUI resources it uses, by subclassing. This means that we use the basic functionality of the generic IP, while specializing the layout or other features.

However, we must frequently expand the functionality of a tool and adapt it to the set of a user's task. This is normally necessary when a generic form develops gradually into a specialized material. In that case, we have to adapt a generic tool to the specialized form to be able to utilize the new interactions with the material. Note, however, that this is not a general rule. Once a form has been expanded by a more specialized interaction, this may be of interest for only a single tool. At this point, remember our discussion of the *aspects* design pattern (see Section 8.3), which represents relations between tools and materials. We can add a new aspect to a generic form to make it suitable for manipulation by a specialized tool.

The important thing to remember is that we should continue taking advantage of the generic parts of a form and the relating tools. When a form grows to become a special material, then this doesn't mean that we have to give up our generic use of this form. This also applies to tools, where we could use the generic FP of a forms editor to easily fill in fields in a form and send the appropriate events to the IP.

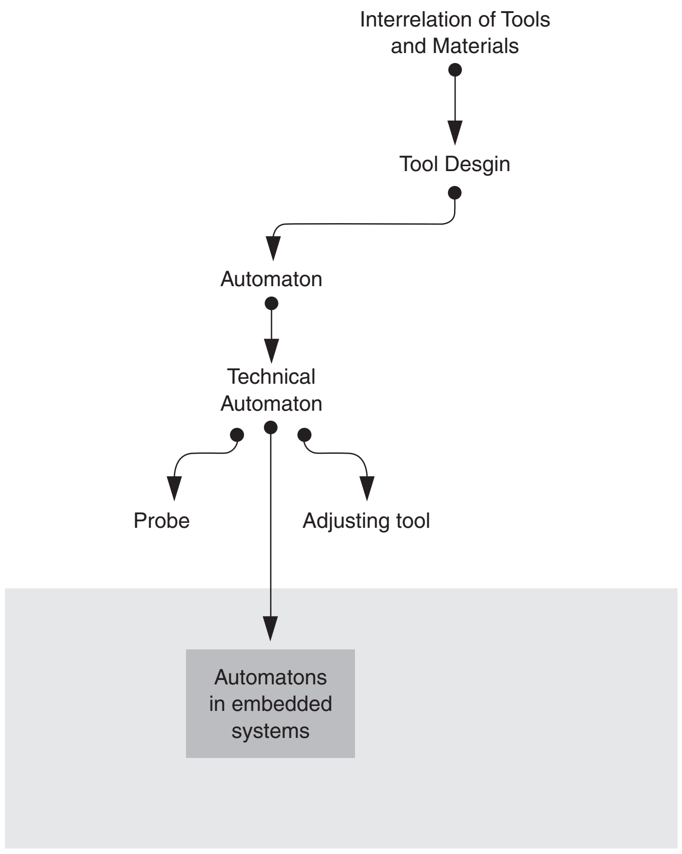
Rationale

Consider building a form system when you have to support an office-like work context with many different paper forms as the relevant objects of work.

8.13 THE AUTOMATONS IN TECHNICALLY EMBEDDED SYSTEMS PATTERN

FIGURE 8.48

Automatons in technically embedded systems.


Interrelation of Tools and Materials
Tool Desgin
Automaton
Technical Automaton
Probe
Adjusting tool
Automatons in embedded systems

Intent

Technically embedded systems can be interpreted as automatons sending autonomous events. These automatons can be probed and adjusted. This pattern shows how to implement these concepts.

Problem

Technically embedded systems often use automatons to model the system's external components. This allows us to embed an autonomous technical device as an additional source of events in our application system.

We can call altering operations on a technical automaton to set the technical device. These settings change the automaton's state. In addition, there are external events that cause a change of the automaton's state. This means that, in addition to an event channel for the window system, we also have to consider an additional event source in interactive applications. Therefore,

How can we implement an embedded application system based on the conceptual pattern for a technical automaton?

Relate To

The conceptual patterns *technical automaton*, *adjusting tool*, and *probe* introduce the concepts that this design pattern explains on a construction level (see Figure 8.48).

Solution: Adjusting Tools

We connect adjusting tools and technical automatons over asynchronous communication within a multiprocess space. Then we connect probes to the automatons.

The conceptual pattern *technical automaton* has introduced a way of interpreting technical systems within the T&M approach. Technical automatons model the real-world devices and machines. Within the software model, we use special software tools, so-called adjusting tools, to let users operate and set these automatons. Such adjusting tools are normally tailored to their automatons. To achieve a flexible way to model the communication between technical automatons and their adjusting tools, we often use probes to represent parts of an automaton's state. In addition, probes are helpful in modeling the control flow, because they are connected to the tool over an observer mechanism.

Probes are normally connected to a technical automaton and fed with measurement readings by the automaton. Clients can register for events with the probe. They will then be informed about readings in the intervals they chose.

The FP of a tool or the IP of an automaton (if you decide to add a user interface to an automaton) can request a probe from the automaton and register with the automaton to probe it for domain values. Together with the registration, an FP or IP can specify a preferred method to obtain information about changes (e.g., by sampling rate or event). Figure 8.49 shows an IP that uses an automaton and one of two available probes.

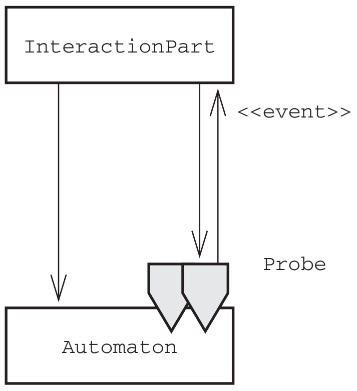


FIGURE 8.49

An IP using an automaton and one of two probes.

Like other tools, adjusting tools are embedded in an environment. The adjusting tool itself will usually consist of an IP and an FP, and it uses an automaton. The IP represents the automaton state and supports requests for changes. The IP does not have its own memory, so it must rely on an FP to obtain the current domain state. Figure 8.50 shows the basic architecture of an adjusting tool.

The FP informs the IP about the domain state and coordinates the connection to the automaton. Though each automaton has its own state, we always need an FP to supply the necessary domain interpretation for the values of an automaton. If a tool is used to control several automatons, then the FP assumes the domain coordination and combination of technical event sources.

An FP buffers the probing results from an automaton, and it can interpret different values and calculate new values for the IP. Moreover, the FP can combine domain states and the settings of several automatons in one single tool.

Solution: Adjusting Tools and Automatons in a Multiprocessing Space

We want to implement adjusting tools and automatons so that the automatons operate as independent components on separate computers. To this end, we have to connect processes. We call this environment a multiprocess space. Let's look at the terminology before we continue our discussion.

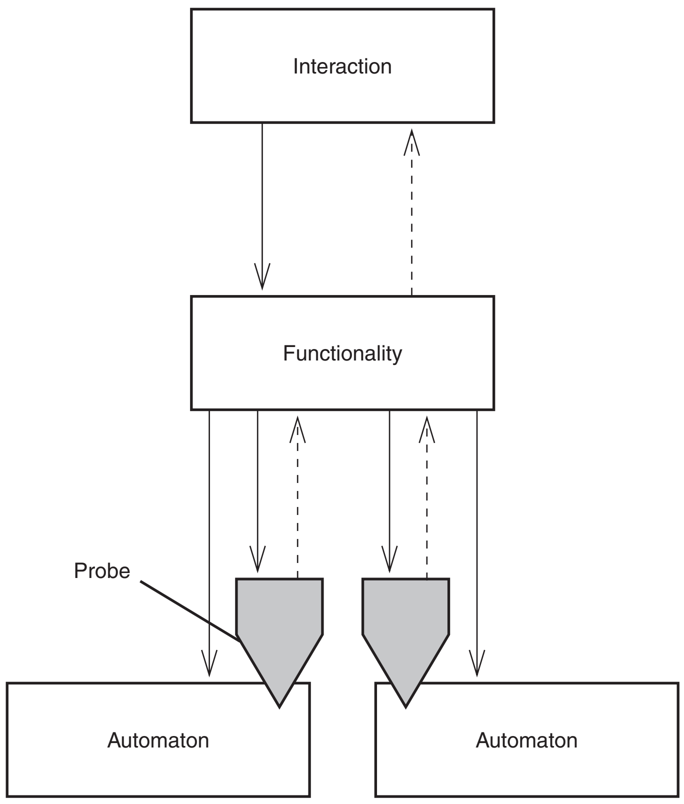


FIGURE 8.50
The basic architecture of an adjusting tool.

A *multiprocess space* is the domain and technical space spanned by an application system when that system's tools and automatons run in different processes connected over a communication medium. We assume that the distribution of components over this space is explicit, both in the design model and in the implementation model.

The separation of components in a multiprocess space requires a connection concept beyond the approaches described so far. In this respect, we speak of asynchronous coupling.

***Asynchronous coupling* means the connection of two objects over an asynchronous communication medium in a multiprocess space. We distinguish three variants:**

- **operation call without expecting a direct result;**
- **operation call and expecting a (direct) result; and**
- **abstract communication via events only.**

An asynchronous communication medium always has to expect connection errors to occur. For this reason, we need connection information on the metalevel.

EXAMPLE

Sequence diagram for interprocess communication

We use the three asynchronous coupling variants to distribute adjusting tools and technical automatons. The sequence diagram in Figure 8.51 shows as an example of how tools and automatons can interact in different processes. Notice that we added two elements to this chart. First, the objects of a process are grouped at the top of the chart, so that several communicating processes can be represented in the same chart. Second, we introduced a broker object to better explain the interprocess communication.

Let's look at the processes and events in Figure 8.51:

- *Registering*: The FP uses the proxy to register with the remote automaton. Note that there is a time delay in this asynchronous process. When the automaton receives the registration request, it generates an observer proxy. If the FP cannot be registered for technical reasons (e.g., because the network connection broke down), then the client process is informed about the failure. This error message is sent to the proxy object after a timeout specified by the system. The proxy converts the error into a communication status event. If the FP can successfully register for the corresponding event, it can now respond.
- *Adjusting*: To adjust the automaton, the FP calls the appropriate operation on the proxy. The proxy creates a data packet and sends it over the communication system; then the control flow is returned to the FP.

  The data packet is sent to the automaton's `Entry` object with a time delay, where it is converted into an operation call on the automaton. The automaton sets the requested values, and the control flow is returned to the communication system via the `Entry` object. When several clients communicate with the automaton in this way, then all adjusting requests are serialized by the communication system. Each successful change leads to an announcement about an effected state change. The consecutive execution of all requested changes determines the final state that is announced to the clients. This ensures the fundamental construction principle that all interested tools have to be informed about changes to materials (or the automaton, in this example).

- *Announcing*: The automaton changes its state upon request, and because it represents an independent technical device, this situation occurs frequently. A probe is informed about each state change. The probe informs the observer proxy about each state change. Once the observer proxy has been fed with the current state information, it converts this information into parameters and passes them to the communication system. Subsequently, the control is returned to the probe.

  After a time delay, the communication system informs the automaton proxy that it has received data. The automaton proxy encapsulates the conversion of data packets arriving from the communication system, interprets the contents, and changes its internal state. Subsequently, the automaton proxy passes this state change to the responsible probes, which inform all observing local objects, such as a tool's FP. The FP of a tool can probe both the probe itself and the automaton, because the current model of the automaton state is present in both objects. This entire process of announcement and probing can be thought of as a synchronous communication at the tool process end, which guarantees all required consistency criteria. Subsequently, the control over the probe and the automaton proxy is returned to the communication system. Finally, the control is returned to the tool's user via the window system.

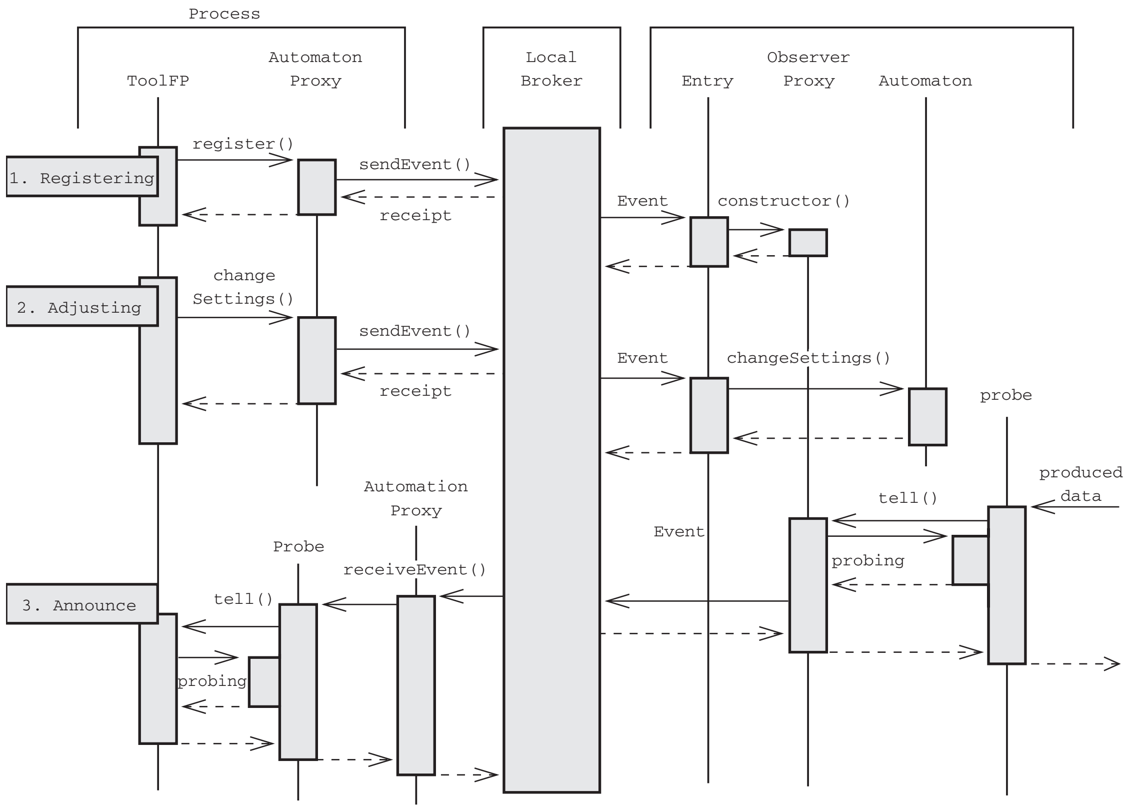


FIGURE 8.51 Asynchronous communication between a tool and an automaton.

The multiprocess architecture just described offers a nice way to handle reactive user inputs in one process and forward new automaton states concurrently. This architecture has been implemented in several projects, based on a special integration of events in the event queue of the window system.

To better understand this approach let's look at the flow of control among the three processes of the above example. Figure 8.52 shows how these three processes interact. The circled areas in this figure denote each of the three processes.

TRADE-OFFS

Usage model for distributed communication

The usage model for faultless and quick asynchronous communication in multiprocess spaces does not have to differ substantially from the synchronous model that we would use in a single-process system. Communication errors will be handled on the basis of known principles. Users are normally informed via a modal dialog when a request is unsuccessful, for example, because a remote component did not respond.

It has proven useful to symbolize the network state for remote components that are used frequently in a system. To this end, we could use entries in a status line or small icons that change their appearance, depending on the status.

There are similar options to represent time delays of communication processes. We could also use status line entries or icons to display the progress of communication

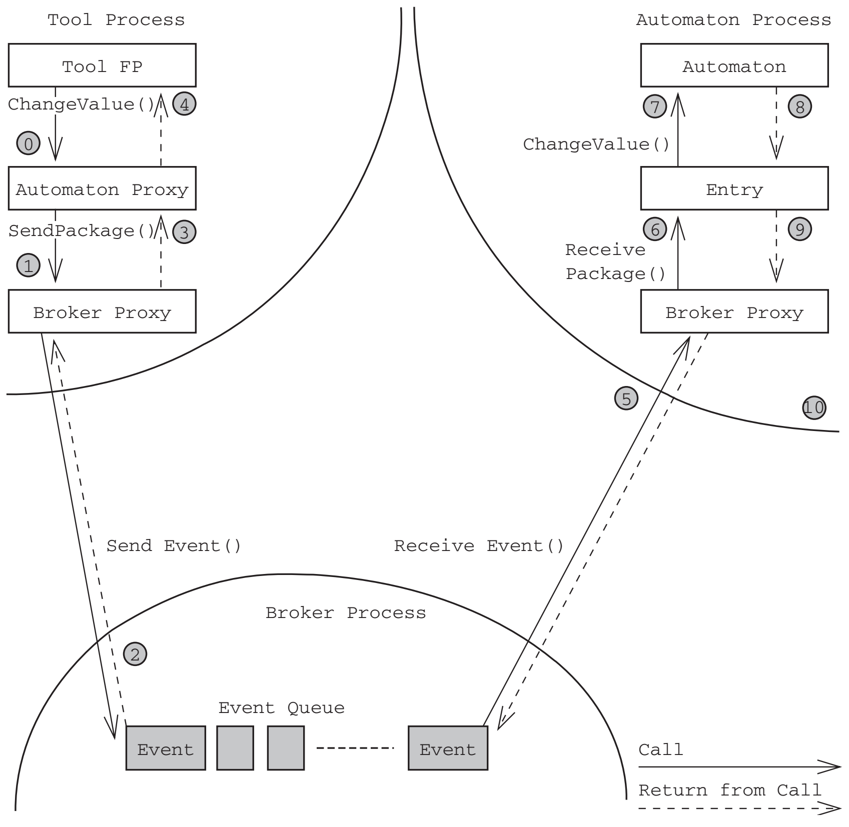


FIGURE 8.52 Flow of control and interaction between the three processes of Figure 8.51.

processes for the user. On the other hand, we found in many projects that the use of a "sleeping mouse pointer" (e.g., represented by an hourglass) is less successful. The reason is that most users know this symbol from synchronous interprocess communication, expecting that they cannot continue working in this phase.

RATIONALE

Consider this pattern when you have to integrate autonomous technical components as part of an application system.

8.14 THE DOMAIN SERVICES PATTERN

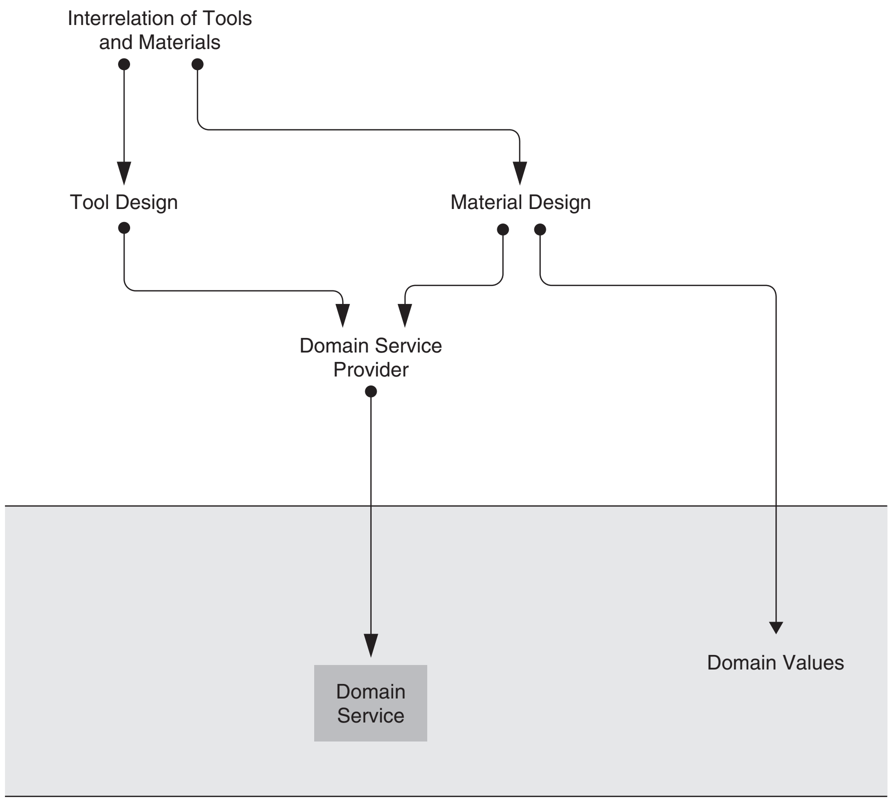


FIGURE 8.53
The *Domain services* pattern.

INTENT

Domain services offer the means to separate domain functionality, both from different front-ends and back-ends. They abstract from the various interaction models of frontend technologies.

PROBLEM

When looking at the equipment of different workplace types, we often find that many common domain features are implemented in several tools. If we subtract the interaction parts of these tools with their different technologies, we can identify a domain functionality that is independent of different representations.

In this context, independent representation means for us independence both of the presentation and the interaction. For example, in Internet applications based on HTML, clients not only look different than clients in a fully fledged desktop application, because an HTML browser uses different GUI elements, they also behave differently. Although it is absolutely common for a desktop application to change a series of field contents immediately upon some user action without any noticeable delay, we normally cannot and will not implement this behavior for HTML clients.

This example leads to another issue. Many servers (HTML and other) implement domain functionality by directly accessing a database. This is another issue of independence that we could call back-end independence. In short,

We are looking for a concept to implement the same domain functionality, but use different interaction options, and design it for different workplace types and different technical front-ends (desktop, webtop, etc.). The solution we search for would also be useful for many issues relating to the distributed client-server architecture.

Relate To

The conceptual pattern *domain service provider* sketches the concept realized here. Domain values are useful for implementing the interface of a domain service (see Figure 8.53).

Solution

We combine functionality and knowledge about a service or about the use of a product, regardless of the specific user interface. We make this domain service available in the form of a bundled service package. This means that we encapsulate the materials it manages in this domain service.

A domain service is built internally so that it can handle materials. Domain services can also delegate a requested service to other services or automatons. References to materials within a domain service are never made public. This means that the interface either offers values, or it provides a copy of a material (the difference between a technical and domain copy of a material is explained in Section 10.1.1).

Requirement for domain services

A domain service is implemented so that it meets the following requirements:

- It provides services at its value-based interface.
- It accepts domain requests at its interface, that is, clients delegate standardized subtasks to the service.
- It encapsulates the materials it manages and the specific implementation of the domain interactions.
- It can be implemented to act as an equipment component and have different plug-in components to that end.
- It does not make any assumptions about the interaction model and the representation at the user interface.
- It basically supports multiuser and distributed systems.
- It should offer an appropriate state and session model.

Technically, a domain service can be implemented in a totally different way, but this shouldn't change its semantics. For example, if we implement a domain service as

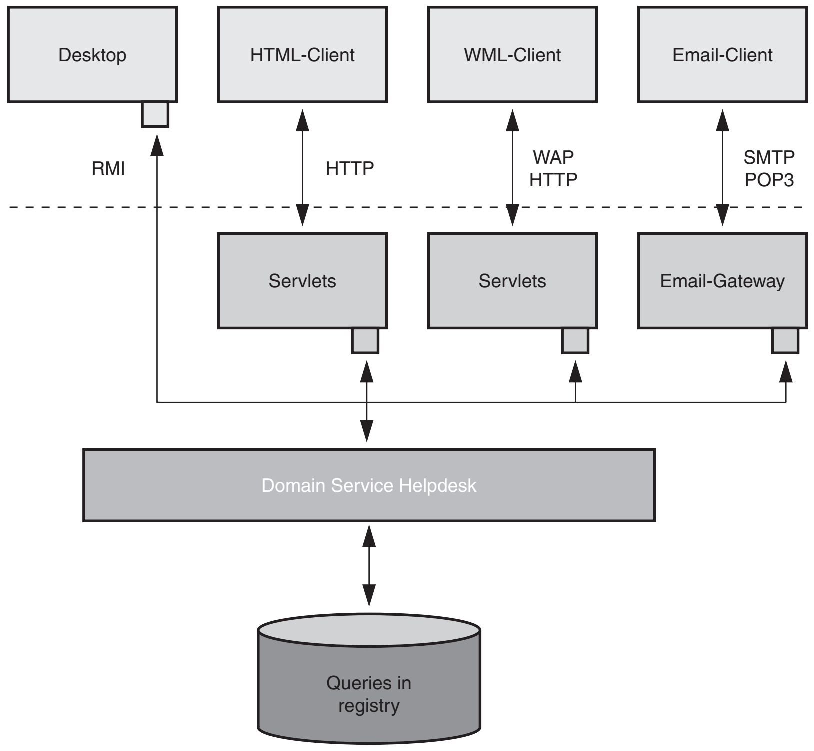


FIGURE 8.54
Domain Service Helpdesk.

an Enterprise Java Bean (SessionBean), we have to add a home interface for technical reasons. However, this does not change the domain use of that service.

EXAMPLE

To better understand the domain service concept, we use the example of a helpdesk. Users can send their questions to providers and obtain answers from experts over this helpdesk. Figure 8.54 shows the domain service `Helpdesk`, which can be used by desktop tools and servlets or over a mail gateway. In this example, the requests sent by users are the materials of this application. The domain service manages these requests. To this end, it uses a registry. The dashed line separating the frontends from the domain service denotes a logical separation. The important thing for our client-server distribution is where we introduce an intermediate layer that connects clients to the domain service.

Let's look at the details of the example in Figure 8.54. The Web server appears useful for HTML-servlet clients. However, we could also opt for a two-layer solution, including a Web server and a separate server for the domain service.

For desktop tools, the FPs discussed in Section 8.7 appear to be suitable candidates to connect to the domain service. For simple tools, we could even do without separating IP and FP as different classes. The tool connects directly to the domain service. Since our tools should be highly interactive, we accommodate them on the client side. Practical experience has shown that it normally makes no sense to move a tool's FP to a server.

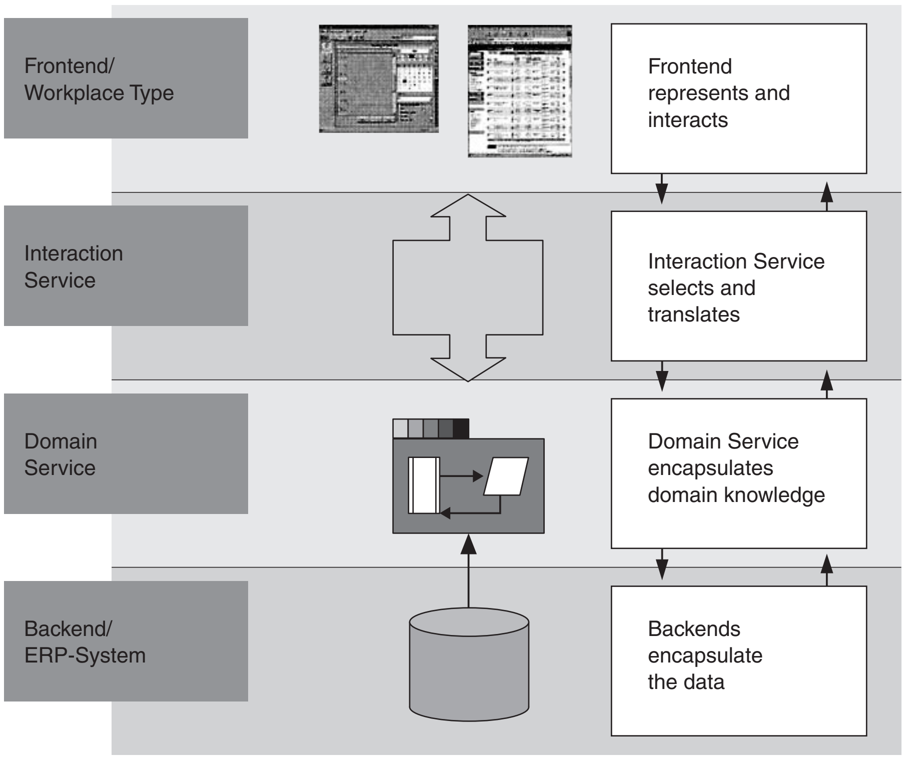


FIGURE 8.55
Logical layers of a service architecture.

Figure 8.55 shows the logical layers and their client-server separation. The domain service is usually located on a separate server and the user interface on the client. Then we need an intermediate layer that transforms the functionality of the service to the different interaction model of the frontends. We call it an interaction service, which can reside either on the client or the server.

TRADE-OFFS

Operations of domain services should be atomic

The technical features of a domain service include the requirement that we treat its operations as being atomic, to the widest possible extent. This means that we should try to design all services to be stateless. Naturally, stateless services will have an impact on the interface. All implicit or explicit parameters required for a method have to be passed with each call. Parameters always have value semantics, that is, they do not include references to parameter objects. In general, you cannot use a domain service to directly access any of the materials it contains. To access a specific material, it is often useful to exchange unique identifiers across process boundaries.

Coping with state changes

Clients of many domain services need to be informed about state changes. For this reason, we generally allow clients to register with the interface of a domain service, so that they can receive announcements. This means that we specify an additional technical dependence, depending on the eventing mechanism used. Note at this point that the interaction model of some frontend types (e.g., HTML browsers) does not support such announcements. In this case, we have to provide expensive auxiliary constructions in the interaction service.

Rationale

Whenever you build a distributed application with different workplace types and technical front-ends, consider encapsulating the domain-related funtionality as domain services.

8.15 THE ENVIRONMENT PATTERN

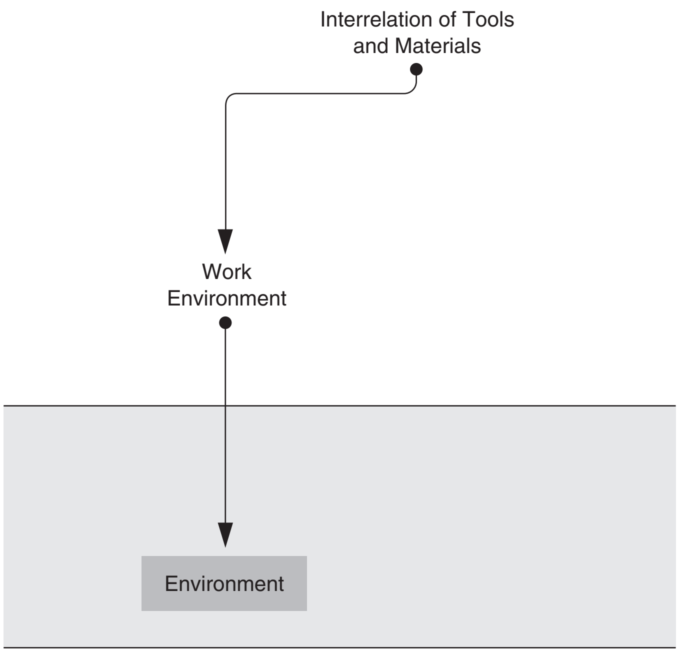


FIGURE 8.56 The *Environment* pattern.

Problem

A work environment is a very important concept in the development of workplace systems for a specific application domain.

We use tools and materials within a limited space, such as on a desktop. In such an environment, we find a number of tools and materials. To manipulate a material by use of a tool, we bring both together and can always identify when we need which tools to work on which materials.

How can we implement an environment so that it is an abstraction of all the limited work spaces? How can an environment help to couple the appropriate tools and materials?

Relate To

The design pattern *environment* realizes the concepts outlined by the conceptual pattern *work environment* (see Figure 8.56).

Background: Environment

An environment is a personalized space

The environment delimits the area that users move in to complete their tasks within an application domain. Since the environment can be different for each user, and each user can have their individual environment, different environments normally have to know their individual users, for example, by identifiers (user IDs). We can use these identities for building communication channels between environments.

Tools and materials in an environment

Also, an environment manages tool and material types. It knows all tools its users can activate, and all the material types, for which its users can create instances. We know that tools operate on materials. An environment can determine which tools can manipulate which materials. In fact, an environment knows aspects, in addition to tool and material types. It can use these aspects to find out which tools and materials can be combined.

In addition to this management function, an environment knows which tools are active at a time, and which materials are currently manipulated by these tools. If several tools manipulate one single material, then they should always show the most recent material state. To this end, all tools have to be coordinated. The environment has the fundamental knowledge to coordinate tools.

While a tool manipulates a material, it is within the environment. It is not removed from the environment and passed to a tool for exclusive use. This corresponds to our notion of an environment, where we can use tools to manipulate materials, allowing several tools to operate on the same material within their environment. Also, it confirms our notion of spatial and logical dimensions discussed in Section 7.6.

If we say that a material must remain in its environment while it is manipulated by one or more tools, then we have to protect it so that the work effected on the material by a tool cannot be compromised. For example, we want to prevent a user from dragging a material to the trash can while another user is editing the same material.

Concepts of an environment

In summary, an environment knows the following domain elements:

- user identification;
- workplace identification;
- existing tool types;
- existing material types;
- active tools; and
- materials currently manipulated by tools.

We have said in respect to tools construction that a tool object is used as a context representing the entire tool. If we split a tool into subtools, then the tool can be an observer of its subtools. This means that it represents the context of its subtools. Moving to the next higher level, we can think of the environment as the context of all the tools it contains. If the environment represents the context of its tools, then it has to implement the domain context of all of its tools.

Solution

We use a class to build a environment. This class includes a so-called workspace, where we can arrange all objects needed in that environment. In addition to objects that an individual may require, the workspace also includes a number of permanently available containers, which are part of the basic equipment of an environment, for example, a template folder, a toolbox, and a trash bin to remove objects no longer used from the environment.

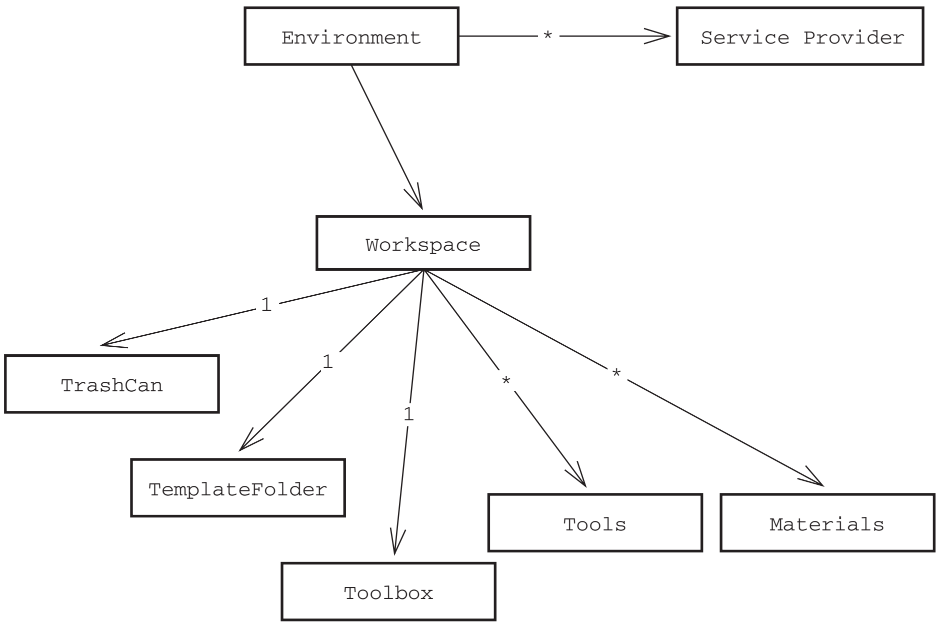


FIGURE 8.57
Environment and workspace.

EXAMPLE

In the example in Figure 8.57 we sketch the standard equipment of an environment.

Features of an environment

The standard containers of an environment shown in Figure 8.57 can be implemented as domain containers. Based on the management functions of domain containers, we can implement a number of useful features.

- The toolbox can be used to identify tools generally available. An appropriate tool can be used to display the toolbox for the user and let users start tools from within the toolbox.
- A template folder could be used to hold available material types and return new instances of a material type upon demand. This expands the options we have from traditional manual template folders, where you normally find a finite set of templates. Once this reserve is exhausted, we have to refill it with new templates. In addition, materials held in template folders can age when the original is modified. We can solve both problems by using a software template folder. This folder is part of the environment, copying the original and returning copies upon request. Consequently, the template reserve will never be exhausted, and users always get the most recent version of the original.
- Material metainformation from within the environment show which tools can be used for which materials.
- As we have said, a tool cannot be started without a material. If no material is started when a tool is activated, then the system should use a new default material. Such a default material can be easily created for a tool from the template folder.
- When a tool modifies a material, then the tool can inform its environment to this effect. The environment can use the workspace to determine other tools that may manipulate the same material, and inform them about the first tool's modification to the material.

TRADE-OFFS

As an alternative to using domain containers, we could implement toolboxes, template folders, and workspaces by technical containers (e.g., arrays or lists). In this case, however, we have to implement the domain functionality in the environment. Since an environment generally assumes a lot of tasks, it is recommended to use domain containers to keep the environment clear and easy to understand.

RATIONALE

Whenever you design and implement workplace systems you should consider using an environment. Even if you don't need a desktop interface, environments are useful for encapsulating and identifying a workplace in a distributed environment.

8.16 USING THE T&M DESIGN PATTERNS FOR THE JWAM FRAMEWORK

In this section we show the interplay among various T&M patterns. We have used these patterns in implementing the JWAM framework, which is a generic framework for constructing interactive applications in line with our approach. This section briefly describes these implementations and how they relate. Our description focuses on the relations and interplay between patterns.

The section begins with a description of the materials construction, followed by a description of the tools construction. Subsequently, we explain how we support domain value construction and domain container construction.

With its component model, the JWAM framework supports different options for user interface connection, including the concept of interaction forms (IAF) and presentation forms (PF). The forms concept supports the construction and use of forms. Similar to IAFs, forms are also components.

In a later section, we explain how domain services are built in JWAM, and finally how we implement a JWAM environment.

8.16.1 Materials Construction

Materials construction is essentially supported by the `Thing` interface (see Figure 8.58) and the `ThingImpl` superclass. The level of `Thing` defines that each material has a name and a unique ID of the type `IdentificatorDV` across the entire system, where `IdentifierDV` is built as a domain value.

```
public interface Thing extends Serializable
{
public void rename (String newName);
public String name ();
public String[] toStrings ();
public void setID (IdentificatorDV id);
public IdentificatorDV id ();
public String typeDescription ();
public ThingDescriptionDV thingDescription ();
}
```

FIGURE 8.58

The Thing interface.

```
public class ThingDescriptionDV extends DomainValueImpl
{
public String name ();
public Class thingClass ();
public String iconName ();
public IdentificatorDV id ();

// Management of named values
public int entryCount ();
public String valueName (int index);
public DomainValue value (int index);
}
```

FIGURE 8.59 The Thing DescriptionDV interface.

In addition, each material can return a description in the form of the `ThingDescriptionDV` class (see Figure 8.59) about itself. This description is also a domain value. It includes information about material names, material IDs, the class a material belongs to, and so on.

Specific materials inherit from `ThingImpl` or fulfill at least the `Thing` interface. In addition, subclasses of `ThingDescriptionDV` can be defined for specific materials to provide additional information. One of the most frequent requirements in real-world applications is that the `ThingDescription` for a special material should carry additional information, but it should not define new operations. This means that no subclass should be derived from `ThingDescriptionDV`. It is normally sufficient to pass such additional information as a list of named values to `ThingDescriptionDV` when it is created. These value lists can then be output in tables.

Note that we derive not only materials but also all "things" from `Thing` or `ThingImpl`, respectively. In addition to materials, this includes tools, domain containers, automatons, and domain services.

8.16.2 Tools Construction

We use the `Tool` interface and the `ToolImpl` superclass for our tools construction. Specific tools inherit from `ToolImpl` or implement the `Tool` interface as a minimum requirement. Since tools are also "things," `Tool` inherits from `Thing` and `ToolImpl` inherits from `ThingImpl`. In addition, `ToolDescriptionDV` is a subclass of `ThingDescriptionDV`, which describes a tool in more detail.

Each tool can provide information about its functionality and its interaction. This information is represented by two interfaces, `Functionality` and `Interaction`.

Depending on whether or not we want to separate function and interaction when building a tool, a tool can be derived from `ToolFpIpImpl` (see Figure 8.60) or `ToolMonoImpl` (see Figure 8.61). `ToolFpIpImpl` represents a tool with its FP and IP separated, and `ToolMonoImpl` is the superclass for monolithic tools with their FP and IP not separated. In this case, the tool object returns itself when it is queried for its functionality or interaction.

At their interfaces, functionalities define events in the form of objects of the `Event` class. Objects that define events at their interface have to implement the

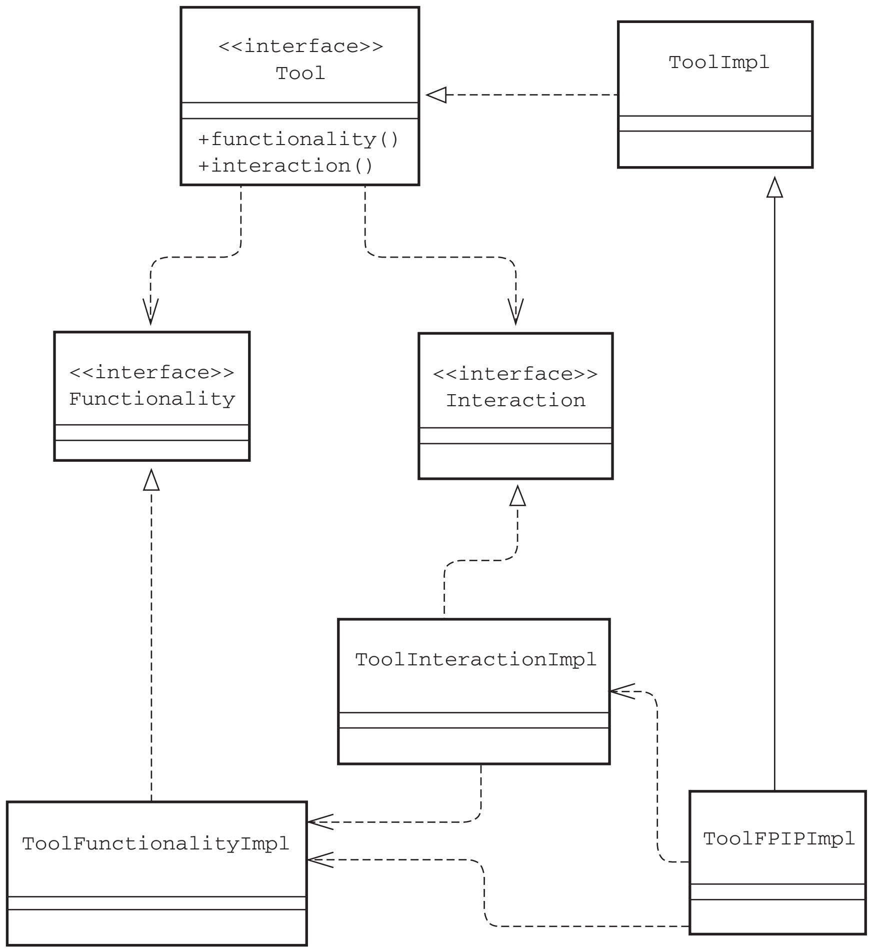


FIGURE 8.60
A tool with separate interaction and functionality.

`EventSubject` interface. Therefore, the functionality interface inherits from `EventSubject`. The observers register directly with the events and concurrently pass objects, the classes of which implement the `EventReaction` interface. Since `EventReaction` has only one defined operation (`update`), we would normally work with anonymous inner classes, which are defined by an observer. In the JWAM framework, observers can be functionalities and interactions. Figure 8.62 shows the classes of the event mechanism.

A chain of responsibility existing between subtools and context tools is implemented by use of the `RequestHandler` interface and the `Request` class. Note that the tool objects are actually request handlers, which means that the `Tool` interface inherits from the `RequestHandler` interface. Each functionality knows its successor under the `RequestHandler` interface and can pass objects of the `Request` class to this request handler for further processing.

JWAM-based tools should not depend on a specific user interface. Therefore, rather than letting `Interaction` directly use the AWT or Swing windows and

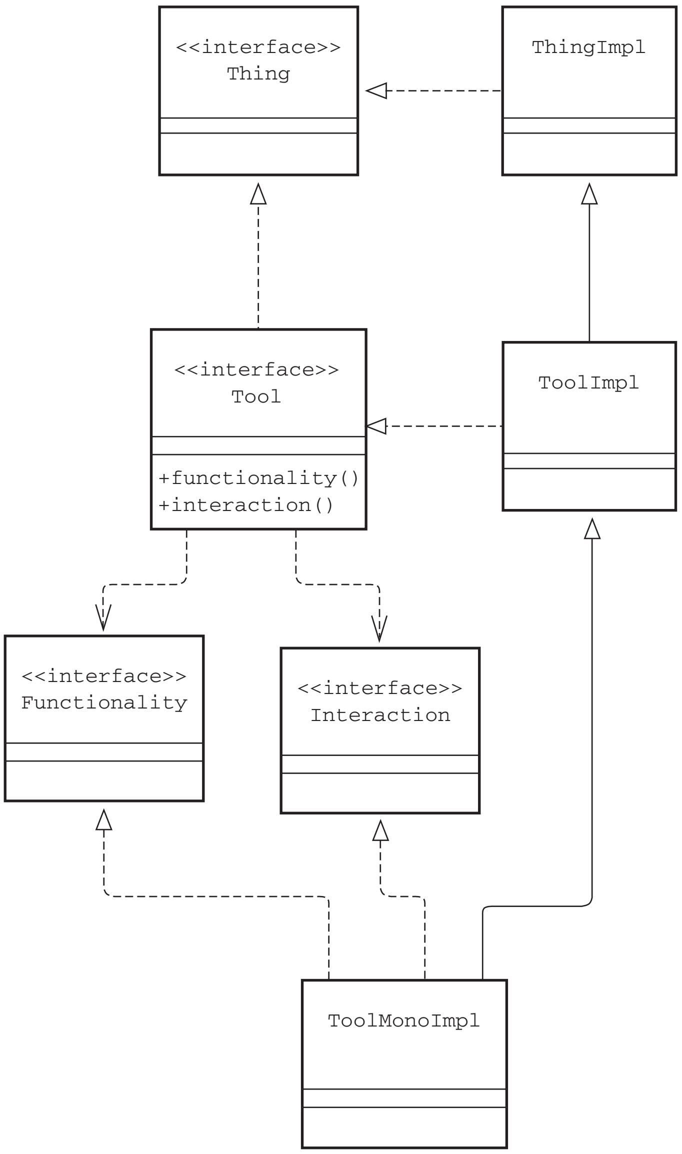


FIGURE 8.61

Monolithic tools.

widgets, it uses GUI contexts. A *GUI context* is an abstraction of windows and panels. A GUI context contains a set of user interface elements and may include embedded GUI contexts. Figure 8.63 shows an overview of the tools construction.

8.16.3 Domain Values

The JWAM framework uses classes to implement domain values. Domain value objects are always created by a related factory. Since there has to be a factory class for each

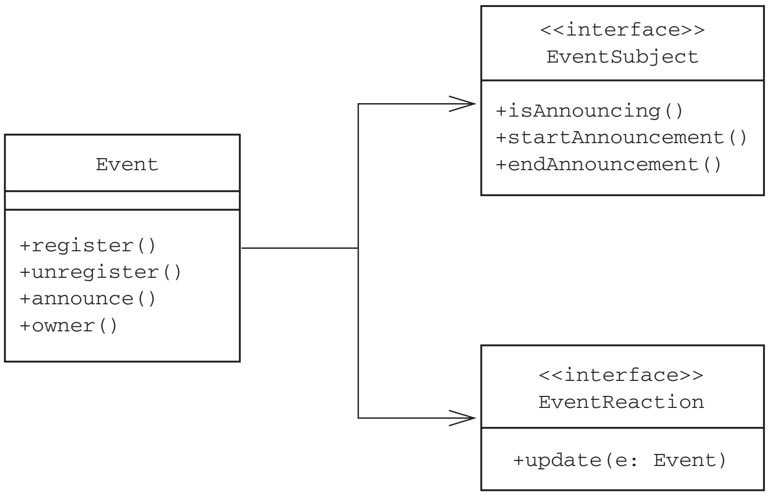


FIGURE 8.62 Example for an event pattern.

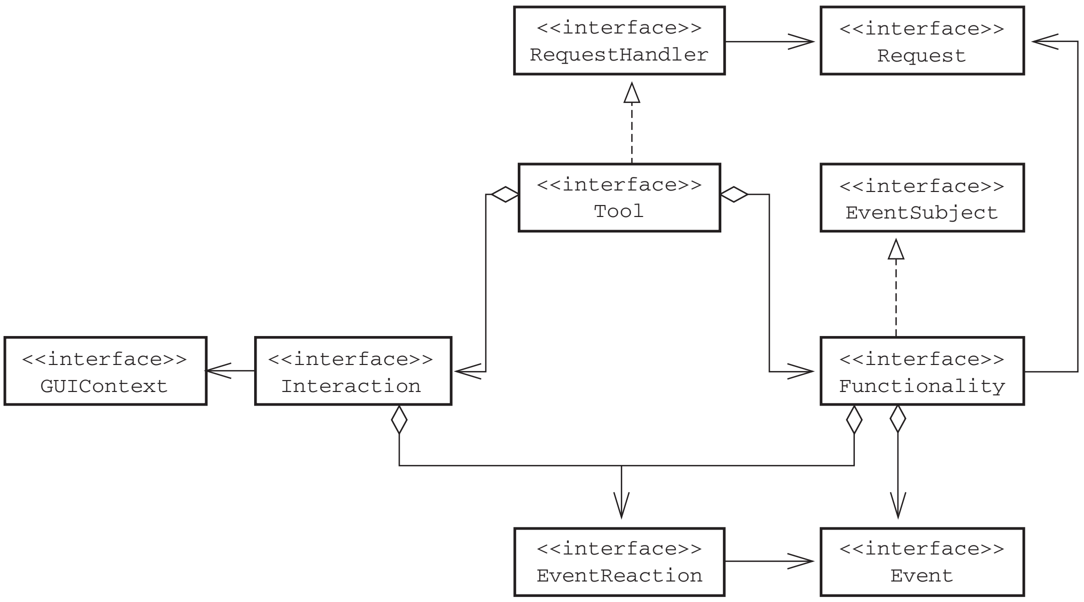


FIGURE 8.63 Tools construction overview.

domain value class, this relation is expressed in that the factory classes are inner classes of the domain value classes. There are two generic interfaces: `DomainValue` for domain values and `DomainValue.Factory` for factories. In addition, abstract implementations are made available: the class `DomainValueImpl` and the class `DomainValueImpl.Factory` (see Figure 8.64).

```
public interface DomainValue extends Serializable
{

    public DomainValue.Factory factory ();
    public String[] toStrings ();
    public boolean isDefined();

public interface Factory extends Serializable
{

    public boolean canCreateFromString ();
    public DomainValue value (String s);
    public boolean isValid (String s);
    public boolean hasUndefinedValue();
    public DomainValue undefinedValue();
    public DomainValue defaultValue ();
    public Class type ();
    public void unregisterValue (DomainValue value);
 }
}
```

FIGURE 8.64 The Domain Value and DomainValue. Factory interfaces.

Each domain value can use the operation `toString` specified in `Object` to return a string representation of itself. The operation `equals`, also specified in `Object`, can be used to check whether two values are equal. To this effect, the standard implementation specified in `Object` has to be redefined, because it compares object references. It is often sufficient to base the equals operation on the `toString` operation. No suitable one-line string representation can be found for complex domain values, so that we additionally use the `toStrings` operation, which returns a multi-line string representation of a domain value.

For domain value objects, we have to solve the conceptual problem of "special values." Though these values are outside the valid range, we still need to model them. Therefore, we distinguish domain value objects in the JWAM context by three states, which can be tested by the use of appropriate operations. These states are:

- `defined`: The object represents a valid value of the domain value's type.
- `undefined`: The object represents an undefined value (i.e., the value is not known yet).
- `invalid`: The object represents a value, which is not a valid domain value type.

As Java objects know their classes, so each domain value object in JWAM knows the factory that created it. The creating factory can be determined by use of the `factory` operation.

The domain value factory provides operations that refer to the domain value type rather than to the domain value. `canCreateFromString` can be used to check whether or not the domain value factory can create a domain value object from a string representation. Theoretically, we could expect this operation for each domain value type. However, it would be unpractical. The reason is that there is a number of domain value types that users would never enter as strings, and where expensive parsing would be necessary to create a domain value object from a string representation. Examples for such domain values are `ThingDescriptionDV` and `TableOfContentsDV`. The operations `isValid` and `value` may be called only provided that a domain value object can be created from a string representation. `isValid` can be used to check whether or not a string represents a valid domain `value` representation, while value can be used to determine the represented domain value.

Not every domain value type knows undefined values. `hasUndefinedValue` can be used to check whether or not this is the case. If, and only if this is the case, then we can use `undefinedValue` to resolve the undefined value. `defaultValue` returns the default value used to initialize things, for example, on the user interface.

Using the `type` operation, the domain value factory can provide information about the domain value type it can create objects for. Finally, `unregisterValue` allows you to delete values from the `DomainValueUniverse`. This function is useful particularly for enumeration types managed by users.

The optional construction shown in Figures 8.65–8.68 is suitable for frequently recurring domain values. To save storage space, we create new value objects from domain value factories, provided that the requested value object does not exist yet. If a value object has been previously created, then it is searched and returned from a pool of created domain values. This means that domain values are implemented by the *flyweight* pattern. To let the garbage collector remove domain value objects that are referenced only by the domain value factory from the memory, we store value objects in a `WeakHashMap`.

```
public interface Collection extends Thing
{
   public int count ();
   public boolean has (IdentificatorDV id);
   public TableOfContentsDV tableOfContents();
}
```

FIGURE 8.65

The Collection interface.

```
public interface Container extends Collection
{
   public void clear ();
   public boolean isAddable (Thing t);
   public void add (Thing t);
   public boolean isRemovable (IdentificatorDV id);
   public void remove (IdentificatorDV id);
}
```

FIGURE 8.66

The Container interface.

```
public class TableOfContentsDV extends DomainValueImpl
{
   public int entryCount();
   public ThingDescriptionDV description (int index);
   public ThingDescriptionDV description (IdentificatorDV id);
   public boolean hasDescription (IdentificatorDV id);
}
```

FIGURE 8.67

The interface of TableofContents DV class.

```
public class TableOfContentsIterator implements Iterator
{
   public TableOfContentsIterator (TableOfContentsDV toc);
   public TableOfContentsIterator (TableOfContentsDV toc,
         Class type);
   public boolean hasNext();
   public Object next();
   public ThingDescriptionDV nextDescription();
}
```

FIGURE 8.68

The interface of the TableOf-Contents Iterator class.

8.16.4 Presentation and Interaction Forms

We have seen in the preceding sections that JWAM supports tool construction based on the notion of components. Naturally, this also applies to the user interface. For example, you can use AWT or Swing to develop and integrate your GUI, or alternatively you can use the interaction forms (IAF) concept. Since the use of interaction forms is not mandatory, JWAM treats interaction forms as an optional concept.

Interaction forms are implemented as Java interfaces, and commands are used to support feedback from an IAF to a tool.

The interfaces defined by IAFs are implemented by the use of presentation forms (PF). Presentation forms are simply subclasses of JavaBeans, which implement additional operations from these interfaces. A presentation form can implement several interaction forms. Similarly, one single interaction form can be implemented by totally different presentation forms. The following interaction forms are currently supported:

- `ActivatorIAF`
- `SingleSelectionIAF`
- `MultiSelectionIAF`
- `FillInIAF`, and the following specializations: `FillInTextIAF`, `FillInIntIAF`, `FillInFloatIAF`, and `FillinDomainValueIAF`
- `SingleOfContentsSelectionIF`

8.16.5 Forms

The JWAM framework lets you easily specialize forms by subclassing the `Form` superclass. A form has a set of form fields of the class `FormField`. Each form field can store a domain value and return it for editing purposes. Since domain values can basically have an arbitrary complexity, this simple construction allows you to implement complex forms. We have successfully used the construction described in Section 8.12 in JWAM.

As mentioned there, generic standard tools are important for forms. The JWAM framework includes `FormEditor`, `FormViewer`, and `FormCopier`, For example, you can use the `FormCopier` to copy field values between different forms, even if these forms are of different types.

The component-like tools construction in JWAM lets you easily embed such standard tools in your own tools.

8.16.6 Domain Services

Domain services are things, like materials, containers, and tools. This means that the `Service` interface inherits from the `Thing` interface. The `ServiceImpl` class represents an abstract basic implementation for services.

8.16.7 Work Environment

The JWAM framework represents an environment by the class `Environment`. The implementation is based on the *singleton* pattern, which ensures that there is only one environment for each workplace. The environment uses a domain container by the name of `Workspace` to manage tools and materials. All tools generally available are held in a `Toolbox`. In turn, the `Toolbox` is a domain container in the `Workspace`. Material templates are stored in a domain container, `TemplateFolder`, which is also in the `Workspace`. We have shown these relations earlier in Figure 8.57.

8.17 REFERENCES

J. Coplien: *Advanced C++: Programming Styles and Idioms*. Reading, Mass.: Addison-Wesley, 1992.

Besides the concept of programming patterns Coplien describes the *body/handle* pattern.

E. W. Dijkstra: "The Structure of the T.H.E. Multiprogramming System." *Communications of the ACM*, August 1968, Vol. 18, No. 8, pp. 453–457.

Dijkstra's classical paper introduces the principles of layered architectures.

E. Evans: *Domain-Driven Design*. Reading, Mass.: Addison-Wesley, 2003.

As the title indicates this book introduces another application-oriented approach to software development.

E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*. Reading, Mass.: Addison-Wesley, 1995.

The fundamental reference for all basic patterns in this chapter.

A. Goldberg: *Information Models, Views, and Controllers*. Dr. Dobb's Journal, July 1990, pp. 54–61.

A relevant paper on model—view—controller.

C. A. R. Hoare: "Notes on Data Structuring." In O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, *Structured Programming*. London: Academic Press, 1972.

Seminal work on the classical type concept.

A. Koenig, B. Moo: *Ruminations on C++*. Reading, Mass.: Addison-Wesley, 1997.

Another version of the body/handle pattern.

G. Krasner and S. Pope: *A CookBook for using the Model-View Controller User Interface in Smalltalk-80*. ParcPlace Systems 2400 GengRoad Palo Alto, CA 94303, 1988.

A classical paper on model—view—controller.

B. Meyer: "Design by Contract." In D. Mandrioli, B. Meyer (eds.): *Advances in Object-Oriented Software Engineering*. New York, London: Prentice-Hall, 1991, pp. 1–50.

This paper explains the design by contract principle.

R. T. Monroe, A. Kompanek, R. Melton, D. Garlan: "Architectural Styles, Design Patterns, and Objects." *IEEE Software*, Vol. 14, No. 1, January/February 1997, pp. 43–52.

A relevant paper on architectural styles.

D. Notkin, D. Garlan, W. G. Griswood, K. Sullivan: "Adding Implicit Invocation to Languages: Three Approaches." In S. Nishio, A. Yonezawa (eds.): *Proceedings of JSSST-93*, LNCS 742, Berlin, Heidelberg: Springer-Verlag, 1993, pp. 489–510.

This paper introduces coordination mechanisms for interactive components.

D. L. Parnas: "On the Criteria to be Used in Decomposing Systems into Modules." *Communication of the ACM*, Vol. 5, No. 12, December 1972, pp. 1053–1058.

The seminal paper on modularization principles and information hiding.

A. Weinand, E. Gamma: "ET++ A Portable Homogeneous Class Library and Framework." In T. Lewis: *Object-Oriented Application Frameworks*. Greenwich: Manning Publications Co., 1995, pp. 154–194.

This paper introduces relevant concepts for framework construction.

R. J. Wirfs-Brock, B. Wilkerson, L. Wiener: *Designing Object-Oriented Software*. New York, London: Prentice-Hall, 1990.

This books introduces the concept of responsibilities and other relevant object-oriented design principles.

9

T&M Model Architecture

Conceptual and design patterns can be used as guidelines for designing and implementing the elements or components of application software. Medium-sized to large systems need a structure beyond these "micro architectures," but we cannot define the overall software architecture of a concrete application as it depends on the business logic of the domain at hand. We can, however, at least outline the structuring principles of an application built in line with the T&M approach.

This chapter describes these essential elements and features of what we call a model architecture for families of interactive application systems based on the T&M approach. This model architecture uses frameworks and components. Our model architecture is based on the two general principles described in earlier chapters: application orientation and structural similarity.

This chapter addresses readers who work as software architects, that is, those who design the overall structure and relationships of a software system. Frequently they need to develop a whole family of logically and technically related applications within an organization. This chapter provides answers to issues arising from these kinds of tasks.

The various topics presented in this chapter are based on our project experience, but we do not claim to have destilled this experience into a set of architectural patterns. Nevertheless, we frequently structure our reasonings along the lines of the pattern format used in Chapters 7 and 8.

9.1 THE T&M MODEL ARCHITECTURE

When developing large-scale software systems, we no longer choose a "big-bang" approach by building the entire system within one project. Evolutionary and iterative strategies tell us to partition a large complex system into a family of related application components, for example, according to the principles of core system and extension levels (see Section 12.7). Following this approach, we need an explicit software architecture. This has also been recognized in the Unified Process (UP), which is probably the reason why UP is also called an *architecture-centric process*. The T&M approach guides such a process with its principles of application orientation and structural similarity. According to these principles, we need to identify concepts and things found in an application domain, which can be used as elements of a software architecture.

Elements of a software architecture

We defined the elements of the object metamodel discussed in Chapter 2 so that we can use them as microelements to model application systems. To develop large object-oriented systems, however, we need macrostructures that allow us to organize these systems in a clear and easily manageable way. We have to map these macrostructures to corresponding modeling units. We call this structuring of software with modeling units that correspond to the macrostructures of our application domain *software architecture*.

A *software architecture* describes the elements of the model and of the concrete construction of a software system in their static and dynamic interplay. A software architecture itself can be represented as an explicit model. It describes a specific system in its application context.

This means that we think of software architectures in project-specific terms, because they determine how we build a software system.[1] For this purpose, we have to define modeling units, identify domain and software elements for all our models, and put them in static and dynamic relationships.

Model architecture

Consequently, we will not describe a concrete software architecture that could be used as is for many projects. What we will discuss here is a basic software architecture, including generalizations and examples. We call such a generalization *model architecture*.

A *model architecture* abstracts from the characteristic features of a set of similar software architectures. It defines the kind of elements used, their connection, and the rules for their combination. In addition, a model architecture includes criteria for the composition of elements into modeling and construction units, and guidelines for the design and quality of a specific architecture.

Notice that other authors use the term *architectural style* when talking about the basic elements of a software architecture, along with its links and rules. Our definition of "model architecture" also includes an instructive model to build families of application systems based on the T&M approach. This means that a model architecture has a similar function for the software construction as the guiding metaphor with its design metaphors for domain modeling.

With regard to iterative or sustained application development projects in an organization, we recommend using frameworks and components for such a model architecture.

1. Many developers use the term "software architecture" in a more general sense, including among other things, e.g., the way a system is distributed over different processes or computers. We do not use the term in this sense.

9.2 THE DOMAIN CORE OF A SOFTWARE ARCHITECTURE

We encourage developers to build the domain core of an application system on the basis of the large structures of the application domain in question. We illustrate this approach with examples from the financial and services industries. Based on workplace types found in the application domain, we define the use context layer. The products and services form the product layer of our architecture. Communication and cooperation of the individual workplaces can be supported by a system, provided that the domain core of a model architecture is built on a common understanding of the domain.

PROBLEM

We want more than to simply design application software based on our principles of application orientation and structural similarity within a micro world, that is, on workplace level. Therefore, we are looking for macrostructures in the application domain that can form the basis for modeling architectural units. They should serve to achieve a structural similarity between domain and software system on the macro level.

BACKGROUND: THE APPLICATION DOMAIN CORE

Large (and object-oriented) systems are often structured on the basis of technical criteria. For example, a division of systems into user interface, application logic, and data repository is customary in practice. This structure is normally called *three-tier architecture* and is very popular today. We will discuss this architectural approach in more detail in Section 9.3.6.

Macro structures

In contrast, application orientation suggests that, rather than orienting ourselves to technical criteria when structuring a software system, we look for suitable macrostructures in the application domain. In this connection, we take the following requirements (see Section 2.6.8) into account:

- The domain model of the application domain determines the technical model for our application system.
- The concept model as part of the application domain model should be mapped without mismatches to classes and class relationships. To this end, domain concepts and their structures have to be recognizable in their system representation. Classes and their relationships form the microelements of a software architecture.

SOLUTION

Many software applications are developed for specific organizations or parts of an organization. When identifying macrostructures for our software architecture, we orient ourselves to the structures prevailing in that organization.

DISCUSSION: THE APPLICATION DOMAIN CORE

Once a large system has been introduced, basic architectural decisions can hardly be changed due to the high costs such changes involve. For this reason, we should structure the system so that the macrostructures we select will remain valid over the long term.

Application orientation suggests that we should find these structures inside the organization that will be using our application software. In this respect, we are interested in structures that represent macrostructures and a relatively high life cycle, that is, more than ten years.

Organization structures as macro elements

Economists normally distinguish between the *company organization* structure and the *flow organization*. We are less interested in the flow organization, where we often observe quick changes in the dynamics of task allocation and completion due to information technology. The same applies to the work organization, which deals with the organization of the work of individuals and groups.

The organization structure represents the static aspect of a company. This corresponds to the notion of software architectures, which primarily describe the static aspects of a software system, rather than its dynamic behavior at runtime.

Our experiences in the design of software architectures are based mainly on the financial and service industries, where companies are normally organized by business sectors, product lines, or other object-specific principles. We describe a model architecture that has proven useful for this corporate type.

This does not mean that other organizational types are not suitable for structuring software architectures. The important point here is that we want to characterize the organizational type behind the application domain of the framework-based architectural model we selected to ensure that our model is highly applicable.

Background: The Object Principle as an Organizational Structure

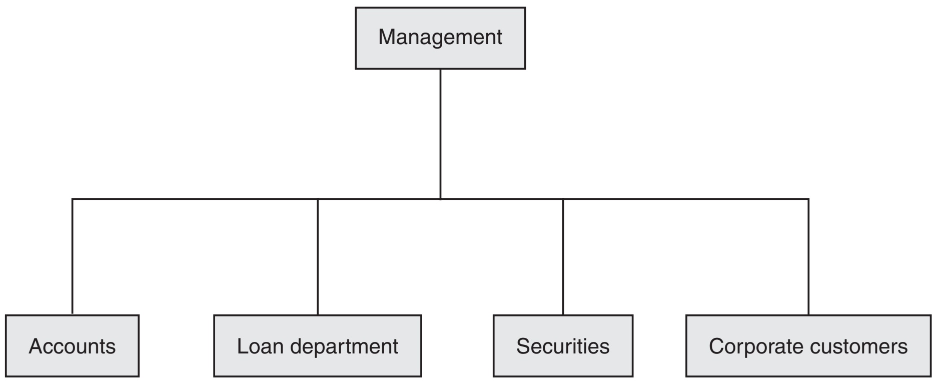
This section will discuss companies that are organized by business sectors. This organizational structure treats products, regions, or customer groups as objects, while the corporate organization is oriented to these objects. For this reason, economists often refer to this organizational principle as the *object principle*.

In a bank, for example, you normally find credit, securities, account management, or corporate customer divisions, where each division is responsible for a specific group of financial products (see Figure 9.1).

The object principle follows the goal of allocating a product or service within a division so that it has maximum independence of the other divisions in completing its job. This independence of divisions means that the work objects of a domain have little or no relationship to the work objects of other domains.

In banks, for example, this object principle is normally realized so that customers can handle all their business in relation to their checking accounts at a bank's counter

FIGURE 9.1



Simplified chart of a bank organization by the object.

or teller. Only when a customer expresses interest in taking a mortgage or another important transaction is this customer referred to the credit division.

In this organizational form, the work objects are in the center. Each workplace and its organizational unit is responsible for carrying out a complex task with the work object. The object principle matches our expert workplace T&M guiding metaphor (see Section 2.3.5). This means that it is oriented along objects, where these objects can be specific products or services for customer groups. In our context, we can speak of work objects in either case.

Modularization

This type of corporate organization represents a macrostructure that we are looking for as a foundation for our model architecture. It is relatively long-living and stable. In addition, it offers an organizational principle similar to the modularization principles of Parnas or Yourdon. This means that it allows us to apply two important concepts, minimum coupling and maximum cohesion, to the macrostructures of a software architecture.

9.2.1 The Use Context

Domain analysis and modeling by the T&M approach are based on the central idea of dealing with the specific tasks at each individual workplace, because these are the locations where our future application system will run. When studying the number of different workplaces in financial and service industries, we often observe that many workplaces handle related products or "service packages," due to the customer orientation of these companies (see Section 1.1.1).

Customer orientation

In this connection, customer orientation means that the needs, expectations, and requirements of the customers are the center of attention. Basically, customer orientation is implemented so that the company orients the organization of its "interface" to the external world primarily to customer needs and wishes, in addition to its product and service offer.

The "internal structure" of many service companies is divided by appropriate business sectors or divisions, while the customer-contact domains have been reorganized. This means that we often find customer-oriented workplaces that are no longer oriented to single product domains but rather to customer requirements. This leads to a wide product and service range that has to be represented and handled differently at different workplaces. If we want to develop application systems for these workplaces, then these systems also have to offer differentiated support for a large number of products. In addition, such companies often operate separate application systems that are used by their customers.

At the same time, the above reorganization means that the customer image changes, that is, there is no longer *the* customer. Customers are segmented in different groups. The aim is to identify and address homogeneous customer groups, depending on their requirements and potentials. The result is that the same product or service package is offered in many different ways.

Use context

First, let's define our notion of use context.

The term *use context* refers to the part of an application domain where specific work is supported by an application system. A use context includes the specific work context supported by an application system within an organization. This work context is normally realized as an independent workplace based on a

well-defined set of tasks. Products and services from different divisions or areas can be handled, depending on the use context.

The use context forms the background or starting point for domain analysis and modeling. It also defines the extent of each workplace system.

EXAMPLE: USE CONTEXT

Bank example

Modern banks are increasingly shifting from traditional banking services to comprehensive financial services. Thus the financing of a house purchase is no longer only implemented as a mortgage. Investment opportunities as well as life insurance to secure loans and cover long-term repayments are offered during the capital structuring phase.

This example can be used to show a differentiated product presentation. We find that loans granted within the context of personal advisory services are geared to retail banking customers. Very well-off private customers are often offered a separate service that includes more sophisticated credit products. Then there is a standardized spectrum of credit products in the service center, simple customer loans in home banking, and online banking.

DISCUSSION: USE CONTEXT AND WORKPLACE TYPES

Use contexts can often be easily mapped to the different workplace types in the application domains discussed in this book. This means that specific tasks are completed at specific workplaces, handling related products or services in a customer-oriented way. For example, we identified the following workplace types in the banking industry:

Bank example

- *Workplace for personal account manager*: This workplace type is intended for private customer account managers. Products from the loan, investment, and securities departments are offered based on the "one face to the customer" principle, concentrating on flexible work oriented to the customer (and the bank). The objective is to offer private customers personal assistance in their banking business.
- *Workplace for product specialists*: This workplace type offers specialized support for various product areas, such as loans, investments, and securities. The respective bank specialists deal with customer requirements that cannot be handled by private account managers.
- *Workplace for tellers and general retail banking activities*: The important characteristic of this workplace type is quick and reliable support for standardized services. Examples include workplaces in service centers and monitoring of transfer traffic in banks or back-office areas.
- *Home banking and self-service terminals*: This workplace type offers standardized banking services for people who are not familiar with banking or software, providing support for inexperienced users.

Workplace types

The above bank-specific workplace types correspond to the general workplace types described in Section 3.6. This means that we can allocate them to different guiding metaphors:

- The workplace types for personal account managers and product experts fall in the category of the expert workplace (see Section 3.6.2). An application system for this workplace type should support the account manager or product expert in their tasks while giving them maximum flexibility. While the system for account managers offers a broad range of products from different product

areas for the so-called 80 percent case (i.e., covering 80% of all business cases), the product specialist should be supported in all the details of a product area.
- The workplace types for tellers or business correspond to the functional workplace type, as discussed in Section 3.6.3, which is focused on quick independent completion of standardized tasks. An appropriate application system should offer several standard products from different areas or easily provide optimized support for frequently recurring tasks.
- The home banking and self-service area corresponds to the workplace type for the electronic commerce front-end (see Section 3.6.5). From a banking view, different products have to be covered and presented in an intuitive manner for nonexpert users. This workplace type has to be easy to use and intuitive.

DISCUSSION: USE CONTEXT AND WORKPLACE TYPES

The identification of different use contexts in an application domain is an important step towards reducing complexity for analysis and modeling purposes. This means that we do not have to construct a monolithic system that covers all tasks and every functionality required. Moreover, we do not have to cover all application contexts in one project; instead, we can gradually do our work step-by-step based on domain and software priorities. This is in line with our development strategy, as described in Section 12.7.

Relating use contexts and workplace types

Relating use contexts and workplace types solves a second major design problem. For different detailed tasks people with different qualifications should have shared access to the products and services of an organization. In proposing a workplace concept for software systems within an organization, we suggest equipping these workplaces in different ways. Thus, the way a loan is offered at different workplace types can be differentiated in a technical sense, and the presentation and handling of each workplace type can also vary in the usage model. Nevertheless, we want to ensure that it is based on the same banking functionality.

Consequently, we assume that workplace systems have to be designed differently, depending on the use context, in order to meet the respective requirements. This means that each use context always forms a separate modeling unit. It is, however, important that these systems can be used in the same way, allowing an intuitive understanding of common domain concepts. Bank employees often move between different workplaces or assume different roles, sometimes more than once on the same day. This means that a family of application systems is based on a common usage model so that interaction forms familiar to a user can be transferred.

Common design metaphors and patterns help us in designing a family of integrated and uniform application systems. In addition, tools, materials, automatons, and all the other design elements of the T&M approach form constructive software development units.

We demand that application systems for different use contexts be constructed through an extensive combination of elements that have already been implemented in the individual product domains.

9.2.2 The Product Domain

In the financial sector, we often find companies that are organized according to business or technical departments. Each organizational unit groups itself around a product or service type. Examples in the banking sector include the loan and securities departments. This classification has the advantage that it produces homogeneous technical units that are coherent in themselves and have no major cross-references to the products and services of other units.

One of the most important prerequisites for the model architecture we propose is that these departments are units motivated by the domain and largely independent of other units. In this case, the products or services located there can be modeled so that the individual models, in turn, incorporate minor dependencies.

This business organization is also important when specific use contexts, as presented in the preceding section, are customer-oriented. Product domains offered at the different workplace types can still be identified.

Definition

We separate the term *product domain* from the specific organizational unit, that is, we talk about a product domain even if a company no longer has a corresponding department.

A *product domain* is a domain group of relating products or services. Within an organization, there are normally several identifiable product domains, reflected also in the organization's internal language. In addition, there may be several organizational units, each in charge of one product domain.

From the software development view, a product domain is a part of the application domain, which can be analyzed, modeled, and built independently of other product domains. A product domain is a macrostructure relevant for the software architecture, and this macrostructure can be used as a modeling unit.

Example

Bank example

A banking example can be used to illustrate the product domain concept in its traditional and current form.

A customer's financing request to build a house is dealt with extensively in the loan department. This is a classic business organization. This means that the product domain, a loan, still corresponds to an organizational unit.

Alternatively, a bank has a service center where customers can obtain simple consumer credits along with other standard products. A person in the service center advises customers about the range of products offered and looks after their requirements. However, there is still a clear understanding among the bank employees that the services offered at this center, such as loans, investment advice, and teller services, belong to different product domains.

Discussion: Product Domain and Use Context

The banking example shows that use contexts and product domains seldom form a one-to-one relationship in today's companies. Use contexts supported by appropriate workplace types are subject to high dynamics, and they are oriented towards the needs of customers and changes in the market.

In contrast, business domains are more stable conceptual dimensions that can easily be analyzed and modeled one by one. We could say that, at the domain level, they correspond to the modularization principles of Parnas or Yourdon. They normally show strong coherence and weak coupling.

Modeling the different product domains as separate modeling and construction units has the following advantages:

- Within an overall project, each domain can be modeled and implemented in a separate subproject.
- Tools, materials, and other equipment elements used to support a product domain can be arranged differently for different workplace types, depending on the use contexts.
- Product domains can serve as a basis for the logical architecture (see Section 9.37) and for dynamic components, that is, domain services (see Section 8.14).

9.2.3 The Business Domain

The previous section demonstrates the importance of uniformity in the design of different application systems. This uniformity must have a technical and a domain basis, which means that it is not sufficient to ensure that all elements of a family of application systems behave in the same way, such as tool-like or material-like. They also have to be backed up by a uniform *domain* structure of terms and concepts. This is something we have to ensure beyond the use contexts and product domains already discussed, since there the emphasis is on the domain differences between products, services, and tasks.

Business domains overlap

When looking at the individual product domains, we can identify conceptual overlaps. For example, the loan and securities departments in a bank need to share concepts, such as account or currency. In addition, there are overlaps not immediately apparent, for example, concepts like "borrower" and "securities holder" can relate to the same customer.

These overlaps should not come as a surprise, since a common business stands behind the different product domains. After all, employees at different workplaces are able to communicate with one another about their work, although they all have different stores of detailed knowledge and a different understanding of specialized terms. In this sense, we can still say that all use contexts and product domains share the same common abstractions.

The uniform structure of terms and concepts is important for another reason. It is the only way to support cooperative tasks that extend beyond one workplace or product domain. Naturally, the people involved have to speak a common language to be able to cooperate. In addition, cooperation works only provided that there are common work objects and materials. For example, when a bank employee forwards a transfer form in an electronic folder, and this form appears identically on the electronic desktop of a colleague, then these two employees can cooperate smoothly.

EXAMPLE

Bank example

Let's look at another example. In a bank almost all workplaces need to have basic information about customers in a consolidated form. Technically, this is called a customer information first screen. In the securities area, this screen contains the deposits belonging to a customer, the shares contained, and general information about this customer (e.g., name, number, and a listing of all activities with the bank). Detailed information about deposits is not necessary for customer information first screens in the loan department. This department's prime interest are the securities provided by a customer for collateral of credits, in addition to credit accounts and credit data.

However, both departments need to know that this customer is represented by the same information, such as name or customer number. Without such a common domain basis, two bank employees could not talk about the same customer across their departments.

Evaluating this example shows that modeling the conceptual commonalities of an application domain as part of the different product domains leads to problems. The common identification data about a customer would have to be managed in duplicate in both product domains, in this case loans and securities. From the technical and domain viewpoints, this leads to inadequate solutions. In addition, it entails the risk of introducing inconsistencies in further development.

Definition

We need a common domain basis to be able to build integrated application systems. We build this basis by use of an independent modeling unit called *business domain*.

> **A *business domain* includes the domain concepts or core abstractions that are fundamental for different product domains. It characterizes the business of an organization on a general abstraction level. The business domain forms the domain and constructive basis for all product domains. It is a separate modeling unit.**

Discussion: Business Domain

A business domain includes common basic concepts for the different product domains, defining the domain language necessary to a common understanding and common work objects. Constructively, the business domain represents a model common to all product domains. The elements contained in this model and their relationships are binding for all product domains.

This may initially sound like the requirements associated with corporate data modeling. We know that corporate data modeling is difficult. In the banking industry especially, there have been many unsuccessful attempts (at least from the user's perspective) to implement standard data and conceptual models. To avoid such problems, we have to bear the following things in mind:

Criteria for designing business domains

- A business domain must include all concepts and objects necessary for communication and cooperation between individual product domains. These concepts have to be represented in commonly used construction units. If the business domain is overly constricted, then it will either be difficult for the product domains to communicate, or domain elements have to be modeled several times for each product domain. This hinders the development of integrated application systems.
- The business domain should be minimal, that is, it should not include concepts or objects required in only one or a few product domains. This would make the idea of the business domain unclear and unnecessarily increase the number of constructive dependencies between a business domain and product domains.
- The concepts and objects included in the business domain should ideally be implemented in the form of interfaces and abstractions. The business domain should specify only those implementations that may be useful for all product domains concerned. Accordingly, few elements of a business domain are normally provided as components or completely implemented classes.

The Business Domain as an Ideal Reconstruction

One of the most important characteristics of a business domain is that it does not have a direct correspondence in the company's organization; instead it is a conceptual entity, we can only identify on the modeling level of the domain model and the system model. There is normally no business unit or department which represent the business domain directly. The business domain is an abstraction form the entire common business of a company. However, this is not constructive, because we cannot model the organization or its business as such. We are always confronted with specific workplaces or product domains as organizational units. Behind these organizational units, there are common elements that we want to model in the business domain, but which are idealizations and normally not physically present. In a way, we have to reconstruct the core abstractions behind these common elements or notions.

In addition, the people in an organization seldom look at their work in the sense of our business domain. We already mentioned its difference from traditional data modeling. This difference also applies to business process engineering, where modeling is focused on processes and rational flows or value-adding chains rather than on the common abstractions behind work objects and interactions.

We can conclude, then, that modeling a business domain is a complex and demanding task for development teams, and one that should be handled with great care and based on extensive experience.

Example

Bank example

We can use bank accounts to understand the problems inherent in an ideal business domain. Almost all departments and workplaces deal with customers and accounts. However, there are different views of what characterizes these customers and accounts, and which tasks have to be completed in what form.

Using bank accounts as our example shows a major problem. There is no location in a modern bank where developers can see an account as such. The days when accounts existed in the form of entries in ledgers and cashbooks are long gone. Accounts have become virtual artifacts that ultimately can only be reconstructed. We understand what an account is when we analyze how it is processed at the various workplaces involved.

Looking at existing bank applications doesn't help us either, because in conventional banking mainframe software, data belonging to the account concept is stored at various locations, and its processing is distributed over many separate programs.

An Approach to Business Domain Modeling

As we have seen, modeling a business domain is a demanding task that can only be handled with a suitable approach. For example, you will have to build a separate business domain only if your software has to support at least two overlapping product domains (for example accounts and loans). If this is clear in advance, you can identify and model an initial business domain during the construction phase of the first product domain (for example accounts) by generalizing potential candidates. This means that developers need to have domain knowledge, in addition to being able to create good object-oriented designs. The reason is that they have to anticipate which objects and concepts will be part of the business domain.

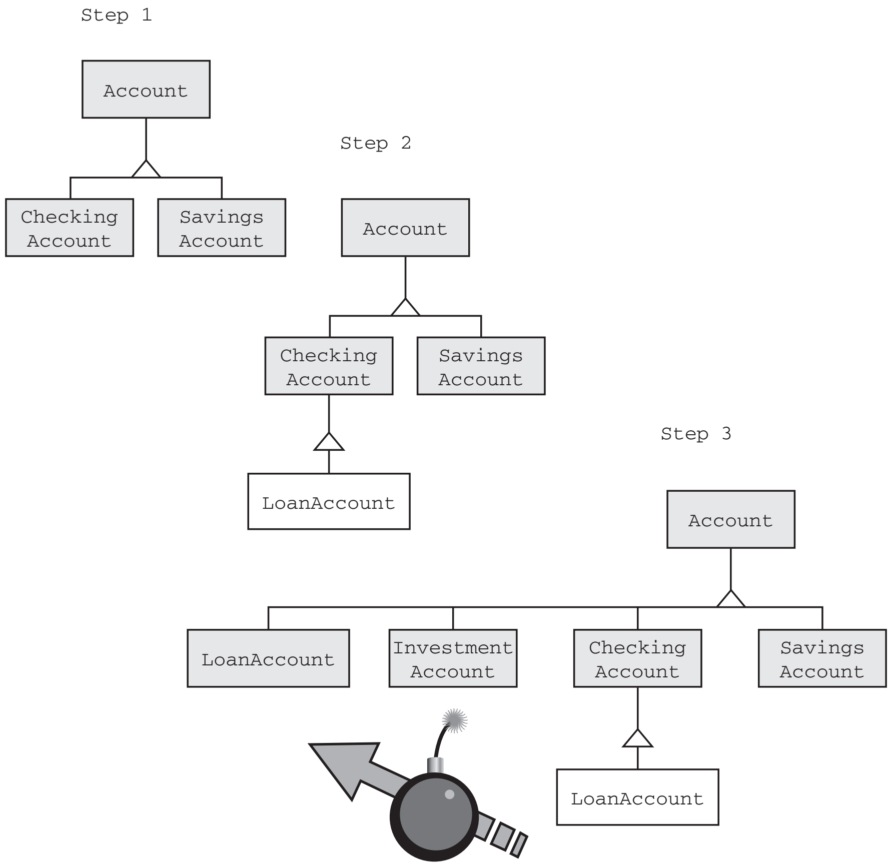
Our project experiences have shown that we must begin developing a common business domain when we model a second product domain (for example loans). This appears to be the only way to ensure that two product domains remain independent of one another. Normally we are tempted to develop the second product domain on the basis of the first one especially when there are hard deadlines (see Figure 9.2). This is not a good idea, because we would then introduce unnecessary interdependencies between the two product domains, so that changes in the first (accounts) could have unwanted implications in the second (loans). It would prevent the two product domains from developing independently, which is one of the most important reasons for our search for macrostructures as the basis of our model architecture.

EXAMPLE

Bank example

Assume that we have to deal with the following situation. The product domain for teller business in a bank has been modeled and an application system has been built. The developers elaborated a simple concept model for *account* realized by a superclass, `Account`, and two derived classes, `CheckingAccount` and `SavingsAccount` (see Figure 9.2, step 1). When requested to additionally support the bank's loan business (see step 2), the developers find that important characteristics of a loan account had already been modeled in the `CheckingAccount` class. They merely add the management of securities in the subclass `LoanAccount` to `CheckingAccount` and specify a precondition to ensure

FIGURE 9.2 Problematic coupling of two product domains.



Zull-09.qxd 20/8/04 7:40 AM Page 293

that the loan account cannot appear on the credit side. During modeling of yet another product domain (see step 3), the bank's investment business, it becomes clear that the class `InvestmentAccount`, positioned on the same level as `CheckingAccount` and `LoanAccount`, would describe the concepts and relations more appropriately. But this change would have an undesired impact on the application for the loan business.

This example shows that we need to continuously adapt and enhance the business domain to support newly added product domains. This is the only way to ensure flawless extensions of an integrated application system.

9.2.4 How Different Domains Relate

We analyzed the organizational structures in the application domain to structure large-scale software systems, focusing on companies organized by the object principle.

Macro structures and modeling units

For these companies, we have identified different workplace types and use contexts as our primary level for application-oriented modeling. We identified various product domains as the important underlying macrostructure for the domain concepts and objects. Finally, we reconstructed the basic concepts for the business domain of a company. When we try to transfer these structures to the potential modeling units, we can see the following relations:

- The *business domain* is modeled only once for each application domain. Its elements are the fundamental concepts and objects of the company. If a company has more than one business (e.g., a hospital with clinical departments and a nursing school), then we have to check whether or not we should model separate business domains. This depends on tasks spanning these domains and the need to support them within the new system.
- The business domain is reconstructed on the basis of several *product domains*. It is part of the application system model, because it has no direct counterpart in an organizational unit.

  The product domains model appropriate units of the corporate organization. Each product domain is represented as a current concept both in the application domain model and in the software model. The models of the set of product domains should be independent of one another. When modeling a new product domain, we either extend existing concepts of the business domain or introduce new concepts for that product domain.
- Corresponding to the workplace types in a company, there are different *use contexts*. Each use context associates a workplace type with the appropriate application system. Each use context is considered a modeling unit in the domain model. In addition, we create one software model of every use context for each application system. One use context can cover several product domains. Accordingly, our application systems are normally composed of elements from the product domains we need.

EXAMPLE

Bank example

In our banking example, we can see the models involved and some of the elements each of these models contains (see Figure 9.3).

The bank in this example has use contexts that are bound to specific workplace types, including account manager, securities specialist, or teller workplace. In addition, there are self-service terminals for bank customers.

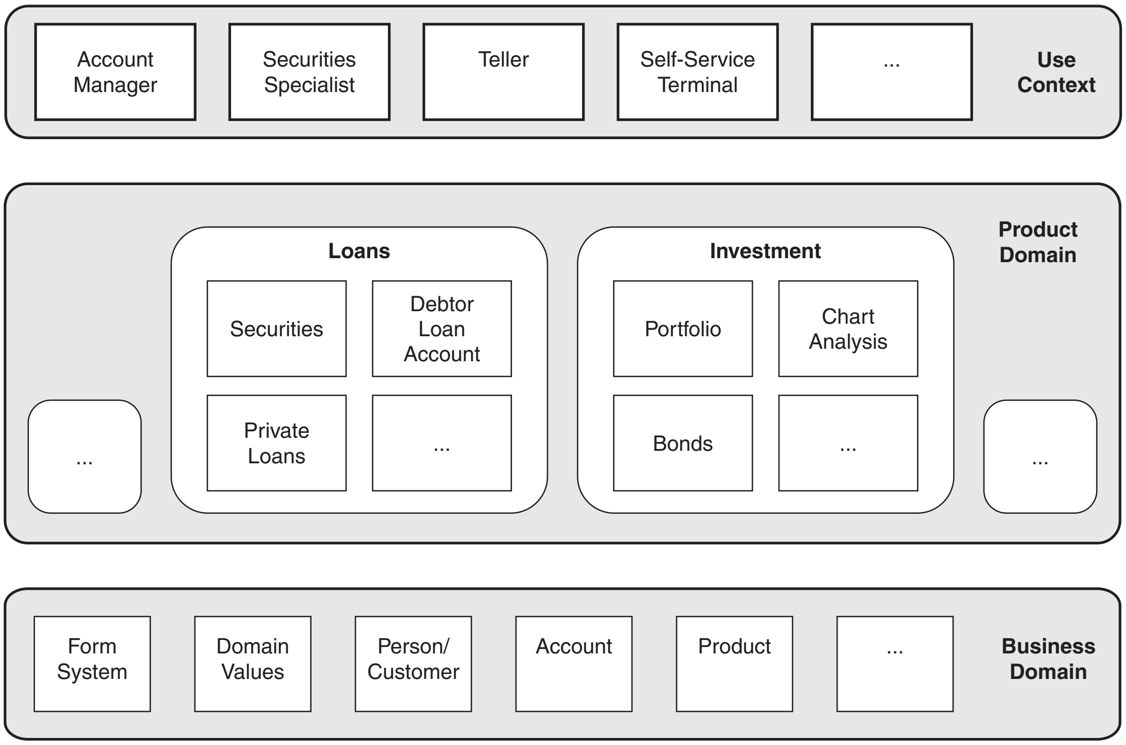


FIGURE 9.3 Business domain, product domains, and use contexts for a bank.

We already mentioned the traditional product domains of a bank, including loan, investment, securities, and teller divisions. All of these product domains have the customer and account concepts in common. More specifically, all of them use the object checking account, because this does not require any specification in the product domains involved. In contrast, the customer concept is extended in the loan domain. A totally new interaction is the management of securities, which is added to the application.

In the business domain, there are initially the bank-specific values, such as amount, interest rate, and account number. In addition, there are basic concepts, such as person and customer, where customer extends the concept of person.

Impact of Domain Changes to the Architecture

New workplace types that can be implemented on the basis of existing product domains do not cause any problem. The same applies when we can create a new product domain with its own workplace types so that the existing business domain and other use contexts are not affected.

Changes due to new products

If conceptual changes in the business domain are required due to the introduction of a new product domain, then they will not be problematic as long as we can extend existing concepts of the business domain. Also, it would be relatively easy to add a new element to the business domain. Both types of changes are covered by the open-closed principle proposed by Meyer (see Section 2.2.3).

New concepts

Moving concepts

In contrast, extensive changes are required if we have to move a concept from an existing product domain, that is, when we have to generalize it in the business domain. In this case, we have to modify both the business and the product domain.

Bad abstractions

We will have to deal with serious modeling problems if we find that the concepts of the business domain are poor or wrong abstractions. Such problems typically stem from a limited comprehension of the application domain on the developer's side. The risk of such errors can never be totally eliminated. It is especially high when developers try to create a complete model based on the idea of the waterfall model (see Section 12.1.3).

Changing business

A fatal problem occurs when the concepts of the business domain no longer match the real-world work situation of an organization due to fundamental changes in an organization's business. If even the underlying abstractions have changed considerably, the application software is normally no longer useful and will have to be rewritten to the new business of an organization.

HANDLING *DOMAIN CHANGES* WITHIN THE PROJECT TEAM

We assume that several project teams work in parallel to implement one domain concept model in different technical models for an entire organization. The question is, who should estimate and implement changes to this concept model?

- As long as new use contexts and workplace types are implemented, there should be no changes at all, or minor changes may be implemented by each project team directly.
- We have to carefully consider two important aspects for each new product domain. First, are there unidentified common concepts in parallel projects? Does the modeling effort in one product domain entail changes to the business domain, and what impact would these changes have to the other product domains? These changes have to be coordinated among all projects.
- The business domain itself has to be verified regularly to ensure consistency. Any impact on dependent product domains has to be identified and implemented involving all projects.

The above factors have organizational consequences. All changes have to be coordinated or implemented by a cross-project instance. We suggest a so-called architecture group (see Section 12.2.6) for this purpose.

9.3 CONCEPTS AND ELEMENTS OF A T&M MODEL ARCHITECTURE

The previous sections describe corporate organizational structures and show a way to use them in our modeling process. We define macrostructures and modeling units that primarily concern the application domain model, but also represent a fundamental definition of structures for our software model. This section explains the technical concepts and elements required to define a model architecture.

To define a model architecture we initially orient ourselves to the principles of a classic layered architecture. We use this layered architecture to explain the basic idea of a model architecture and explain why the popular *three-tier architecture* entails a number of problems. Next, we will define important rules that an object-oriented layered architecture should observe.

BACKGROUND: WHAT TYPES OF ARCHITECTURAL CONCEPTS ARE REQUIRED?
Section 9.2 divided the domain concepts of an example domain into possible model elements. However, we dealt with neither the technology required for programming nor the aspect of handling and presentation (see Section 6.7.5).

To build interactive application systems based on the T&M approach, the underlying software architecture has to include important object-oriented characteristics, such as abstraction, polymorphism, or dynamic binding (see Chapter 2). To this end, we have already presented general software engineering principles, such as, cohesion and coupling or the open-closed principle, which are used to build design and construction units.

Interaction of elements

We have to understand the interacting elements of a software architecture, as well as how they can be described in a model architecture. For this purpose, we define how these elements are organized and structured. One particularly important point is to understand how these elements interact. Examples for this interaction include an operation call on an object or the eventing mechanism between functional parts (FP) and interactive parts (IP).

Since we talk of an object-oriented model architecture, the use relationship and the inheritance relationship are suitable mechanisms to link elements. However, since we deal with frameworks or components as the elements of our architecture, it is reasonable to describe the interaction of these elements on a higher conceptual level. For this purpose, design patterns are ideal, as described in Chapter 4. We will present two additional patterns in Section 9.4, which have proven useful for large software architectures for interactive systems.

9.3.1 Components of an Object-Oriented Software Architecture

Based on our discussion thus far, we can identify the following elements and connectors of an object-oriented software architecture (see Chapters 2 and 4):

- Elements include class libraries, components, and black-box and white-box frameworks, because they are object-oriented macroelements above classes and objects.
- Connectors include inheritance and the use relationship, typically structured according to design patterns. Both connector types, inheritance and use relationship, can couple frameworks. For components, connectors are restricted to the use relationship. According to our idea of components (see Section 4.5), there should be no inheritance relationship between components and the embedded application system. Class libraries are normally linked by inheritance and use.

9.3.2 Elementary Rules for Combining Elements of a Software Architecture

Based on software engineering principles and our project experiences, we can describe the following general rules for combining the elements of a software architecture:

- Frameworks can be connected in various ways, but the coupling should be minimal in all cases, and loose coupling is the preferred method (see Section 2.1.23).
- Cyclic structures should be avoided, since they can make the development and configuration of new elements more difficult. If structures have to be linked

reciprocally, then loose coupling should be used in one direction (see, e.g., the *observer* pattern between FP and IP in Section 8.6).

- Components should always be used, and they should employ only interfaces of the embedding environment. Inheritance relationships should be avoided.
- Class libraries should be used as basic elements only; they should not depend on other elements of the architecture.
- A well-defined and sufficiently documented set of valid design patterns should be available for each specific software architecture. This set should be extended only gradually and by careful coordination; otherwise, there is a risk that new patterns will not become part of the common development culture.

9.3.3 Protocol-Based Layer Architectures

We assume an application domain as our starting point for the domain core of a software architecture. Figure 9.3 depicts this layered structure, as we saw earlier. These three layers are, from bottom to top, common abstractions of the business domain, the set of different parallel product domains, and the use contexts. Each of these layers depends on the next lower layer. If we look at these domains as modeling units, then we obtain a structure that is commonly called *layer architecture* in software engineering.

EXAMPLE: THREE-TIER ARCHITECTURE

The most common example of a layer architecture in the literature and in practice is a *three-tier architecture*, normally used for client-server applications (see Figure 9.4). This architecture divides a software system into a presentation layer, a functional layer, and a data layer. On the top of the architecture, the *presentation layer* is responsible for graphical representation of an application. The *functional layer* implements the business logic. And finally, the *data layer* at the bottom of this architecture represents data and provides an interface to a database, if the system uses one.

A three-tier architecture meets the fundamental requirements for a layered architecture, that is, elements may access only elements of their own or a lower layer.

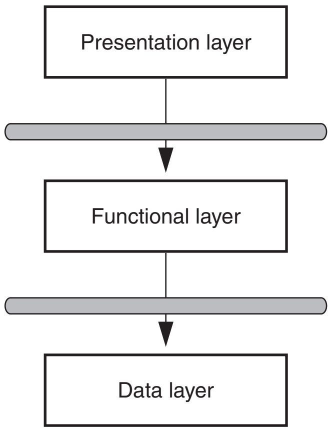


FIGURE 9.4
A three-tier architecture.

This means that each layer provides an interface for the next higher layer, and this interface represents a service, that abstracts from specific characteristics of the lower layer.

Characteristics of a Protocol-Based Layer Architecture

Using the example of a three-tier architecture, we can easily identify the characteristics of a specific type of layer architecture, commonly called protocol-based architecture:

Characteristics of a protocol-based layer architecture

- A protocol-based layer architecture groups related elements in a layer. This layer provides a protocol for the use of a coherent set of bundled services.
- This architecture organizes its layers in a hierarchy, which means that the services of one layer are available to the elements of this layer and the next higher layer. Thus one layer knows the next lower layer but it does not know a higher layer. A three-tier architecture shows that the next higher layer should be informed when there are changes, such as, to data objects, in its lower layer. Otherwise, we would not be able to update the presentation at the interface. To allow layers to inform their upper layers about such changes, we should use a minimal signaling mechanism.
- The hierarchy should express an abstraction between the layers. The lowest layer is then the concrete basis for abstraction. Higher layers abstract from the characteristics of the lower layers by providing more abstract and extensive protocols. We will discuss this issue for our example of a three-tier architecture in Section 9.3.6.
- For downward communication, it appears meaningful to limit use to the immediate lower layer, because the deeper layers should be hidden by protocols residing at the upper layers.
- Each layer is normally implemented in the form of a module, which offers a protocol at its interface, while strictly encapsulating the implementation of this protocol. The protocol is separated from its implementation to ensure that any change to the implementation will not have any impact on the use of this layer, according to the information-hiding principle. Of course, protocol changes will have an impact on the elements of higher layers that use this protocol.

We can use these characteristics to define the term *protocol-based layer architecture*:

A *protocol-based layer architecture* organizes a system in a hierarchy. A *layer* provides a *protocol* of well-defined services for the elements of its own layer or the next higher layer. Each layer abstracts from the next lower layer by offering a higher protocol, where *higher* means that the protocol is closer to the application domain.

The protocol is separated from its implementation to ensure that changes to one layer will be limited to local impact and that each layer can be developed independently of all other layers.

Discussion

Building software along protocol-based layers is suitable particularly when (technical) components or concepts have to be encapsulated and provided in an application-specific way. We can encapsulate the technical components of one layer and use a more abstract protocol to offer them to higher layers. In addition, strict encapsulation ensures that each protocol-based layer can be developed independently and that changes will have no impact on other layers.

Such a layered architecture is well suited for the implementation of a client/server application.

EXAMPLE: THREE-TIER ARCHITECTURE FOR DISTRIBUTED SYSTEMS

The layers of a three-tier architecture are often implemented as independent processes that run on separate computers. The presentation layer assumes the role of a client, the functional layer assumes the role of a server for the user interface and the role of a client for data objects, and the data layer acts as a server. This allows us to describe specific configurations for front-end PCs, local application servers, and centralized back-end machines. Figure 9.5 shows an example of a three-tier architecture.

9.3.4 Object-Oriented Layer Architectures

We have to understand what a protocol-based layer architecture means in the object-oriented sense. Obviously, we use frameworks, components, and class libraries to implement layers. If we consider only the static logical architecture of an application, then a layer organizes its elements in a logical unit, but a layer is not a software element itself. This means that a layer does not have its own interface; instead, it offers the interfaces of its elements.

DEFINITION

In contrast to protocol-based layer architectures, we define object-oriented layer architectures as follows:

> **An *object-oriented layer architecture* is composed of layers, forming a software or domain unit. The layers are organized hierarchically by the principles of**

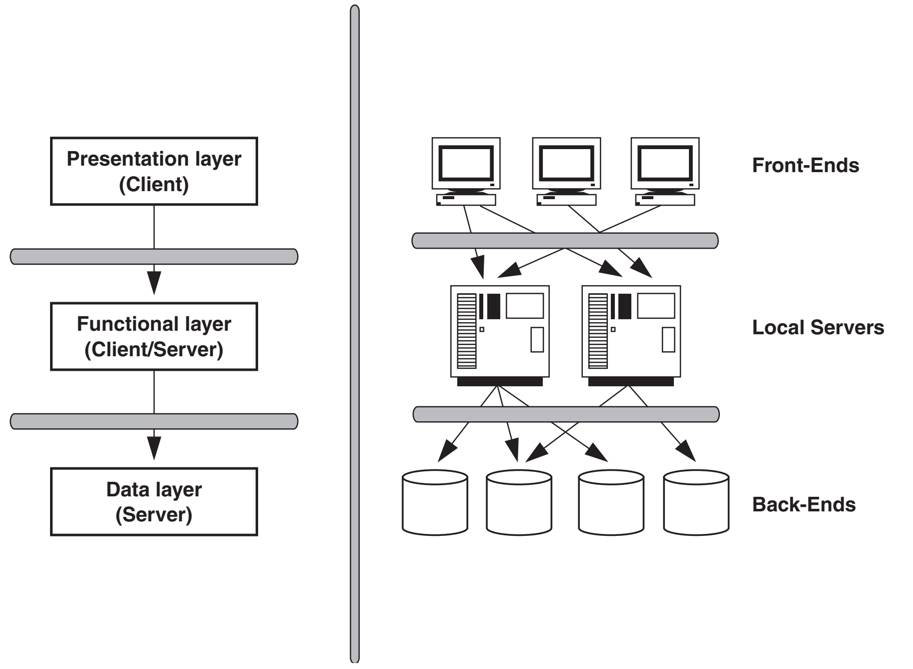


FIGURE 9.5

A specialized three-tier architecture.

generalization and specialization. The microstructures of their elements are classes or interfaces. The classes or interfaces of a layer may use only the classes or interfaces of their own or lower layers. The inheritance relationship is valid within one layer. Elements of one layer may also inherit form elements from lower layers.

The following sections will explain why it is reasonable for object-oriented layers not to limit inheritance and use to the next lower layer.

DISCUSSION

If we need to implement the interaction of elements of a layer to its next higher layer, then we have to use an abstract interface, as in the case of the observer mechanism (see Section 8.6). The concepts of the observer mechanism (e.g., the `Observer` or `Observed` interfaces) have to be located within one layer. Only the objects that implement this pattern at runtime can then belong to different layers. Since they are known only by the type of their abstract mechanism, there will be no undesired dependence between layers.

A protocol-based layer architecture does not make full use of object orientation. We will discusss in the next section how we can transfer the generalization and specialization concepts to an object-oriented layer architecture, and how this is compatible with a protocol-based layer architecture.

We know that generalization and specialization are expressed by the inheritance relationship in the object-oriented world. This is obvious for the classes within one layer. But what does it mean for superclasses and their subclasses in different layers?

EXAMPLE

Bank example

Figure 9.6 shows a simplified example of an object-oriented layer architecture with a product domain, investment business, and a banking business domain as well as three classes. We know from our discussion in Section 9.2.3 that the business domain contains the core abstractions of an organization, which are specialized in different product domains and then represented in the respective use contexts. If we maintain the arrangement of the protocol-based layer architecture in an object-oriented layer architecture, then the business domain forms the basis underneath the respective product domain, while the use contexts are above it. In the example shown in Figure 9.6, the `Account` class resides in the business domain. In the product domain layer, the classes `InvestmentAccount` and `SavingsAccount` are derived from `Account`. This is conceptually in line with our previous discussion, except for the representation: the superclasses and subclasses are upside down.

Notice the result: It is sensible to allow inheritance across layers in an object-oriented layer architecture. The concepts of a higher layer, closer to the application, specialize the generic concepts of a lower layer.

BACKGROUND: THE OPEN-CLOSED PRINCIPLE AND THE LAYER ARCHITECTURE

If we look at the open-closed principle discussed in Section 2.2.3, we can further justify the use of inheritance between the layers of an architecture. Applied to an object-oriented layer architecture, this principle means that a layer with its elements should be open for extensions but closed for modification.

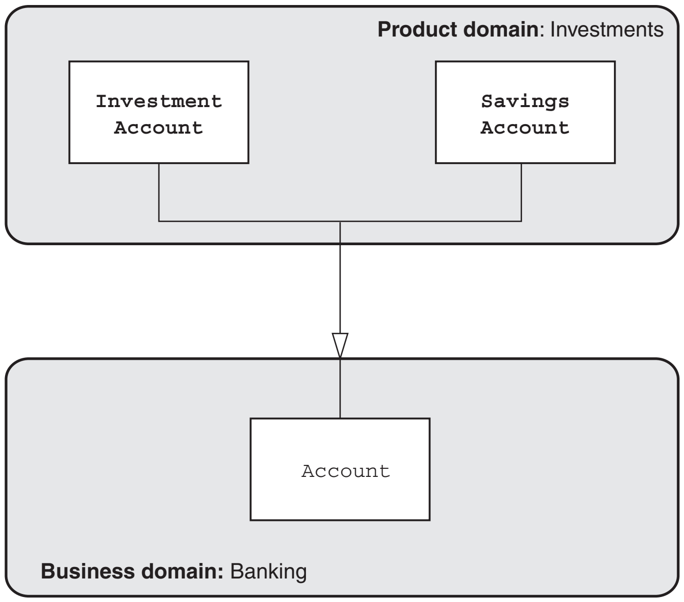


FIGURE 9.6
Layered architecture and inheritance.

Inheritance between layers

This requirement can be met in an object-oriented layer architecture if we allow inheritance between layers. This means that we can provide a generic concept, such as `Account` in the business domain, in the form of a "closed" class. Its required specializations are then moved to the higher product domain layer by inheritance.

If we implement these generalization and specialization relationships between the classes of different layers, using the program language means, that is, abstract class, inheritance, polymorphism, and dynamic binding, then we can observe an interesting effect: Objects that are instances of classes from a *higher* layer are used by objects of a *lower* layer. The superclass in a lower layer defines a type in our object metamodel (see Chapter 2). At runtime, polymorphic objects of a subtype, or a subclass, can then be assigned to variables of this type. This subclass can be an element of a higher layer. Operations called on objects are defined in the superclass. As long as the inheritance hierarchy represents a type hierarchy, this usage of the open-closed principle is straightforward and trouble-free.

In summary, let's look at a few important points relating to the open-closed principle of an object-oriented layer architecture. Inheritance, polymorphism, and dynamic binding bridge the contradiction between open and closed characteristics of construction units. This means that layered architectures with permissible inheritance between layers allow us to implement required extensions for one layer in higher layers.

EXAMPLE

Bank example

Figure 9.7 uses a banking example to show the open-closed principle. In this example, within the business domain a class, `Customer`, manages a list of accounts of that customer. These accounts are of a generalized type, `Account`, which was implemented as an abstract class. A product domain defines two subtypes, `InvestmentAccount`

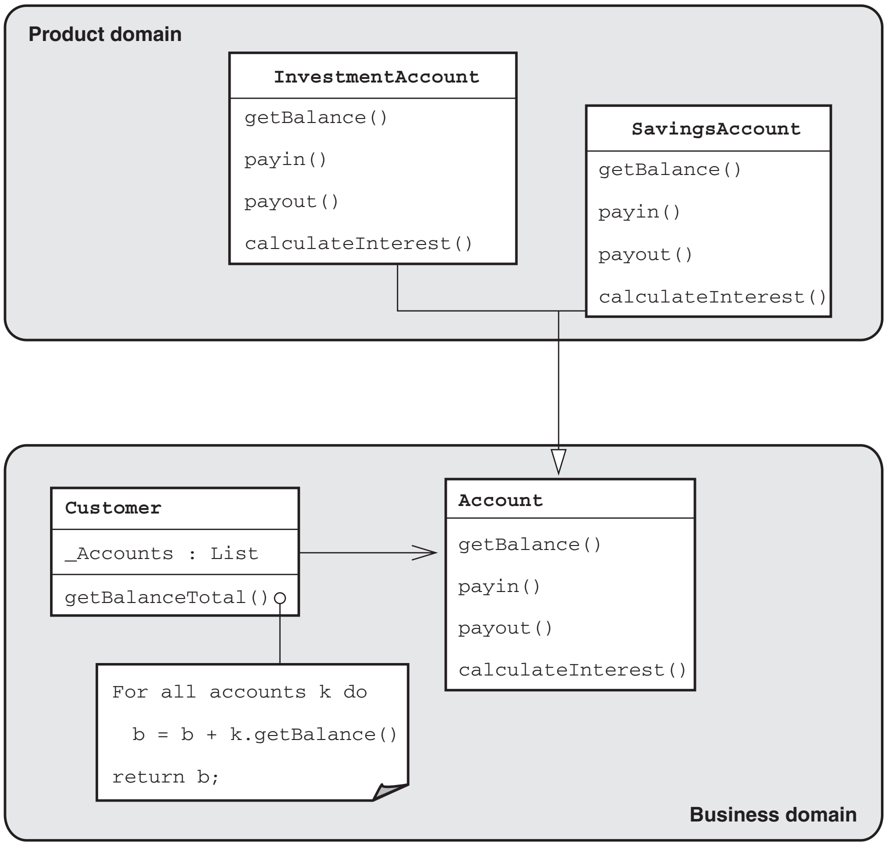


FIGURE 9.7 A bank example showing the open-closed principle.

and SavingsAccount. Both accounts are subclasses of the Account class. The operation, getBalanceTotal(), of the Customer class is implemented by calling the operation getBalance on each account object contained in the list and summating the result.

The inheritance relationship between the class Account and the two classes, InvestmentAccount and SavingsAccount, forms a type hierarchy. The reason is that an object of the type InvestmentAccount or SavingsAccount can be used in all places where an object of the type Account is expected in the class Customer.

The object diagram in Figure 9.8 shows that the Customer object resides in the business domain. It uses a list to access Account objects that were created in the product domain. The semantics of the customer operation getBalance Total() is defined by the implementation of the deferred Account operation, getBalance(), in the subclasses of the product domain.

DISCUSSION: IMPLEMENTING THE OPEN-CLOSED PRINCIPLE WITH LAYERS

The example in Figure 9.8 shows that we can use inheritance across layers to implement a "closed" layer and concurrently hold it "open" for extensions in higher layers.

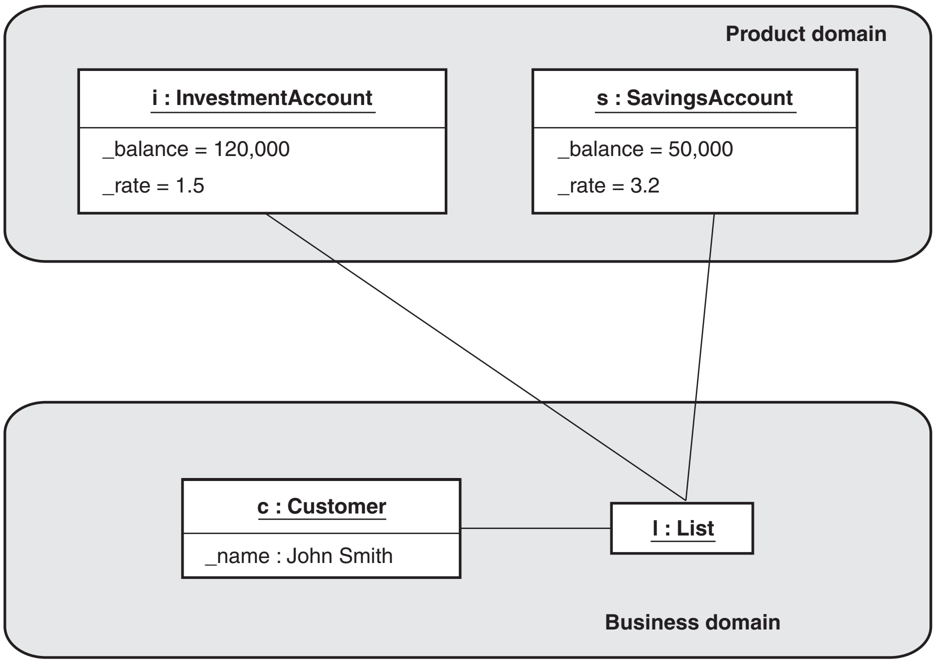


FIGURE 9.8
Object diagram for the customer and account classes.

Inheritance, polymorphism, and dynamic binding represent only one technical means to implement the open-closed principle. For example, changes and extensions can also be implemented by the use of a callback method. Another technique uses reflexive architectures that let you modify construction units at runtime, similarly to the Smalltalk system. Yet another important technique to implement extensions is the role concept introduced in Section 9.4. From a conceptual point of view, the role concept allows us to change the type of application-specific objects at runtime (although this feature is implemented based on inheritance and polymorphism).

9.3.5 The Layer Concept of the T&M Model Architecture

The previous sections clearly showed that protocol-based and object-oriented layer concepts each have respective strengths, so it makes sense to combine them in the T&M model architecture. For this purpose, we use the layer concept as follows:

Using the layer concept

- A layer aggregates the software elements of a model architecture as a design and construction unit, which is coherent from a domain-related and a technical viewpoint. A layer itself has no interface and no relationship to other layers; interfaces and relationships are provided by the included elements only.
- The layers of a model architecture are built hierarchically. Depending on the layer visibility and usage, we speak of a protocol-based or object-oriented layer. We use class libraries, components, and frameworks as elements within a layer. These elements are linked by use relationships or inheritance, where the connector structure is governed by design patterns.

Discussion: Comparing the Two Layer Concepts

Neighboring layers of an object-oriented architecture are built by the generalization principle. For example, the business domain layer contains common concepts, which can be specialized in the product domains. This kind of relation is rather rare between the layers of a protocol-based architecture. The presentation layer of the three-tier architecture uses objects of the functional layer to represent them. This is not a form of specialization.

Transparency of layers

Another important question relates to the *transparency* of layers. The use of protocol-based layers is opaque. More specifically, they show a protocol but hide the underlying layers, because they abstract from their implementation. In contrast, object-oriented layers follow the generalization and specialization principles, which means that they can't completely hide the underlying layers. Only if lower layers remain visible for higher layers are the developers able to utilize existing concepts, either directly in the form of classes that can be instantiated, or as an extension point for a specialized implementation.

In this respect, it is rather difficult, both from the conceptual and constructive viewpoints, to unify both layer types in one layer. In some cases, this problem can be solved by introducing a separate protocol-based layer on top of an object-oriented layer. This new layer allows access to a lower layer, and at the same time ensures proper decoupling.

Example: Direct Combination of Layers

Bank example

In the example shown in Figure 9.9, a class, `CustomerEditor`, of the product domain uses the generic class, `RelationalDatabase`, which is available from a database library in the system base layer, to implement a service, `storeCustomer()`. Note that, between the product domain layer and the system base layer, only protocols are used, and there is no generalization relationship. The class `CustomerEditor` includes a reference pointing to an abstract class, `Customer`, of the business domain. Polymorphism can be used to bind an object of the `CorporateCustomer` class to this reference.

One major drawback of this simple solution is obvious: A layer representing domain concepts has direct access to a technical base layer. Each change to the protocol of this base will have an impact on the conceptual layer. Object-oriented techniques offer a more elegant solution, compared to the direct use in this example. For example, the *bridge* pattern proposed by Erich Gamma et al. is suitable to separate a technological concept from its technical implementation. This opens up a way to let a conceptual domain layer, such as the product domain layer, to access a conceptual software layer, such as, the technology domain, while deferring the actual implementation of the technological concepts to the system base layer. In effect, the system base behaves like a protocol-based layer towards the technology domain, although it uses object-oriented techniques.

Unfortunately, it is often difficult to clearly separate these two concepts in real-world applications. This is due to the way modern class libraries or technical frameworks are implemented. For example, many persistence frameworks require a common interface, such as `Persistent`. All classes to be saved to a database normally have to implement this interface. With a simple coupling of the persistence framework, this interface would "show through," and application-specific classes would have to implement

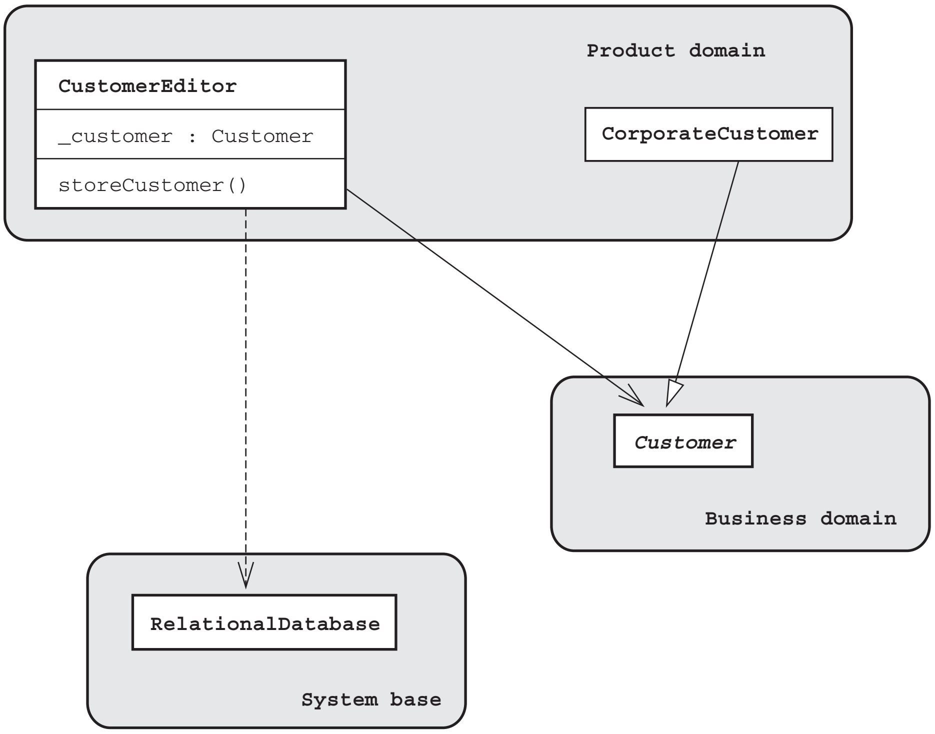


FIGURE 9.9
An object-oriented layer, or the product domain layer, uses a protocol-based layer, or the system base layer.

an interface of the system base layer. There are several solutions to solve this problem, including the use of other design patterns, such as *adapter*. In summary, we have to find pragmatic compromises, but this does not affect the basic distinction between the two layer types for a model architecture.

EXAMPLE: USING THE BRIDGE PATTERN TO COMBINE LAYERS

Bank example

The example discussed in this section is based on the banking example shown in Figure 9.9. However, we have modified it, as shown in Figure 9.10, so that the class `CustomerEditor` now has access to the abstract concept and protocol of the class `Database` in the technology domain. Our new example uses a *bridge* class to connect the `CustomerEditor` class to the root class, `RelationalDatabase`, in an implementation hierarchy, thus decoupling the tool from the concrete object store.

9.3.6 The Three-Tier Architecture

The previous sections discussed several important characteristics of the three-tier architecture. We can conclude that many aims related to this architecture cannot be fulfilled. Instead, we propose a multitier architecture.

One important argument in favor of the three-tier architecture is that it separates domains, so that each layer handles and encapsulates a coherent set of related concepts. The logical consequence is the maximum independence of the implementations and optimum support for their modification. For example, popular network protocols

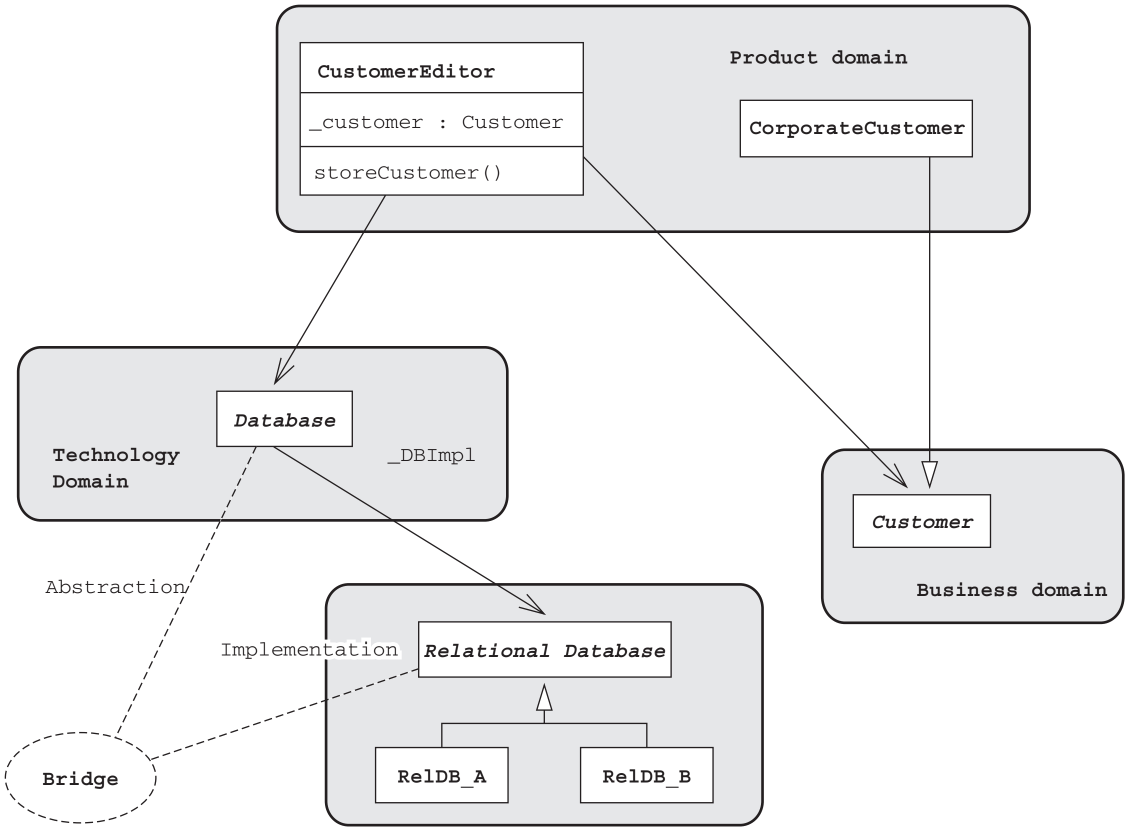


FIGURE 9.10 Using a *bridge* pattern to *decouple* a protocol from its implementation.

use a layer architecture to provide several abstraction levels, ranging from bit block transport up to application-specific services. On the other hand, there is no similar set of stepwise abstractions in the three-tier architecture.

Drawbacks of the three-tier architecture

Though many argue to the contrary, the three-tier architecture does not meet the requirement that changes should remain local to their layers. For example, changes to the user interface normally extend to the domain functionality and the data objects. The reason is that newly added information leads to new attributes in data objects. Domain changes to the handling of the system, such as items represented in menu options, require extensions to the functional layer. In turn, changes to data objects mean that the representation has to be changed, as when additional attributes have to be represented at the user interface. This shows that the layers are not really independent of one another in either direction, and they cannot be developed separately.

Consequently, the three layers do not decompose an application system into independent models or macrostructures. Nevertheless, a common approach in this architecture is to develop the user interface, the functionality, and the data objects separately. This often leads to application systems that map the domain functionality as pure data objects in software, where large parts of the domain dynamics are implicitly implemented in the GUI component. This type of applications is hard to understand and makes further development difficult.

Though a three-tier architecture does not exclude inheritance between layers, this is normally not possible due to a lack of "is-a" relationships (see Section 2.1.16). For example, it is not reasonable to subclass functional layer classes from data layer classes, and the situation is similar between the presentation and the functional layers.

In terms of its character, the three-tier architecture is a protocol-based layer architecture. One major problem is related to mixing three contexts, that is, application domain modeling, handling and presentation, and technical realization. As mentioned in Section 9.2.4, we want to separate these contexts in our model architecture to ensure their independent development. In summary, we have seen that the popular three-tier architecture is not suitable to form the basis for a generic T&M model architecture. However, we still consider it very suitable as a process architecture.

9.3.7 The T&M Model Architecture

We propose a model architecture that takes the three dimensions of an application system into account. The design of suitable layers allows us to introduce a separation of concerns early on at the logical level. The layers are arranged so that changes in one dimensions will have little or no impact on the other dimensions.

Background: The T&M Model Architecture

Section 6.8 introduced the three dimensions proposed in the T&M model architecture that determine the contexts of an application system. When we orient a model architecture along the lines of these contexts, then this primarily means that we encapsulate each dimension in a separate layer or at least in a distinct hierarchy of layers. Skillful selection of the dependencies between the layers of different dimensions ensures that we prevent changes in one dimension to affect the layers of another dimension.

Figure 9.11 shows that many important design decisions and parameters can be allocated to one of these dimensions. For example, the workplace type influences the way we design the handling and presentation of a system, as well as the technological decision for a specific front-end system.

Problem

We want to define a layered architecture that allows us to encapsulate the set of software development contexts. Next, we have to arrange the layers so that they allow maximum potential for changes to each dimension of these contexts. In addition, such a model architecture should scale to the size of the application system and the technologies used.

Solution

We propose implementing a model architecture for interactive application software based on the U model shown in Figure 9.12. The U is formed by the system base, the technology, and the handling and presentation. These are called the generic or technical layers. They embed the domain core of a specific application domain.

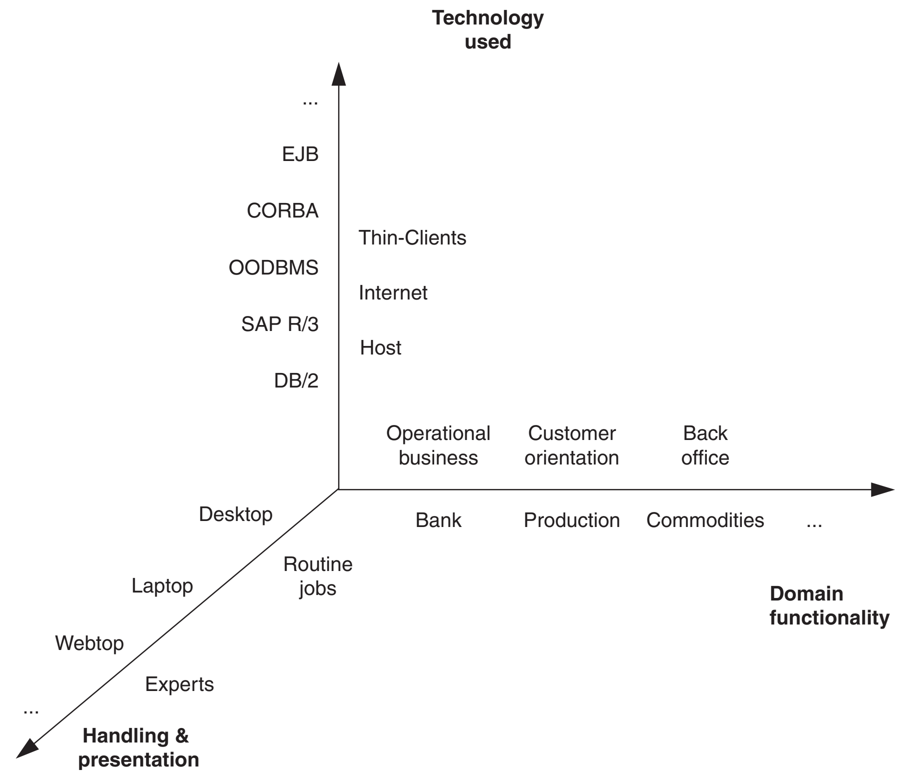


FIGURE 9.11 Software development contexts in relation to the model architecture.

DISCUSSION

Criteria for a model architecture

The requirements and principles for the construction of a model architecture as discussed in the previous sections can be evaluated as follows:

- The technical foundation used to build a software system decomposes into a system base and the technology used. In addition, we may need to supply auxiliary implementations to substitute necessary but missing programming language features.
- Handling and presentation determine how an application can be used. In our approach, they conform to a guiding metaphor and its design metaphors.
- The structures of the software model should be oriented to the application domain structures. This means that the concept model of the application domain has to be mapped to the software model. In addition to these microelements, the application-specific macrostructures (see Section 9.2.4) should be reflected in the layers of our model architecture.

These points characterize the basic structure of our model architecture. Note that this is a static, logical structure, which should be present at least when you are developing families of application systems based on the T&M approach.

ELEMENTS OF THE T&M MODEL ARCHITECTURE

Our model architecture is built as a layered architecture, where protocol-based and object-oriented layers ideally should be separated.

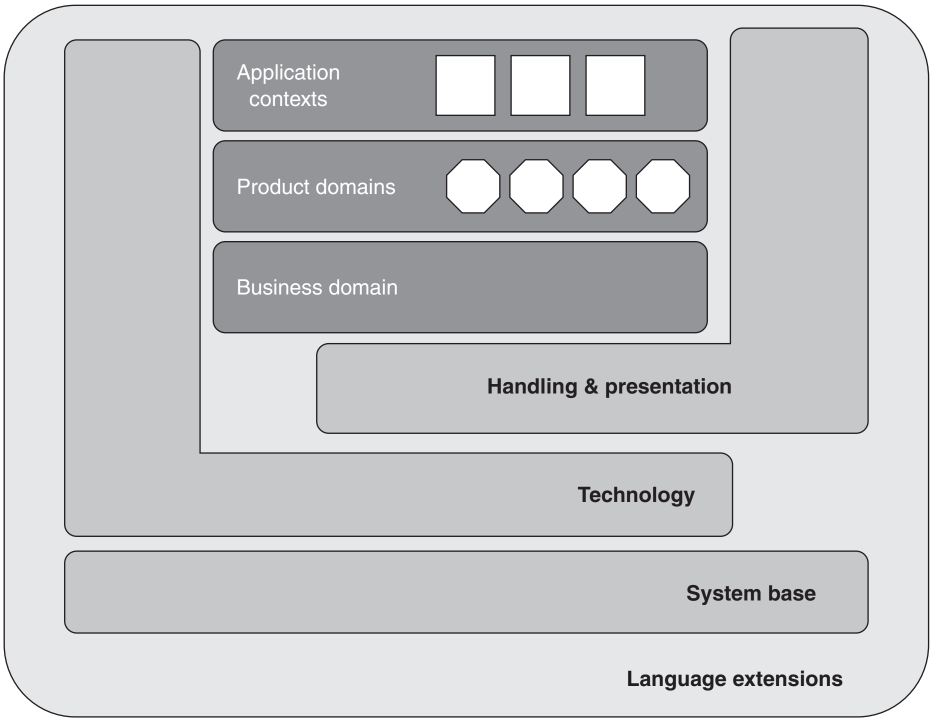


FIGURE 9.12
A U model for the architecture of interactive application software.

Protocol-based layers encapsulate the implementation of services provided for higher layers. If the protocol-based layers are hierarchical, then the protocols of the higher layers are closer to the application domain and abstract from the implementation of services in the lower layers.

Object-oriented layers are connected by inheritance and use relationships. Again, the higher layers are closer to the application domain than the lower ones, but in the sense that they extend lower-layer concepts. Inheritance is used only within one layer and from a lower to a higher layer. This means that inheritance can extend over more than one layer boundary or skip a layer. The reverse is true for a use relationship, which, if it spans layers, is always directed from a higher to a lower layer.

LANGUAGE EXTENSIONS

Language extensions are actually not a layer. They provide interfaces and implementations that we would like to have in a programming language for all layers of our model:

- In programming languages with weak metalanguage properties, such as C++, we would like to have generic elements for object-oriented construction, such as reflection, a garbage collector, or serialization mechanisms to transform object structures into flat structures, and vice versa.
- Support for the implementation of domain value types (which should behave as far as possible like primitive types and not like object types.
- Support for the design-by-contract programming approach (see Section 2.3).

Note that the elements of language extensions are totally independent of the specific application domain. What we implement, and how we do it, depends merely on the programming language we use. This means that we can assume that the corresponding interfaces will be pretty robust. Language extensions will be used by all other layers through use and inheritance.

The following sections describe the actual layers of the T&M model architecture in more detail.

The System Base Layer

Section 6.7 already introduced the system base layer on a conceptual level. This layer includes all interfaces to existing technical and legacy systems. The systems themselves are encapsulated in black-box frameworks and class libraries, that is, they are hidden from the higher layers. The system base layer includes (among others) the following services: encapsulation of the operating system, of the used middleware, and of persistency providers, such as relational or mainframe databases or electronic archives.

The elements of this layer are completely independent of a specific application domain, which means that they can be used to build interactive application systems for arbitrary domains. Since the elements of this layer normally encapsulate third-party products and standard software, we usually have to expect changes. Accordingly, this layer has to be implemented very carefully, keeping the set of protocols as generic as possible to ensure that a potential replacement of construction units will not have a negative impact.

The construction units of this layer are used by all higher layers, particularly by the technology layer, which will be described in the next subsection.

The size of the system base layer depends on the programming language we use and the available standard libraries and frameworks. For example, in Java-based systems the system base layer is typically small or completely omitted, because most required abstractions are already provided as standard libraries.

The Technology Layer

The technology layer represents the concepts described in Section 6.7, that is, it groups all models of the technologies we use. The relationship between this layer and the system base layer is characterized by the *bridge* pattern we commonly use. While the interfaces of implementation units reside in the system base layer, the technology layer groups the more general technological concepts. The technology layer consists mainly of white-box frameworks, which can be completed by extensible black-box frameworks to support turnkey standard solutions:

- A framework to store and load objects in and from a persistence medium, such as a generic data repository concept.
- A framework to communicate with other environments or processes, such as, a generic communication concept.
- Interaction forms to link tools with the window system.

This layer can also be reused for different specific applications. However, it only provides concepts required for the concrete system under development.

The Handling and Presentation Layer

This layer accommodates both concepts and concrete objects, and it objectifies the underlying guiding metaphor and its design metaphors. This means that this layer is closely related to the T&M approach. Since tools, materials, automatons, and other metaphors realized in this layer determine the user interaction with the application software, the representation of these basic elements in a separate layer is useful. For example:

- The selected environment concept with electronic desktop or other spatial metaphors.
- Collections (containers, folders, and stacks) to organize workplaces.
- Default implementations of design patterns for tools, materials, automatons, and service providers.
- Support for the connection of Web applications to service providers.
- Implementations of generic tool components, such as listers.

This layer is not yet oriented to a specific application domain, so that it can be used for all systems that correspond to the selected guiding metaphor and its design metaphors. Naturally, if you develop a totally different application type, e.g., a data warehouse, you may have to replace this layer, for example, by a database-specific layer.

Implementing the handling and presentation layer

The handling and presentation layer accesses the technology layer and may access parts of the system base layer. It consists mainly of white-box frameworks, complemented by default implementations of tools, and other elements. This means that, by our definition, it is an object-oriented layer.

The Business Domain Layer

Implementing the business domain layer

The business domain layer groups the fundamental abstractions of an application domain, to the extent that it is structured on the basis of the principles described in Sections 9.3.3 and 9.3.4. This means that the core abstractions from all product domains converge in this layer. Obviously, these generalizations can only be built as white-box frameworks. In addition, there are normally a few objects used generally in the application domain, so that they can be implemented in this layer. For example, this could be the class `CheckingAccount` in a bank. Other candidates for this layer are specific domain value types, which are implemented based on language extensions. These common generic classes and domain value types are normally built as part of open black-box frameworks.

The business domain layer itself builds on the technical layers of the U model. We should attempt to make minimal use of the system base layer, while using as many services as possible through encapsulation in the technology layer. Also, the business domain layer contains only a few handling and presentation aspects, so that their references to this layer are limited.

Obviously, the business domain layer is closely connected to an application domain. So the question is whether or not this business domain layer can be developed only for a particular company, or for an entire industry. Naturally, there are concepts and notions that generally characterize the business of each industry. Otherwise, companies in specific industries, such as banking, could not cooperate or merge. On the other hand, the core business of each business within an industry differs so that a business domain cannot simply be transferred. This is not a new insight, because people have had to deal with this fact in the development of corporate or industry-specific standard software. For example, it has been observed in the banking industry that cross-industry models were too abstract to be useful.

For this reason, we suggest checking each individual case for which of the domain concepts of a business domain are valid beyond the company for which that business domain was modeled. Once we have a clear answer to this question, we can identify the parts of the business domain layer that can be reused.

The Product Domain Layer

Section 9.2.2 explained that we model a separate product domain for each application-specific unit formed according to the object principle. All these product domains are implemented in the product domain layer as separate modeling units. This means that the concepts and objects in a product domain should ideally be implemented so that they do not overlap to ensure that they have no references to other product domains.

Implementing the product domain layer

The elements of the product domains in this layer are mainly black-box frameworks, built on the basis of the business domain's white-box frameworks. Also, they use the technology layer and to some extent the system base layer. In this respect, we have to take care that all specializations and extensions take the open-closed principle into account. Frameworks of a product domain often specialize several white-box frameworks from other domains. We only develop separate white-box frameworks for the product domain layer if they can be reused to build other frameworks for the same product domain. The use context layer described in the following section uses exclusively black-box frameworks from different product domains.

Note that no frameworks from neighboring product domains should be used. This applies both to the use and inheritance. A product domain is more strongly tailored to the specific application domain than the business domain. After all, it maps the fundamental principles of a company's business to specific products and processes.

A product domain is the place where the differences between the different companies of an industry manifest. Therefore, we do not expect that this layer can be reused in other corporations.

The Use Context Layer

Implementing the use context layer

The use context layer was conceptually described in Section 9.2.1 it is composed of different modeling units. The elements of a use context must not have references to other contexts, as is the case for product domains.

Application systems are built on the basis of the black-box frameworks developed in product domains. Additional frameworks are developed only if special workplace types require particular technological or domain aspects, for example, if touch screens are used for self-service terminals.

Rules to Build a Layered Architecture

Relationship between elements

The T&M model architecture defines on a logical level technological and domain software layers, where each layer consists of defined elements and their connectors. The layers themselves are not interconnected, only their elements, so we cannot specify rules relating to the layers. However, we can limit the relationships between elements (i.e., class libraries, components, black-box and white-box frameworks) of the layers by defining the following dependence rules:

- Elements of one layer may use only elements of the same or the next lower protocol-based layer.
- Elements of one layer may use only elements of the same or an arbitrary lower object-oriented layer.

- Elements of a product domain or use context should not use elements of another product domain or use context.

If we develop the frameworks of different product domains based on the system base, technology, handling and presentation, and business domain layers, then we can ensure that they can be easily combined and integrated in the use contexts of our application system. An additional benefit of this approach is that it promotes a uniform appearance. The system base and the technology layers contribute to better technical coherence, while the business domain ensures a coherent domain model. In addition, a uniform look and feel for our application system is ensured by the handling and presentation layer.

EXAMPLE: A LAYERED ARCHITECTURE FOR THE BANKING INDUSTRY

Bank example

The example described in this section shows how we can develop a layered architecture for a bank based on the T&M model architecture, which was developed in a similar form in the so-called GEBOS projects which designed and implemented a workplace application system for a German banking group. We describe how the existing system fits into our model architecture as discussed in the previous sections. Figure 9.13 shows this layered architecture.

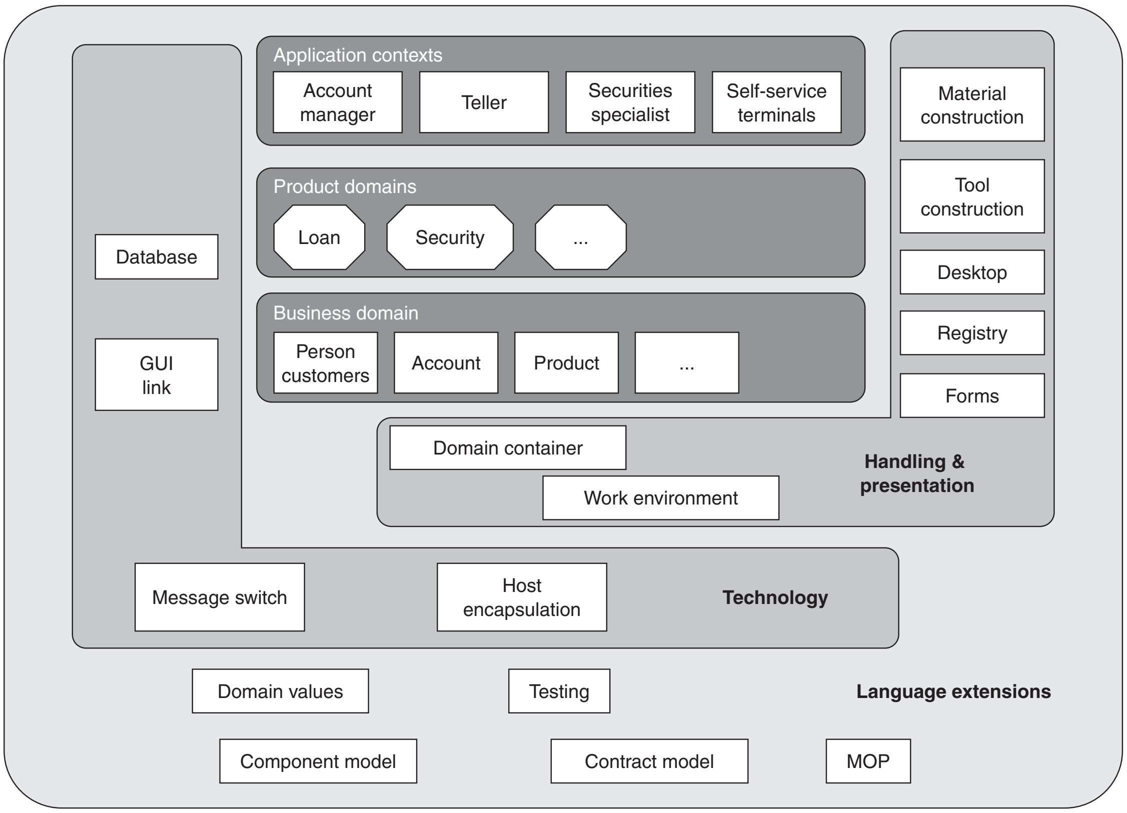


FIGURE 9.13 Example of a layered architecture for the banking.

The white-box framework for tools construction in the handling and presentation layer ensures that all tools created on that basis can be added to an electronic desktop in our application system. Similarly, the database framework in the technology layer ensures consistent storage of objects in a relational database. This means that each application system can access objects stored in this database.

The two white-box frameworks, `Person/Customer` and `Account`, from the business domain form the domain basis for the construction of the `Credit` framework. The white-box frameworks in the business domain layer depend on the existing frameworks in the technology and handling and presentation layers. This layering of frameworks facilitates the technical combination and integration of frameworks in the product domain layer, which are based exclusively on frameworks of the business domain layer. Frameworks in the business domain layer can embed domain concepts in a common technical basis, so that their uniform technical use is guaranteed.

All frameworks of the business domain and product domain layers define their own domain materials and tools, which are linked to the tools construction framework following the *aspect* design pattern.

The frameworks in the technology layer form the technical basis for the product domains and the use contexts, while the business domain layer forms the domain basis. Both ensure that a family of application systems can be assembled by combination of elements from (possibly several) product domains.

TECHNICAL IMPLEMENTATION OF A MODEL ARCHITECTURE

The previous sections explains the domain and technical design of frameworks from the elements of a T&M model architecture. Another aspect of this approach is how we can partition and assemble an application system for deployment in the different use contexts at hand. In this respect, we have to consider several points. The system has to be shipped in different configurations, and each version should be minimal, that is it shouldn't include unnecessary parts. The layered architecture can help us solve this problem, because it makes clear which parts depend on which other parts.

The directories and program packages should be organized so that the organizational principle of the layered architecture becomes clear. For this purpose, we normally use an appropriately structured directory, and each layer is mapped to a separate subdirectory. In turn, our use contexts and product domains are subdirectories of the product domain or use context directory. Also, our frameworks are substructures within their layer directories.

Java lets us use packages to elegantly implement this structural similarity. Each layer is a separate package, and the components and frameworks contained in it are subpackages. Figure 9.14 shows how we can arrange directories for layers and frameworks.

9.4 DESIGN PATTERNS FOR THE T&M MODEL ARCHITECTURE

We often have to deal with the following two problems when trying to loosely couple elements in a framework-based architecture:

- We cannot directly implement the role concept by generalization and specialization and by inheritance.

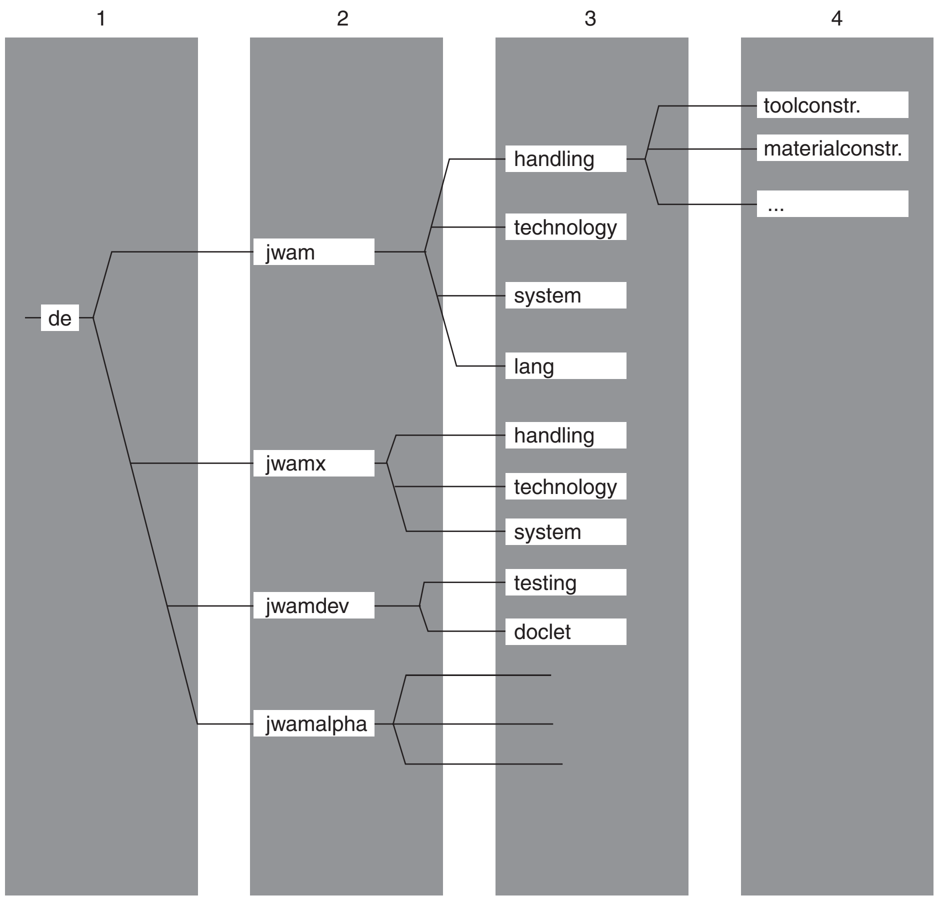


FIGURE 9.14

Directories for layers and framework structures.

- In many cases, we can achieve loose coupling between elements by using an abstract (super)class. But there is still one place where the specific class has to be known, namely the point where the object is created.

On this foundation, we introduce two design patterns, which can help solve the above problems:

- The *role pattern* offers a way to handle a domain object in different roles without violating its domain identity.
- The *trader pattern* is a creator pattern that moves the knowledge of a specific class to be created from the actual application to a separate one.

9.4.1 The Role Pattern

Intent

You can use the role pattern to adapt an object dynamically to special client requirements. In the context of the T&M model architecture, this means that attributes and interfaces can be dynamically added to or removed from objects instantiated by classes of the business domain, depending on a client's view.

Problem

Sometimes we need to model a domain-motivated entity that shows different characteristics (roles) in changing contexts but is still regarded as one identity.

Two objects with different technical identities can represent one identity from the domain-specific view. Within the T&M model architecture this applies particularly to objects of the application domain, which are described as concepts in the business domain but which are seen with different interactions and domain states in different product domains. In this case, we have to ensure that an object that moves from one product domain to another product domain shows these different interactions and states, while keeping the same domain identity

We always model an application system under a specific view. This means that classes, representing objects or concepts of the application domain, are characterized by the view we have when developing the application domain. This is particularly true for different product domains (see Section 9.2.2). This differentiated view won't cause any problems as long as we model different objects of the application domain from a view of the respective product domains. However, if a concept of the business domain has different attributes and interactions depending on the different product domains' views, then we need solutions that help us prevent having to model all views within this single concept. Otherwise, the concept as part of the business domain would have dependencies on the product domains. Obviously, such dependencies lead to one common domain that includes all product domains and the business domain.

Example

Bank example

We use another example from the banking business to show that the same domain object is modeled differently for different product domains. Assume that we have two concepts, *`borrower`* (loans product domain) and *`investor`* (investments product domain), which are different specific views of the general concept *`customer`*. Let's further assume that a *`borrower`* manages securities transferred to the bank for collateral. This type of information is insignificant for the *`investor`* concept, so we won't model it for this concept. In addition, properties modeled from the view of one product domain often represent dynamic aspects of a domain object. For example, a customer becomes a borrower of a bank when he or she takes out a loan. Otherwise, that customer will not have the properties related to a borrower.

Discussion

In this example, a naive solution based on object-oriented means would model the three notions *`customer`*, *`borrower`*, and *`investor`* as classes. The *`customer`* concept would be part of the business domain, while the *`borrower`* concept would be part of the loan product domain, and the *`investor`* concept would be an element of the investments product domain. Since the *`investor`* and *`borrower`* concepts extend the *`customer`* concept, and *`investor`* or *`borrower`* objects can be used instead of a *`customer`* object (a borrower "is a" customer), we could implement the `Customer` class as a generalization of the two classes, `Borrower` and `Investor`. The class `Customer` would then model properties like a customer number or a `getBalanceTotal()` operation (see Figure 9.15).

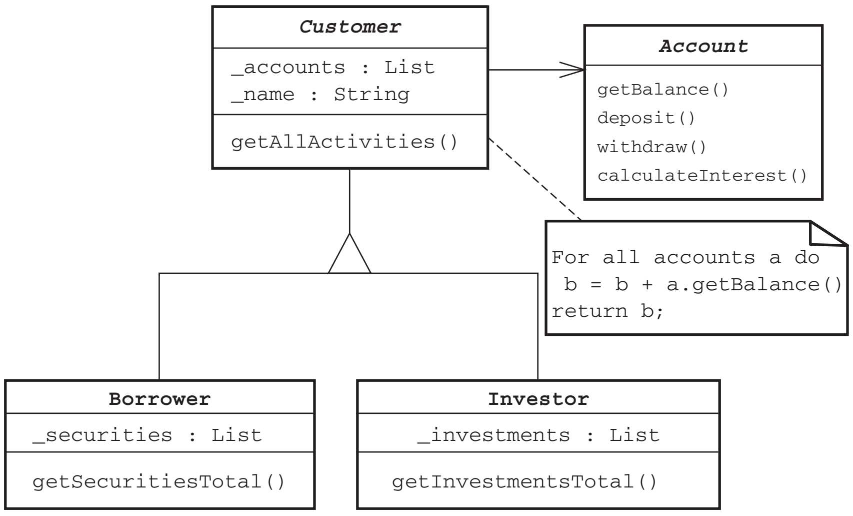


FIGURE 9.15
Using inheritance to specialize a customer for a product domain.

For example, if a customer, say John Smith, is managed as an investor and borrower (i.e., an object from both classes is created) at the system's runtime, then the implementation shown in this example duplicates the attributes modeled in the `Customer` class. The reason is that the object of the `Investor` class and the object of the `Borrower` class both have common attributes, that is, `_accounts` and `_name`. This means that we need expensive synchronization mechanisms to maintain the domain identify of these objects. Changes to one object have to be propagated to the other object; in other words we have to use value equality to simulate their domain identity.

Technical implementation in the form of multiple inheritance (if this is possible), which forms a subclass, `BorrowerAndInvestor`, of the two classes, `Investor` and `Borrower`, prevents duplicating attributes of the `Customer` class, but it also leads to an increasing number of subclasses. This flood of subclasses will quickly make the design unclear and hard to understand. In addition, all product domains use the combined classes (e.g., the loans and investments product domains both use the class `BorrowerAndInvestor`). Another problem of this solution relates to object migration. For example, a customer is not necessarily a borrower and an investor at the same time. He or she may initially be managed as an investor. This customer may additionally become a borrower when he or she takes out a loan from the bank. When this happens, a new object of the class `BorrowerAndInvestor` has to be created with the attributes of the investor object. In addition, all references pointing to the "old" investor object have to be set to point to the new object. Unless we have suitable language support (e.g., `become:` in Smalltalk), this process of changing references is difficult; also, bear in mind that some languages, such as Java, do not support multiple inheritance.

SOLUTION

We will model the different views of a domain object in separate objects, or so-called *role objects*. These role objects can be dynamically added to and removed from a *core object*. A core object shows the characteristics of a domain object common to all contexts or domains. We ensure that the core and role objects share the same domain identity.

BACKGROUND

The literature (e.g., Gottlob et al. and Reenskaug) uses the term *role* to discuss the problem inherent in modeling domain objects dependent on the respective view (product domain) and the time when they are viewed. Kristensen and Østerbye define the term *role* as the set of properties of an object in its behavior toward a set of other objects.

A tool used to edit customer securities works with customer objects under the special view (role) of a *borrower*.

A core object represents the characteristics of a domain object modeled in the business domain, that is, characteristics which are independent of the product domains.

To better understand these concepts, let's look at a few important definitions:

A *role object* represents the set of attributes and operations that an object has in an application-specific view.

A *core object* is an object that can play different roles in one system. It can include states and operations that, regardless of a specific view, are represented externally by all of its role objects. A core object can exist independently of its role objects.

For example, the core object for a customer includes the address, a customer number, and a checking account. This core object and its role objects are designed on the basis of the information hiding principle. All attributes are manipulated exclusively by operation calls. This means that a role object has no direct access to the attributes of its core object. Since a core object has to be able to live independently of its role objects, it must not use operation calls to access them. Otherwise, dependencies would form between the core object and its role objects, and, in turn, to dependencies between the product domains.

To implement role objects and core objects we use classes. To be able to relate classes to the different kinds of objects, we call a class that models a core object a *core concept* and one that models a role object the *role concept*. Figure 9.16 shows an example of the objects involved.

Note that the technical distinction between a core object and a role object allows us to differentiate our notion of identity. The core object has its own technical identity, which is different from the technical identities of the role objects. At the same time, together with its role objects, the core object forms a logical unit with a domain identity. For example, if we have a core object, `customer`, with two role objects, `borrower` and `investor`, then all three objects have both their own technical identity and a common domain identity, namely that of the logical unit, consisting of the core object, `customer`, and its role objects (see Figure 9.16). Following Kristensen and Østerbye, we call this logical unit *subject*.[2]

A *subject* is the logical unit, consisting of a core object and all current roles of this core object. It represents a domain identity, also called a subject identity.

2. Kristensen and Østerbye distinguish further between a *subject* and a *closured subject*. A subject includes the core object plus an arbitrary number of role objects, while a closured object represents the core object plus all its role objects. This means that the term *subject* as we use it is identical to the definition of *closured subject* by Kristensen and Østerbye. We did not need a further differentiation in our application domains.

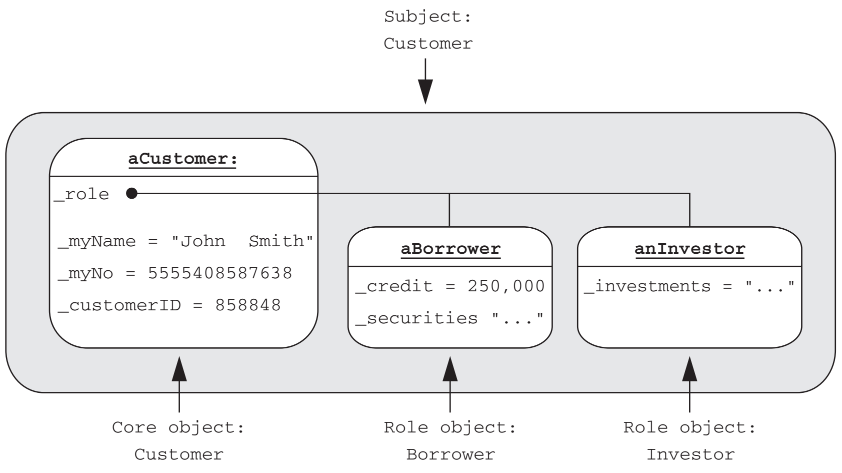


FIGURE 9.16

A core object, customer, with two role objects, borrower and investor.

Pattern Structure

Figure 9.17 shows the structure of the *role* pattern. The specification (`Spec`) used in this figure is similar to the specification of the *trader* pattern, which will be described in Section 9.4.2

Participants

`Subject` (the subject *customer*):

- Models the subject in the form of domain operations, for example, `changeAddress or getPhoneNumber`. The operations are defined as abstract operations and have to be implemented in the core concept. Since role objects must be used polymorphically instead of the core object, the role concepts also implement the subject's interface.
- Provides operations for role management (query, add, and remove roles, and check for their existence). The simplest specification would use the name of a role as a character string or the role concept type.
- Does not define attributes, because it represents the logical unit, composed of core object and role objects.

`CoreConcept` (the customer's *core concept*)

- Manages the role objects added to a core object.
- Implements abstract domain operations of the subject.

The RoleConcept

- Manages a reference to the core object.
- Offers a standard implementation of the subject's abstract domain operations, which delegates operation calls to the core subject.

SpecificRole (*borrower, investor*)

- Models and implements an extension of the core concept for an domain view. In our model architecture, this is a product domain's view of a core concept from the business domain.

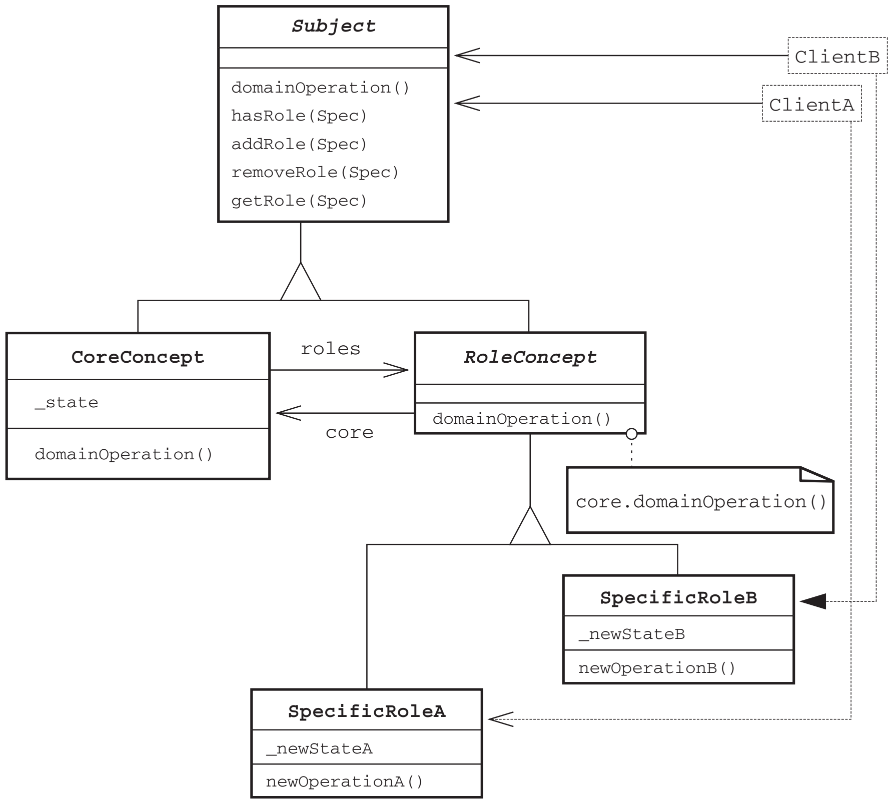


FIGURE 9.17

The *Role* pattern.

Interactions

The interactions between core and role objects can be described as follows:

- A role object passes the subject operation calls it receives to the core object.
- All role objects of a subject use the same core object.

A client interacts with a core object and its role objects as follows:

- A client can add a role object to a core object. To do this, the client describes the roles in the form of a specification object. If no `Product Trader` is used, then the client creates a new role object directly.
- A client requests a desired role from a core or role object. If the requested role object exists, then it is used for further processing.
- If the core object does not have the desired role object, then it informs the client by sending an error message. Core objects never create role objects directly; they always have to be added to the core object by a client.

Trade-offs

The *role* pattern offers the following benefits:

- The core concept has a high degree of cohesion, because it does not include interfaces and attributes motivated by one specific view (e.g., product domain).

- Role concepts can be developed independently of one another, because no changes to the core concept are required.
- Role objects can be dynamically added to and removed from core objects. At runtime, the main memory contains only objects that are actually used.
- Within our T&M model architecture, we can separate product domains easily, because changes effected from a product domain's view to a role have no impact on the role concepts of other product domains.

The *role* pattern has the following drawbacks:

- Rules referring to the interaction between role objects and their core object cannot be checked by the type system. For example, if adding a role depends on an existing role, then this can be tested only at runtime.
- The role pattern introduces additional coding on the client side. In general, clients have to first request a role object from a core object. If the requested role object does not exist, then the client has to create it. In addition, the client has to ensure that the core object can actually play that role, so that it can later address the subject in the desired role.
- The role pattern introduces additional coding to implement the subject identity. This means that the subject, the core concept, and the role concept need appropriate methods to be able to map the subject identity to the technical object identity of core and role objects.

IMPLEMENTATION

- *Role and core objects have to have the same interfaces*, otherwise, we cannot use a role object in place of a core object. This requirement can only be implemented by inheritance in a typed object-oriented programming language. This means that role concepts have to inherit from core concepts. In addition, inheritance allows us to extend properties of the core concept in a role concept. However, since all role objects share the attributes of their core object, they have to decorate their core object. In the implementation, this leads to the abstract class Subject, which defines the deferred operations. These abstract operations are the core's interface. The class `CoreConcept` implements the core. A class `RoleConcept` defines the interface for specific roles and the interaction with the core object. However, a core object is always used over the subject's abstract interface.
- *The creation of specific roles should be hidden from the client*. This ensures that the client does not need to know which roles are implemented by what classes. We can solve this requirement easily by using the *trader* pattern (see Section 9.4.2). The abstract class `RoleConcept` would act as trader, procuring roles for a core concept. The client would deal with the core object over the subject's interface to create the desired role.
- *Managing role objects*. The class `Subject` has to provide a suitable abstract interface, which is implemented by the specific core concept to ensure that a core object can manage its role objects. The core object manages its role objects in a hash table. If a role can be added only once to a core object, and if it is guaranteed that several roles cannot be viewed under a common abstract role, then we can use the role specification in Figure 9.17 for creation as our key. Otherwise, we would have to use different specifications to create and manage roles.

- *Core objects have to be dynamically extended by role objects during their lifecycle*. The requirement for dynamics can be implemented only by a use relationship between the core object and role objects. The reason is that popular object-oriented programming languages do not support dynamic reclassification, such as by changing the inheritance relationship at runtime.
- *The use relationship between the core object and role objects can be used to implement the required subject identity*. When all role objects that extend a core object know this object via a use relationship, we can use the technical identity of the core object as a subject identity. Role objects will then have the same subject identity when they reference the same core object.
- *The state integrity of the conceptual subject identity has to be maintained*. The properties of an object in the application domain can be modeled in several technical classes by use of the role concept. This means that the identity of that object is distributed over several technical identities. Apart from the fact that we have to build an additional domain subject identity, this form of implementation can cause problems with regard to the state integrity of the subject. The reason is that the state space of a domain object is distributed over several state spaces in the software model. If there are dependencies between the individual state spaces, then suitable mechanisms (e.g., the *observer* pattern) are required to synchronize them.
- *Extending the properties of a core concept in a role concept can cause problems*. To better understand this statement, consider changes to a customer address as a practical example. Assume that the operation `changeAddress` is extended both in the `borrower` role and in the `investor` role (technically, this can be implemented both for the borrower and for the investor by redefining the operation `changeAddress`). In this situation, if you call the operation, you will obtain three different results, depending on whether `changeAddress` is processed by the `Investor`, the `Borrower`, or the `CustomerCore` object, for example:
  – If the core object is called, then none of the extensions will be executed.
  – If one of the role objects is called, then the extension (redefinition) in that role object is processed, but without affecting the other role objects.

  The problem is that a subject is implemented as several classes, so that only one of the three operations will be executed. However, from the application-specific view, all of these objects should behave like a subject. For this reason, if we need to change an address, we have to run all three operations. To solve this problems, we could implement the core and role concepts so that they rely on the *observer* pattern or template operations.
- *The roles of roles*. We can implement roles by applying the *role* pattern to role concepts recursively (see Figure 9.18). For example, the role `customer` of the core concept `person` in the example shown in Figure 9.15 has been decomposed again into a core concept, `CustomerCore`, and the role concepts `investor` and `borrower`, by use of the *role* pattern.

  Note that the recursive application of the role pattern produced two core concept classes. For this reason, we have to distinguish between a *root concept* and a *core concept* in an implementation.

- *Managing a subject identity*. In addition to the technical identity of core and role objects, there is a domain identity, which identifies the role objects and their core object as one subject. Since all role objects know their core object, this subject identity can be easily maintained through the core object's technical identity. Two role objects are part of the same subject if they extend the same core object. In a recursive application of the *role* pattern, the subject identity is realized by the root object. Two role objects, such as `Investor` and `CorporatePartner`, are based on the same subject if they refer to the same root object, such as `NaturalPerson`.
- *Combining inheritance and role hierarchies*. Since they are implemented by the use of normal inheritance, role hierarchies can be easily combined with inheritance hierarchies. Both core and role concepts can be specialized by inheritance. For example, the classes `NaturalPerson` and `LegalPerson` specialize the core concept `Customer`, and the classes `CorporatePartner` and `Associate` specialize the `Shareholder` concept.

The object diagram in Figure 9.19 shows a subject, `Person`, which consists of a core concept, `NaturalPerson`, and two role concepts, `CorporatePartner` and `Customer`. The role `Customer` assumes additionally the `Borrower` role.

SAMPLE CODE

The program code in Figure 9.20 shows how we can implement the following example: a core concept, `Person`, should get a role, `Customer`. To keep this example simple,

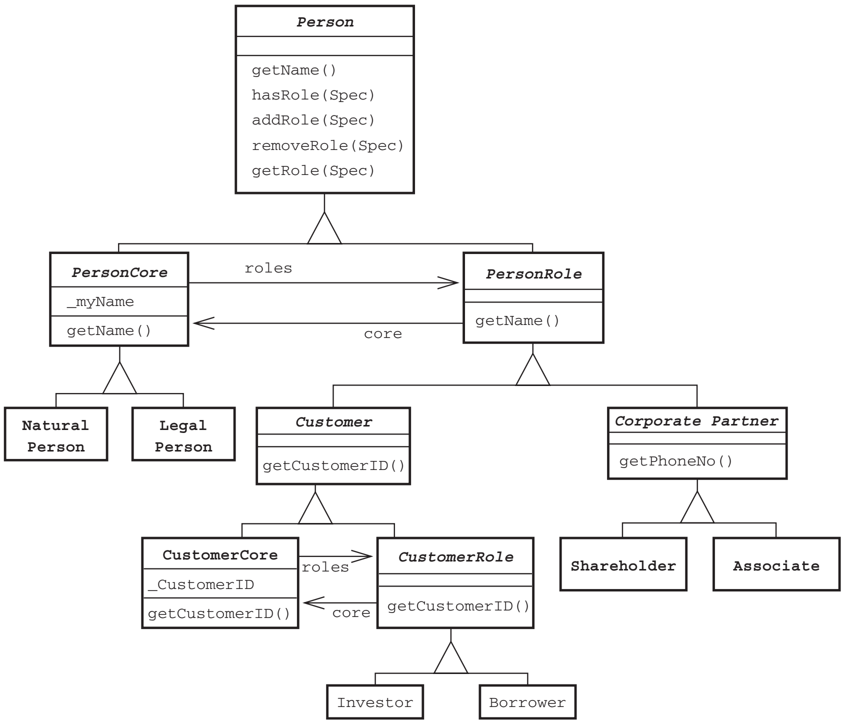


FIGURE 9.18 Recursive application of the *Role* pattern.

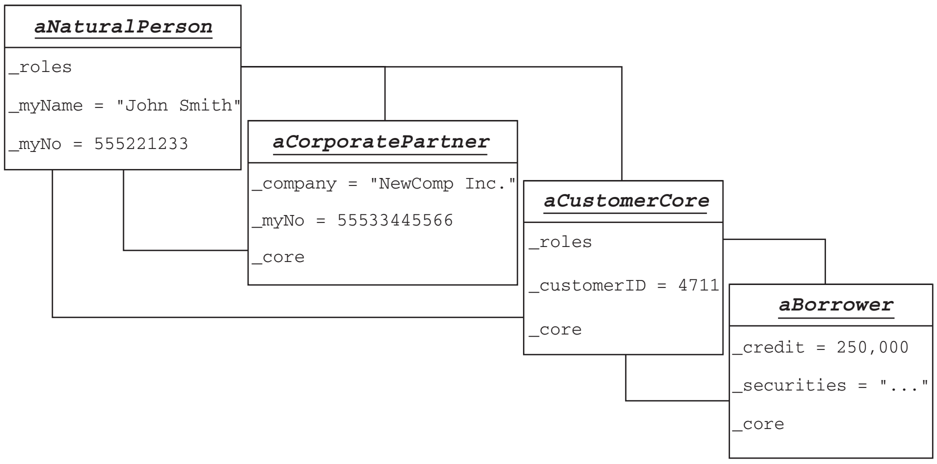


FIGURE 9.19 An object diagram showing roles of roles.

```
public abstract class Person {
// domain-related operations
   public abstract PhoneNo getPhoneNo();
   public abstract void setName(String aName);
   ...
// role management
   public abstract boolean isSameSubjectAs(Person aPerson)
   public abstract PersonRole getRole(Spec aSpec)
   public abstract void addRole(Spec aSpec);
   ...
   protected abstract PersonCore getRootCore();
   protected abstract Map getRoles();
}
```

FIGURE 9.20 Code example for class `Person`.

it does not consider that a role can be added to a core object more than once. This means that we will not distinguish between a specification to create and one to manage roles. First, let's see how we implement the class `Person` (see Figure 9.20).

This implementation defines the superclass for the two classes `PersonCore` and `PersonRole`. The class `PersonCore` is implemented as shown in Figure 9.21.

Next, we implement the class `PersonRole` as the superclass for all role concepts (see Figure 9.22). It delegates all abstract operations to the core object wrapped by a role object. In addition, this class defines an abstract operation for the domain operations that extend a role concept.

The subclasses of the class `PersonRole` define specific role concepts. The code in Figure 9.23 implements the role `Customer`.

A client would deal with a `Person` and a `Customer` role as shown in Figure 9.24.

```
public class PersonCore extends Person {
 public PhoneNo getPhoneNo()
              {return _myNo;}
 public void setName(String aName)
 {
   _myName = aName;
 // make operation extendable
 // for all roles call
 //_aRole.setNameImpl(aName)
 ...
 public PersonRole getRole(Spec aSpec)
 {
   return _roles.get(aSpec);
 }
 public void addRole(Spec aSpec)
 {
   getRoles().put(aspec,
                 PersonRole.createFor(aSpec, this));
 }
 protected PersonCore getRootCore() {return this;}
 protected Map getRoles()
                        {return _roles;}
   private PhoneNo _myNo;
   private String _myName;
   private Map _roles;
 }
```

FIGURE 9.21 Code example for class `PersonCore`.

FIGURE 9.22 Code example for class `PersonRole`.

```
public class PersonRole extends Person {
  PersonRole (PersonCore aCore)
  {
    core - acore;
  }
  public PhoneNo getPhoneNo()
  {
    return getPersonCore().getPhone();
  }
  public void final setName(String aName)
  {
    getPersonCore().setName(aName);
  }
  public void addRole(Spec aSpec)
  {
    getPersonCore().addRole(aSpec);
  }
  ...
```

(*Continued*)

```
  // trader operations
  protected static PersonRole createFor (Spec s,
                               PersonCore po)
  {
    ...
  }
  protected abstract PersonCore getPersonCore();
  protected PersonCore getRootCore()
                        {return core();}
  protected Map getRoles()
  {
    return getPersonCore(), getRoles();
  }
  //allow Roles to extend setName()
  protected void setNameRoleSpecific(string aName) {};
  private PersonCore core;
}
```

FIGURE 9.22
(*Continued*)

FIGURE 9.23
Code example for class `Customer`.

```
public class Customer extends PersonRole {
  public CustomerID getCustomerID()
  {
    return _myCustomerID;
  }
  protected void setNameRoleSpecific(String aName)
  {
    // extended Implementation
    ...
  }
  protected Customer(PersonCore aCore)
  {
    _core = aCore;
  }
  protected PersonCore getPersonCore()
  {
    return _core;
  }
  protected abstract CustomerCore getCustomerCore();
  private PersonCore _core;
  private CustomerID _myCustomerID;
}
```

```
Person aPerson =
  DataBase.loadPerson("Jackie Jones");
Customer aCustomer = (Customer)
aPerson.getRole(Customer.class);
if (aCustomer ! = null)
{
  aCustomer.setName("Jackie Johnson");
              // has married
  System.out.println("Changed name for customer id" +
  aCustomer.getCustomerID() + "to" + aCustomer.getName()};
}
```

```
"Changed name for customer id 55585878889 to Jackie Johnson"
```

FIGURE 9.24

Code example for using `Person`.

Rationale

The *role* pattern should be used in the following cases:

- If core concepts in different contexts should have specific interfaces and attributes.
- If we need to dynamically add the context-specific view to a core object and remove it again.
- If the subject, consisting of a core object and role objects, should have a subject identity.
- If the different set of views (roles) of a core concept should be independent with regard to further development.

The role pattern should not be used when there are strong dependencies between the roles.

9.4.2 The Product Trader Pattern

Intent

Clients do not have to instantiate objects directly by calling the explicit create operation (constructor) of the concrete class. They can use the *product trader* to have these objects created indirectly by the abstract superclass of this class. The trader procures between clients and suppliers, which are here interpreted as service product providers. This means that the product trader detaches a client from the specific class hierarchy of the suppliers, offering the prerequisites for easy adaptation and development of this class hierarchy. This particularly facilitates the evolution of frameworks and application systems.

Problem

Only an abstract interface should be used, even for object creation, to strictly maintain loose coupling between clients and service providers.

In most common object-oriented languages, we have to know the specific class to be able to create objects. This is undesirable, such as when concepts of the business domain within the T&M model architecture are concretized by objects from higher layers. The specific classes of these objects must not be known in creating operations of the business domain's classes, because it would mean that these layers would be invalidly coupled.

Background

We know from our discussion of the T&M model architecture in Section 9.3.7 that objects in a lower layer (e.g., the business domain) can reference objects in a higher layer due to polymorphic assignment. A referenced object is handled over the interface of an abstract superclass, which is also part of the lower layer. This technique forms the basis for the open-closed principle, allowing us to implement anticipated extensions of one layer in higher layers.

Objects from higher layers must not be instantiated in lower layers by directly calling the specific create operation. This would cause dependencies of lower-layer classes on higher-layer classes. Such dependencies are not allowed in our model architecture.

In many cases, however, we want to create objects in such a lower layer. On the abstract class level, we could then describe the interaction of objects and define how they should be specifically created.

Example

Figure 9.25 shows a class hierarchy for a bank domain that represents the relevant (banking) values. Examples of domain values include the classes `AccountNumber`, `Amount`, `InterestRate`, and `Date` (see Section 8.10). Domain values are used in almost all frameworks of the business and product domain layers as well as in the use context layer, as in the case of electronic forms, modeling bank forms in classes. The fields of these forms are represented by domain values.

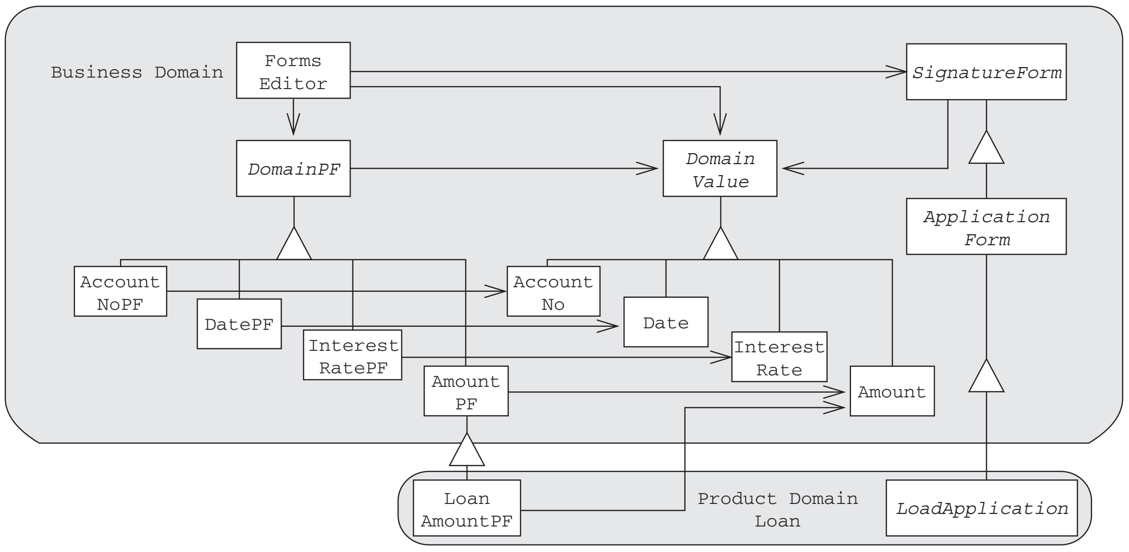


FIGURE 9.25 Example of class hierarchy in a bank domain.

```
public class LoanApplication extends ApplicationForm
{
  public DomainValue[] getFields(){...};
  ...
  private AccountNo _accountNo;
  private Date _openingDate;
  private Amount _loanAmount;
  ...
}
```

FIGURE 9.26 Code example of class `LoanApplication`.

A program fragment of the class `LoanApplication`, limited to the most important characteristics, modeling a domain object, `LoanApplication`, could look like the code in Figure 9.26.

A forms editor used to represent and edit forms on the screen requires appropriate presentation forms to represent domain values (see Section 8.8). These presentation forms utilize the specific representation and editing options for domain values. For example, an amount with decimal places for thousands in a bank application is separated by commas, and the sign is put after the amount. Figure 9.25 also shows a class hierarchy, corresponding to the domain values and presentation forms (PFs). A presentation form, `LoanAmountPF`, specialized for the bank's loan business, represents the domain value `Amount` with a leading sign.

Figure 9.25 shows how the classes are distributed in the T&M model architecture. For example, the classes `LoanAmountPF` and `LoanApplication` are part of the `Loan` product domain. All other classes reside in the business domain layer, because they represent core concepts for the product domains.

The example discussed in this section uses a forms editor for editing signature forms. This forms editor has to create a matching presentation form for all domain values it requests by calling the `getFields` method. We can identify the following important points for this example:

- The forms editor contains a large case statement to create a suitable presentation form, depending on the domain value type. Since the editor has to create the specialized presentation form, `LoanAmountPF`, for a loan amount, an invalid dependence to a class (`LoanAmountPF`) in a higher layer is produced in the class `FormsEditor`. In addition, the case statement has to be adapted for each extension of the domain value hierarchy. Each change to the class hierarchy of the domain values can cause the business domain to be opened.
- The domain values implement a so-called *factory method* (as proposed by Gamma et al.), which creates an instance of the appropriate presentation form. The allocation of a domain value to a presentation form cannot then be dynamically configured. Instead, it is specified once and then used to create the same presentation form in all applications when the factory method is called. For example, if we allocate the presentation form `AmountPF` to the domain value `Amount` in the factory method, then we wouldn't be able to adapt this allocation for a loan application without having to change the code or set additional parameters. Again, we would have to open the business domain.

- An *abstract factory* as proposed by Gamma et al., hides the creation of objects from a client (`FormsEditor`, in this example), by providing an abstract operation to create a matching presentation form for each domain value. A specific factory implements abstract operations by creating the desired presentation forms. Though this design pattern allows a specific factory for the loan domain to allocate the domain value `Amount` to the presentation form `LoanAmountPF`, it has a few problems. The `FormsEditor` has to implement an expensive case statement to determine which operation of the abstract factory has to be called, depending on the domain value type. New domain values lead to new abstract operations in the abstract factory. Changes to the domain value hierarchy or the presentation forms require us to open the business domain, even when these changes are implemented in one single product domain.
- Alternatively to using a factory, the forms editor could be specialized in each product domain, with specific requirements to the representation and editing of domain values. Unfortunately, this may cause problems when the forms editor is required in a use context where specialized value representations from several product domains are required. In this case, we would have to add another specialization of the `FormsEditor` class in this use context. To implement such a specialization, we would have to use multiple inheritance of copy code. None of these two implementation methods is elegant.

Solution

We design an instance that assumes the role of a trader for services. Clients can then request the trader to determine a product, that is, a class or object that provides the requested service. The trader searches for a matching service supplier. This means that the client does not need information about the specific service or its supplying product.

The trader acts as a procuring instance. In our example, the forms editor could address this instance for all domain values and would thus obtain the matching presentation form.

In the object-oriented world, services are provided by objects[3] offering operations to other objects. In general, a client requests a service from a service provider in the form of a specification, where specifications themselves can be objects. Such specification objects can be different, including:

- objects that should cooperate with the service provider;
- service descriptions, such as the runtime behavior of an operation; and
- unique identifications, such as a class number or metaclass.

These specification objects form the basis for a further interpretation of the client-provider model of the use relationship (see Section 2.1.6). More specifically, the client passes its requirements for a service in the form of a specification to the trader. The trader uses this specification to identify the specific supplier of the requested service and initiates this creation. It is important to note that the specific type of the desired service product will be defined at runtime. Since the client requests the desired product over an abstract interface, it can use only this interface to access the object. The specific service

3. These objects can be instances of a class or class objects.

Zull-09.qxd 20/8/04 7:41 AM Page 331

```
public class FormsEditor
{
  public void edit(SignatureForm aForm)
  {
    DomainValue[] aValueList=
    aFrom.getFields();

    for (int i=0; i<aValueList.length; i++)
    {
      DomainValue aValue=aValueList[i];

      // create the matching PF
      DomainPF aPF=
      DomainPF.createFor(aValue);
      // position the PF
      ...
    }
  }
  ...
}
```

FIGURE 9.27

Code example for class `FormsEditor`

provider remains totally hidden from the client. In addition, allocation of a specification to a service can be adapted to changing conditions at runtime. The mechanism used by the trader ensures that the frameworks of one layer are closed with respect to object creation while still remaining open for changes. This means that traders facilitate the evolution of class hierarchies, frameworks, and application systems.

EXAMPLE

To better understand this discussion, let's return to our previous example to introduce a trader. If the allocation of a domain value to a presentation form is done by a trader, instead of using the editor, domain values, or a factory, as in the previous example given in Section "Example", then the trader can create a suitable presentation form for a domain value. In this example, the class `DomainPF` could assume the role of a trader. The class `DomainPF` procures an instance of a domain presentation form based on a specification (i.e., the domain value to be edited). In Java, the class `DomainPF` would have to support the following class operation:

```
public static DomainPF createFor(DomainValue dv);
```

The `FormsEditor` would then implement the creation of a presentation form to represent and edit a domain value as shown in Figure 9.27.

Internally, the class `DomainPF` would have to manage only the allocation (e.g., a hash table) that would match the dynamic type[4] of a domain value to a specific presentation form. This process is efficient and runs within a constant period of time.

4. In C++, we could use the Runtime-Type Information (RTTI) interface to determine the dynamic type of an object. Java offers similar metalanguage features.

STRUCTURE

The structure shown in Figure 9.28 assumes that products procured by a trader each have to be created upon request. We have made this restriction because it simplifies the presentation of this structure. Note that procuring an object that exists in the system instead of creating a new object saves resources and does not cause any problems.

The dotted arrow pointing from the abstract class `Creator` to the abstract class `Product` shows that all the `Creator` class has to know is the type of the `Product` class to declare the return value (a reference to an object of the type `Product`) of the operation `create()`. To compile the class `Creator`, the compiler does not have to know anything about how the `Product` class is built. This means that there is no cyclic dependence between the `ProductTrader`, `Creator`, and `Product` classes. Notice that the product trader is generally never used directly by a client. Objects are always created through the `Product` class. In Figure 9.28, this is shown by a dashed arrow pointing from the client to the trader.

PARTICIPANTS

`Client (FormsEditor)`

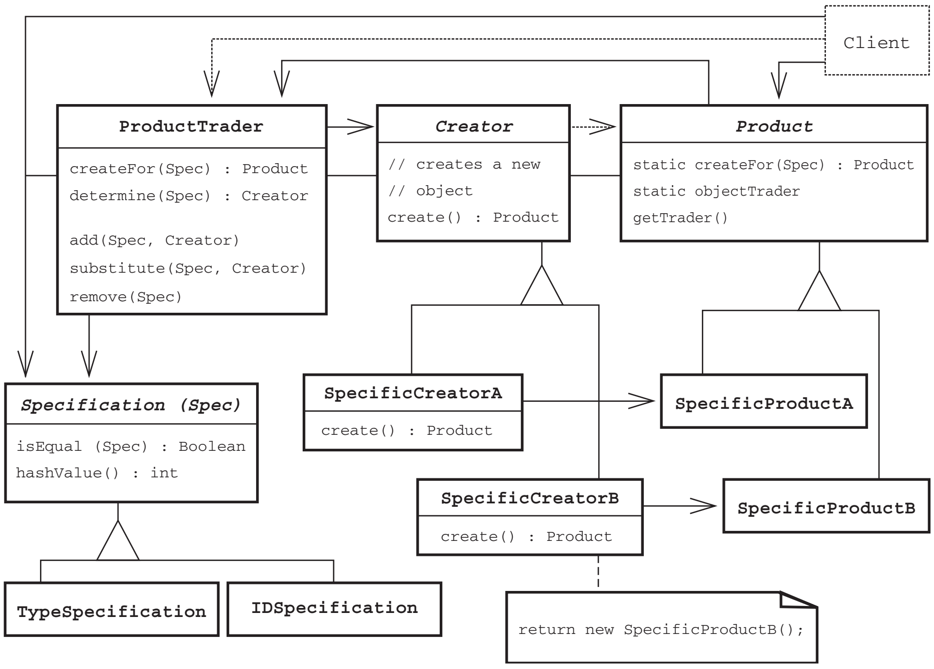
- Creates a specification describing the desired product (object).
- Initiates the creation process by requesting either the class product or the product trader. The desired product is described in a specification.

`Product (DomainPF)`

- Defines the interface of the class hierarchy, which is used to create (procure) products.
- Can use a product trader to create products.

FIGURE 9.28

The structure of a product trader.



`SpecificProduct (AmountPF, DatePF, LoanAmountPF)`

- Represents a specific product.

`ProductTrader`

- Manages and maintains the allocation of specifications to a specific product or a creator object, which can instantiate a specific product.
- Initiates a product to be created based on a specification.

`Creator`

- Defines the interface used to create an instance of a specific product.

`SpecificCreator`

- Creates exactly one specified product.

`Specification` (a domain value)

- Describes a service so that the product trader can create a matching supplier for that service (product).
- Has to be a unique identification of the desired service (product).

INTERACTIONS

The interactions shown in Figure 9.29 are implemented between the objects as follows at runtime:

- The client passes a specification, together with a request to create an object, to a `ProductTrader`.
- The `ProductTrader` manages a hash table, which it uses to determine the correct `Creator` for the specification it received.
- The `ProductTrader` calls the operation `create` on a specific creator (e.g., an object of the type `SpecificCreatorA`).
- A `SpecificCreator` instantiates a specific product by calling the new operator and returns this object to the `ProductTrader`.

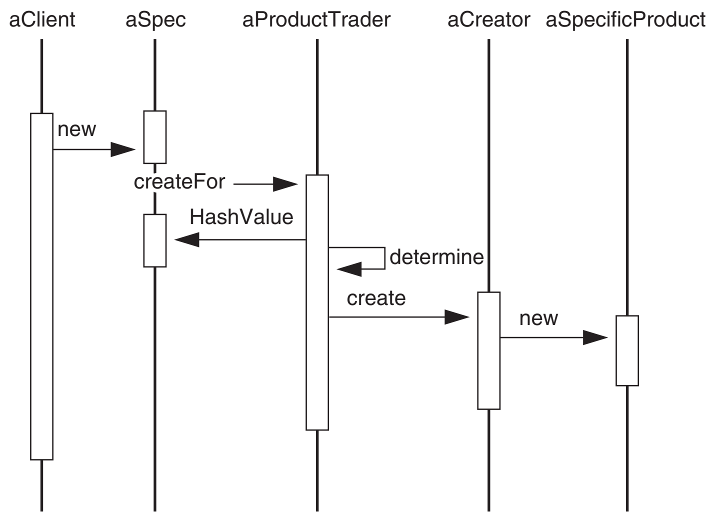


FIGURE 9.29 Cooperating objects in the design pattern.

TRADE-OFFS

The *product trader* design pattern has the following benefits:

- Clients can be fully decoupled from specific products, because they create objects indirectly through a product trader instead of using the `new` operation. The client describes the requested product in a specification.
- *Clients can instantiate lower-layer products from higher layers without creating an undesirable dependence*. If a client accesses a specific product over the interface of the deferred product class, then it can use a trader to reference a specific product without knowing the specific product class.
- *A specific product class, from which we want to create an object, can be determined at runtime*. The desired service is then described in a specification and passed to the product trader for creation.
- *The allocation of a specification to a specific product can be configured*. Using the operations `add, remove`, and `replace`, implemented in the product trader, we can change an allocation statically either when the system starts or at runtime. This allows us to tailor application systems on the basis of frameworks and class libraries.
- *We can use different context-specific product traders for one class hierarchy*. An abstract product class could manage several product traders (not shown in Figure 9.29, to keep the example simple), which means that your application system can choose from different product traders to have products created specifically for a context. In our `FormsEditor` example, the class `DomainPF` could manage two different product traders. Depending on the form to be edited, it would select either the generic product trader to edit the domain value `Amount` with an `AmountPF`, or a specialized product trader, which allocates the `LoanAmountPF` to an `Amount`. To avoid duplicate allocations, your set of different product traders should be linked over a *chain of responsibility*.
- *The class hierarchy of products can be easily developed and extended*. Since the client depends only on the abstract product class, you can easily change specific product classes. In addition, new product classes can easily be added to an existing framework or application system. All you have to do is adapt the configuration descriptions that define the allocations of specifications to specific product classes.

  This means that we can use the *trader* design pattern to close frameworks in one layer against object creation, but still keep them open for changes. This flexibility is not given with other popular constructions. Consequently, the trader design pattern facilitates the extension of class libraries and frameworks. At the same time, it allows you to quickly tailor your application systems (see example in Figure 9.30).

The *trader* design pattern has the following major drawbacks:

- *It adds complexity to structures and dependencies in programs*. The process of creating objects is no longer described statically in the program code to increase flexibility. This means that the compiler is no longer able to check for object creation. In addition, the program code becomes more difficult to read, because it does not show the class where an object is to be created.
- *Our code becomes more error-prone*. Wrong configurations, that is, allocations of specifications to particular product classes, can introduce additional errors that are hard to find, because the program code does not include a new operation.

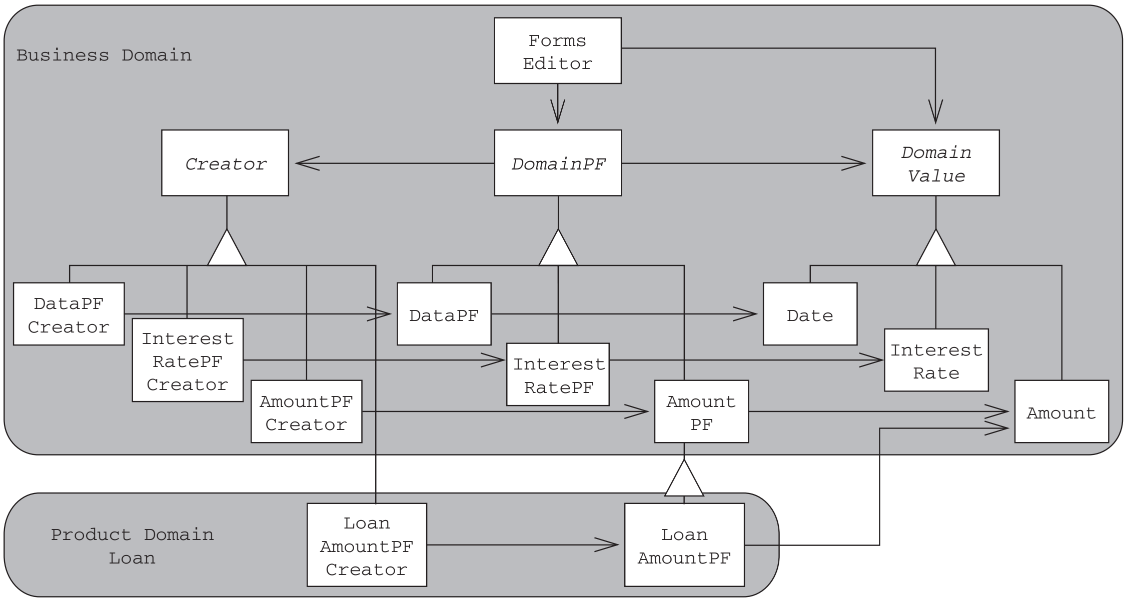


FIGURE 9.30 Constructors in the business and product domains.

- *Extended configuration description (in* C++*).* A product trader can be used to adapt your system's configuration to changing conditions both statically and at runtime. We need special configuration descriptions to be able to write a configuration in the first place, and to ensure that the linker pays attention to specific product classes when linking a library or application system, if we use C++. The configuration description references the specific product class explicitly. This referencing is required, because the client code does not include calls to a `new` operation (the product is procured through the trader). In addition, specific product objects are addressed only over the interface of the product class. This means that no operation of the specific product class is called anywhere in the code. We will discuss several alternatives to a configuration description in the section on implementation that follows.
- *Ambiguous specification.* Depending on the type of specification, it can happen that several specific products match one specification. For example, if a product trader is supposed to select a tool matching a material, then there could be several matching tools in an application system that includes several product domains. To solve this problem, we either have to make the specification unique, based on additional knowledge of the context, or we have to state rules to solve this conflict. Additional context knowledge would be the product domain where the tool should be implemented. Rules could refer to the specialization degree of the specific product class. We could describe this by the distance between the specific product class and the abstract product class within the class hierarchy. In our previous example of presentation forms for domain values, the class `LoanAmountPF` has a higher specialization degree than the class `AmountPF`.

- *Parameters for objects construction*. If specific product classes require additional parameters when creating objects, then these have to be passed. If parameters are passed by the usual mechanism based on a parameter list, then we can pass only parameters that are identical for all product classes. The reason is that the product trader knows only the type of the abstract product class.

  Different parameters can be passed in generic parameter lists, but we have to ensure that the client does not make implicit assumptions about the product to be created. Otherwise, changes to the configuration could cause errors when another object than the one implicitly assumed is instantiated.

IMPLEMENTATION

When implementing the *product trader* design pattern, we have to deal with the following four aspects: the specifications, the creators, the product trader, and the configuration descriptions. A specification can be easily allocated to a specific product (via a create object) by using a hash table.

- *Specifications* can be implemented both as primitive data types (e.g., numbers or character strings) and in an independent class hierarchy. Modeling a class hierarchy has the benefit that we define a uniform interface for specification objects in the form of an abstract specification class (see under "Solution"). Product traders can then be implemented on the basis of this interface, so that they can be easily extended to new specification types.
- *Creators* can be implemented in the form of prototypes or special creator objects. Regardless of the method we select, our implementation should not result in the manual implementation of a constructor class for each specific product class. The two different implementation techniques are as follows:
  – *Prototypes*: This implementation technique lets the product trader manage one prototype object for each specific product. If it needs to create a new instance of this product, then the product trader copies the prototype.
  – *Using an independent creator class* (C++): To avoid having to write separate code for classes in C++, we can use `creator` templates. In this case, we need a template for the class `Creator` and another one for the class `SpecificCreator`. We can use the class of the abstract product to configure the template for the `Creator` class, and a specific product class to instantiate the template for `SpecificCreator`. They both implement the abstract operation `create` in the `Creator`.
  – *Using an independent creator class* (Java): Java lets us use inner classes or anonymous inner classes to easily implement creator classes.
- The *product trader* manages the allocation of specifications to specific products in the form of a hash table. Since it triggers the actual creation of objects, it is product-specific. For this reason, we have to implement a separate product trader for each class hierarchy with classes that should be instantiated by a product trader. We can describe a generic trader by using templates in C++.
- The *client* describes desired services in the form of a *specification*. This means that the product trader configuration, that is, the allocation of specifications to specific products, has to be defined for each application system, with the

following options:

- *Register objects*: We can use register objects for products where the specification is allocated to a specific product class on the basis of a static definition (see Bäumer and Riehle). These register objects are implemented as class variables of the specific product class, so that they are created and initialized automatically when the application system starts. During its initialization, a register object registers a creator object that matches its product class under a statically defined specification with the product trader of its abstract product class.

  In the previous example using domain values and presentation forms, the register object of the class `DatePF` would instantiate a creator object of the type `SpecificCreator` and then register it with the product trader of the class `DomainPF`. The underlying specification would be the type of the class `Date`.

  However, register objects have two major *drawbacks*: First, changes to the configuration require adaptations of the specific product class, which causes the corresponding framework on the relevant layer to be opened. For this reason, we should use register objects only in highly stable configurations or standard cases.

  Second, the use of register objects does not lead to external referencing of the specific product class (the product class is referenced in the product class itself). In C++, for example, the linker would be unable to ensure that all classes required at runtime are actually linked.
- *Configuration scripts*: Each product trader is defined in a special configuration script. This script defines the creator objects to be created and their registration with the product trader configured in the specification. This means that a product trader can be easily configured and extracted from the specific product classes.

• If we combine these two options, the configuration scripts and register objects, we can write standard cases in the program code of the respective classes and describe changes in the scripts. For example, we could use register objects to register the presentation forms with the trader `DomainPF` for specific domain values. Next, all we have to do is describe changes in the configuration script. However, we have to ensure that our register objects and configuration scripts are processed in two steps. First, the system should process all register objects and then the configuration scripts.

• If we don't want to combine configuration scripts and register objects for standard case and changes, then configuration scripts are the preferred method over register objects, because they are easier to understand. In addition, they describe a system's configuration in one place, and because they belong to the application system, they can be easily adapted without having to open a framework.

SAMPLE CODE

The previous section that discussed the *product trader* design pattern showed a detailed example, so we will just give a brief summary of two additional examples in this section. The first example creates objects, which are loaded from a file or a relational database. The second example shows how we can create tools that match materials.

In almost every large application system there is normally a need to store objects persistently in a file system or a relational database, and later load them from there (see Chapter 11). When these objects are stored, the dynamic type of each object is included in the form of a class identifier or class name. When an object is loaded, this identifier is used to create an "empty" object, which is then filled with stored values. We can use a trader to manage the allocation of an identifier to a creator object. To create an object, a client just passes the identifier to the trader. For example, if there is an aspect, `Storable`, to store and load objects, then we could let this class manage an appropriate trader, which could look like the code displayed in Figure 9.31.

The second example shows how we can use the *product trader* pattern to create a tool. Assume that we use tools to handle materials in an application system. Basically, these tools have to be suitable for the materials in question. On the other hand, a material should be suitable to be manipulated by a specific tool, depending on the configuration, the work context, or other runtime factors. Therefore, we should use a trader to dynamically allocate a material to a tool. In this example, the environment would generally assume the role of a trader, and the material class would be used as the specification to determine the matching tool.

Rationale

The product trader should be used in the following situations:

- When a client should not know which services are provided by which suppliers. This means that the client is totally detached from the existence of specific suppliers.
- When the specification of a requested service cannot be determined earlier than at runtime.
- When the allocation of a specification to a service provider should be adapted statically or at runtime, without violating the open-closed principle, for example, when there are several suitable presentation forms for one domain value.

The product trader should not be used as a general replacement for direct object creation or for the factory method or abstract factory patterns. The reason for this is that the use of product traders has some drawbacks: for example, it introduces complexity and makes the application system more error-prone.

FIGURE 9.31
Example code for class `Storable`.

```
public class Storable {
// abstract operations for storing and loading
...
public static Storable createFor(long ClassNo)
{
  return _aTrader.createFor(
  IDSpec(ClassNo));
}
  private static ProductTrader _aTrader;
}
```

9.5 REFERENCES

D. Bäumer, D. Riehle: *Product Trader*. In: R. C. Martin, D. Riehle, F. Buschmann (eds.): *Pattern Language of Program Design 3*. Reading, Mass.: Addison-Wesley, 1998, pp. 29–46.

More concepts and discussions about the *product trader* pattern.

E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*. Reading, Mass.: Addison-Wesley, 1995.

The key reference for this chapter.

G. Gottlob, M. Schrefl, B. Röck: *Extending Object-Oriented Systems with Roles*. ACM Transactions on Information Systems, 14(3), July 1996, pp. 268–296.

Important paper about the concepts behind the *role* pattern.

B. B. Kristensen, K. Østerbye: *Roles: Conceptual Abstraction Theory and Practical Language Issues*. In: Theory and Practice of Object Systems, 2(3), 1996, pp. 143–160.

Important paper about the concepts behind the *role* pattern.

B. Meyer: *Object-Oriented Software Construction*. New York, London: Prentice-Hall, 2nd ed., 1997.

Contains the introduction of the open-close concept.

D. L. Parnas: *On the Criteria to be Used in Decomposing Systems into Modules*. Communication of the ACM, 5(12), December 1972, pp. 1053–1058.

The classical paper on modularization.

T. Reenskaug, P. Wold, O. A. Lehne: *Working with Objects*. Greenwich: Manning, 1996.

An approach to object-oriented design with the modeling.

E. Yourdon, L. L.: *Structured design: fundamentals of a discipline of computer program and systems design*. 2nd ed., York: Yourdon, 1978.

The Seminal work on modularization criteria.

This page intentionally left *blank*

10 Supporting Cooperative Work

This chapter explains how the support of cooperative work processes can be integrated seamlessly into the T&M approach. We first discuss some background to better understand the concept of computer-supported cooperative work, and then introduce several cooperation models, showing how they can be implemented by T&M concepts.

The T&M approach focuses on workplace applications. This usually means a distributed system both in a technical and a domain-related sense. When looking at the support for distributed work in current software systems we will mostly find it on a technical level. There a few explicit cooperation mechanisms. But cooperation has rarely become an integrated part of the application system.

This chapter tries to overcome this problem. Application developers will learn how they can integrate an application-oriented concept for shared work into their software systems.

10.1 BACKGROUND: COMPUTER-SUPPORTED COOPERATIVE WORK

This section introduces the terms and concepts related to computer-supported cooperative work (CSCW). It is both introductory and presents our application-oriented notion of this topic.

10.1.1 CSCW

CSCW as important design issue

Computer-supported cooperative work is one of the most important areas of the current and future office application domain. It includes many different kinds of office applications, distributed systems support, document management and storage, and multimedia information exchange. In developing our T&M design approach, we originally built support for individual workplaces primarily in the software engineering environments and in the banking sector. Soon, however, we found that we had to integrate components to support cooperative work, since cooperation among every organization's staff is an integral part of people's work.

As a consequence, we have extended our approach to include the support of cooperative work. While our original approach was task-oriented, involving users and software system components, our shift to include cooperative work is characterized by the following issues:

- to combine functionality and cooperation within tasks;
- to distinguish between explicit and implicit ways of cooperation; and
- to find the means and media to support cooperation and the cooperative process itself.

In this chapter, we explain which types of cooperative work we should support and how this support can be accomplished.

A good basis for our attempt to support cooperative work is the identification of different workplace types. Each workplace type provides adequate equipment (i.e., tools, materials, automatons). More specifically, we start working from the underlying principle that workplace types are related to how work is organized in an application domain. We discussed workplace types in Section 3.6 and tools, materials, and automatons in Chapters 7 and 8.

DEFINITION: COOPERATION AND COORDINATION

The way in which people cooperate is different, depending on the tasks and goals. We thus say that there are different cooperation models.

A *cooperation model* is a usage model that describes and regulates a cooperative situation either explicitly or implicitly. For our purposes, we are looking at cooperation models that are an integral part of a software application system.

***Computer-supported cooperative work* (*CSCW*) describes coordinated activities performed by a group of participants in order to reach a common result supported by a computer system.**

The term *groupware*, which is commonly used in the literature, describes an actual support system for CSCW, that is, the implementation of specific tools enabling coordinated group activities. The tools forming such a system can range from group decision support programs (e.g., spreadsheets, statistical analysis tools, or brainstorming support) or online group communication facilities (e.g., multicast communication mechanisms or intelligent message filtering systems) to complete coordination environments (e.g., intelligent agents that perform complex group activities in a semiautomated way and controlled by sophisticated user interfaces.

Important characteristics of cooperative work

In the T&M approach, we are mainly interested in the following characteristics of cooperative work:

- The cooperation participants want to produce a *common product* or *service*.
- They have to *share limited resources*, which means that, in the simplest case, they have to share the work object, or a material.
- They have to *coordinate* their *work activities* to maintain temporal and logical sequences or processes.
- They have to reach a common *understanding* of what will be done by whom.

This means that cooperative work is always related to coordination. In many cases, there are common conventions among the participants about how their cooperation should be controlled and coordinated, in addition to having well-established cooperation rules.

In this context, *coordination* is a process or mechanism used to coordinate shared tasks within cooperative work. This coordination can be based on mutual—sometimes implicit—conventions or explicit rules.

If we want to use application software to support cooperative work, we should not forget that we also have to take the coordination aspect into account, in addition to the actual cooperation. In the simplest and most common case, cooperation and coordination are modeled in application software in an extremely rudimentary way. Often, the participants cooperate without system support. We will look at such a case in an example in a moment.

First, let's look at different domain-motivated cooperation models:

***Implicit cooperation:* In an implicit cooperation model, the concurrent access of several users to joint resources is permitted and visible in the usage model. However, the application system does not include a model that specifies and regulates or polices the participants' cooperation (see Section 10.2).**

***Explicit cooperation:* The cooperation is modeled and policed explicitly in the application system. For example, joint rooms are used, processes are supported, and forms of complex cooperation are visible (see Section 10.3 and 10.4).**

Each of the above cooperation models requires cooperation tools and media to become feasible.

In our discussion, a *cooperation material* is a domain object that supports cooperation. It represents the cooperation or the underlying coordination. Examples for cooperation materials include transaction files and routing slips. A transaction file allows us to forward or distribute documents within a shared process. A routing slip represents the sequence of cooperative steps and the people involved.

A *cooperation medium* is a domain object enabling cooperation in an application system. All cooperation media have in common that they can be used to exchange materials or information. Examples for cooperation media include electronic mailing systems, group mailboxes, or electronic blackboards.

Example

Assume that a device manager works independently and autonomously. To manage and plan devices in that organization, the device manager uses a single-user system. The manager coordinates his steps with his colleagues by distributing and discussing printed and manually edited room and equipment plans.

The same system runs on several workstations, or computers at different workplaces, connected over a local area network (LAN), so that all staff involved can access a common file system, which is the *cooperation medium*. This means that the device manager and his cooperating employees can file and access the same room or equipment plans (see Figure 10.1). There is no further system support for this cooperation.

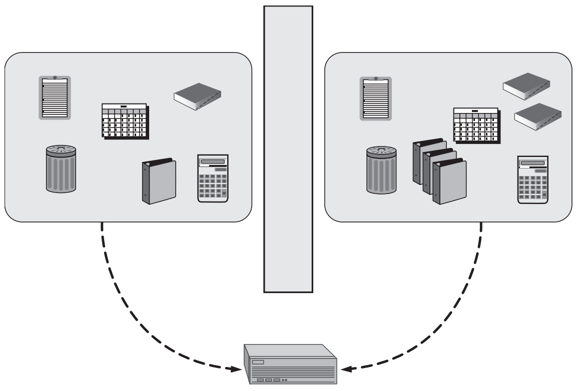


FIGURE 10.1 Two workplaces cooperate over a common file system.

An important notion for working with materials within a cooperative context is the *notion of place and time*:

Notion of place and time: **A material can be available at exactly one location at any given time. If a user handles a material at his or her workplace, then it cannot be handled by other users (at the same time). In general, it is not visible to other users.**

Another important notion is that of *original* and *copy*.

A material can be handled as an *original* or a *copy*. Materials in general, and documents in particular, are often available as originals and copies in most work environments. To follow the usual meaning of these terms, *copies* are materials in their own right, which can be handled and edited, like photocopies. There is no technical connection between an original and its copies, so that changes to one have no impact on the other.

Notice that *technical copies* are different from the copies such as these. Technical copies can be created for software reasons, for example to minimize access times.

10.1.2 Technical and Domain Transparency

Technical transparency

Most discussions about distributed systems and CSCW use the term *transparency*. People understand different things by the term transparency, depending on whether it is used in a technical or an application-oriented sense. In the technical sense, transparency in distributed systems means that the distribution of components should be invisible in the application development and the application software. As a fully transparent glass pane, distribution should not be visible at all. This notion is based on the idealized case of infinitely fast, infinitely large systems totally independent of a specific location.

Application-oriented transparency

In our T&M approach, we talk of transparency in the application-oriented sense. This means that distribution should be visible, or transparent, in a cooperative system. We want to give users (and application developers) a comprehensible image of limited resources and their spatial distribution to better understand the cost and consequences

Type of Transparency	Realized by Use of the Workplace Design Metaphor
Locality	The users know where a material is located. A material is always in a specific location. This location is either the workplace or a location within the common environment, from where a material can be procured.
Access	Users at the workplaces know the locations where materials can be accessed directly or by explicit request.
Migration	To move a material from one location to another within a work environment, the user has to initiate an explicit activity.
Simultaneous access	Different workplaces cannot have write access to the same material. Joint write access is replaced by the concept of copies and original.
Copy management	The users are responsible for the management of modified copies and the original.
Failure	A system failure at remote, jointly accessible locations, where materials are supplied to the work environment, is not hidden from the entire application domain.

TABLE 10.1 Application-oriented transparency.

of accessing nonlocal components. Table 10.1 describes our interpretation of the application-oriented notion of transparency.

10.2 IMPLICIT COOPERATION

We speak of implicit cooperation whenever a user shares a cooperative work situation with other users within a work environment, in which the cooperation does not *initially* transpire. Our assumption is that there are common work materials.

Important characteristics of implicit cooperation

This form of cooperation transpires when several users access materials concurrently from their workplaces, as in the example given above. We define the following important characteristics for this form of cooperation:

- To complete a cooperative task, the users involved have to share a common material.
- There are several workplaces within one common work environment.
- None of the workplaces can see the other workplaces.
- The participants coordinate their joint work through conventions outside the application system.

Transparency in implicit cooperation

But there is a major problem inherent in implicit cooperation—the *transparency* when jointly using the same materials. In this context, transparency means that a user can see in the application system that there is a competitive situation:

- The system shows that more than one user wants to use the same material.
- The notion of location and time is maintained while a material is used.
- The coordination is essentially based on conventions outside the system.

In implicit cooperation, the application-oriented cooperation model is primarily limited to supporting a *cooperation material*. The underlying coordination processes are normally implemented outside of the technical environment.

Popular cooperation materials for implicit cooperation include common archives or a common registry. Since the registry is an important form of persistence service it will be discussed in Chapter 11, when we will also describe the appropriate usage model.

10.3 EXPLICIT COOPERATION BY EXCHANGING MATERIALS

A simple form of explicit cooperation allows users to directly exchange materials. We all know this form of cooperation from the inter-office domain. Files, folders, or documents are exchanged between a limited number of users, and each of these users knows the role that he or she plays in completing a common task.

Explicit cooperation by exchanging materials differs from implicit cooperation mainly in that the cooperation itself is visible. Users know what material they pass to others and who is in charge for what part of their common task.

Important characteristics of explicit cooperation by exchanging materials

We identify the following important characteristics of explicit cooperation by exchanging materials:

- The cooperating participants exchange work materials explicitly within a joint task.
- Cooperation media (see Section 10.1) are used to connect the electronic workplaces involved within a common work environment.
- Each workplace involved can see what other workplaces are basically present or actually involved in the task.
- The participants know the type of their cooperation based on conventions. In addition to a way to pass materials, they do not require any particular mechanism within the application system to coordinate their work.

We can see that this form of cooperation expands the set of T&M metaphors, that is, we have different cooperation media in addition to cooperation materials. Depending on the type of cooperation and the desired support, we can select from

- joint mailboxes;
- point-to-point connections, or
- a mailing system.

We can combine all of the above methods with cooperation materials to explicitly provide coordination support. These cooperation media and models will be discussed in the following sections.

10.3.1 Cooperation Medium: Mailboxes

Mailboxes are a simple form of explicit cooperation by exchanging materials that are normally supported in office environments. If we want to transfer the conventional

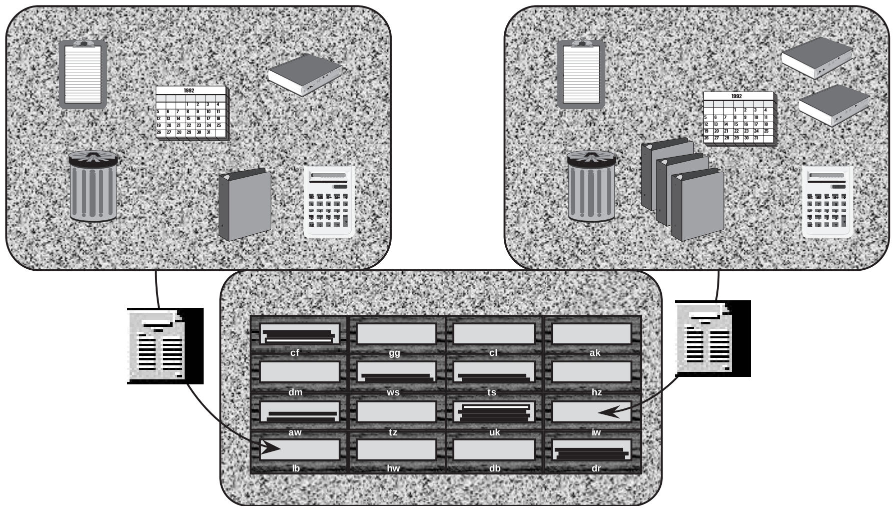


FIGURE 10.2
Two workplaces cooperate through common mailboxes.

concept of mailboxes to application systems based on the T&M approach, at first glance it seems as though we simply have to extend the registry concept. Considering this aspect more closely, we can identify several properties that make mailboxes an independent cooperation medium.

Cooperation Model

In an office environment, a common mailroom with its mailboxes is accessible to each workplace. Each user can place a material in any of these mailboxes or remove materials from them. All users can see the contents of these mailboxes (see Figure 10.2). Removing means that a material is taken from one mailbox and placed on a desk. As soon as this happens, this material is no longer visible and accessible to other users. Within the group that jointly uses these mailboxes, there are conventions regulating who may place or remove what from which mailbox.

In summary, mailboxes can not only be allocated simply to individual users but can also be allocated to groups, departments, or roles.

Discussion

Benefit of unrestricted mailboxes

Mailboxes in a common mailroom represent an important expansion of the space concept. In addition to the individual workplace, there is a room accessible to all members of a group concurrently. This open concept of mailboxes has the following benefits:

- Materials can be exchanged on the basis of a simple model that everybody can see and understand.
- No special mechanisms are required to coordinate the group's cooperation, because each member of the group can see when a material was placed in their mailboxes or on their desks.

- Since each participating user can (technically) access all mailboxes, it is easy to establish conventions to regulate work loads, free capacities, and staff replacement rules outside the system.

Construction Approach

The mailbox concept can be implemented in a very simple common file system for all workplaces involved to store jointly used materials. We could use an automaton for each workplace to regularly check for incoming and outgoing materials and to register these materials with the components that represent these mailboxes.

Of course, we could try to implement more sophisticated technical concepts. For example, we could implement an independent domain service to distribute and manage jointly used materials. This service could internally use a standard E-mail service provided, for example, by the Java E-mail API.

The materials handled by such a domain service could be connected to proxies in each workplace process represented in the mailboxes. The proxy is then replaced by the material object as soon as a user moves a material from a mailbox to his or her desk. Finally, the service itself could use a persistence mechanism.

Example

Assume that in our EMS example there is an office environment with a separate mailbox for each employee. Employees can place notes in the device manager's mailbox to request a change to their computer equipment.

When such a change request is dealt with, the device manager places a note in that employee's mailbox. On the following day, the device manager checks whether or not this employee has removed the note from his or her mailbox.

Personalized Mailboxes

Figure 10.3 shows that we could replace the mailbox concept discussed in this section to *personalized mailboxes*. We can see in Figure 10.3 that the employees do not have to address a receiver, because each mailbox is unique to a workplace.

Users can place arbitrary documents or other materials in these personalized mailboxes. If a material is to be passed on to another cooperating person, then it can be sent automatically or by the push of a button. On the other hand, users see that new material has arrived in their mailboxes.

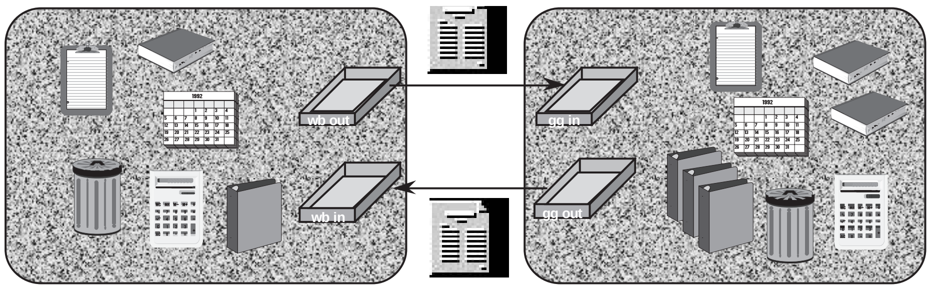


FIGURE 10.3 Using personalized mailboxes to support the cooperation of two workplaces.

10.3.2 Cooperation Medium: Mailing System

A mailing system is another medium to support cooperative work. If we look at a mailing system as a cooperation medium, then we can see that it transports materials in a more flexible way than mailboxes.

Cooperation Model

When using this cooperation model, a user can place materials to be forwarded in a dispatch folder. A dispatch folder is a domain container normally used to hold work materials, such as documents. The user normally adds an address to the dispatch folder. This address can describe a physical person (e.g., Mrs. Jane Smith), or a role (e.g., bill auditing). This shows clearly who should receive the dispatch folder. A dispatch folder is placed in the outgoing mailbox of the sending user and transported similarly to the simple or personalized mailboxes discussed in the previous section.

Discussion

In addition to the conventional transport of materials to specific persons, a mailing system allows us to address materials to roles, as briefly mentioned above, so that we can derive several design alternatives:

- An *algorithm* is used by the mailing system to transport the dispatch folder to a specific workplace, if the user of that workplace can assume the role.
- There is a mailbox for the entire group or this role, and all potential role owners can access this mailbox, so that the dispatch folder is *allocated* to a specific workplace within the group *by convention* in a controlled way.

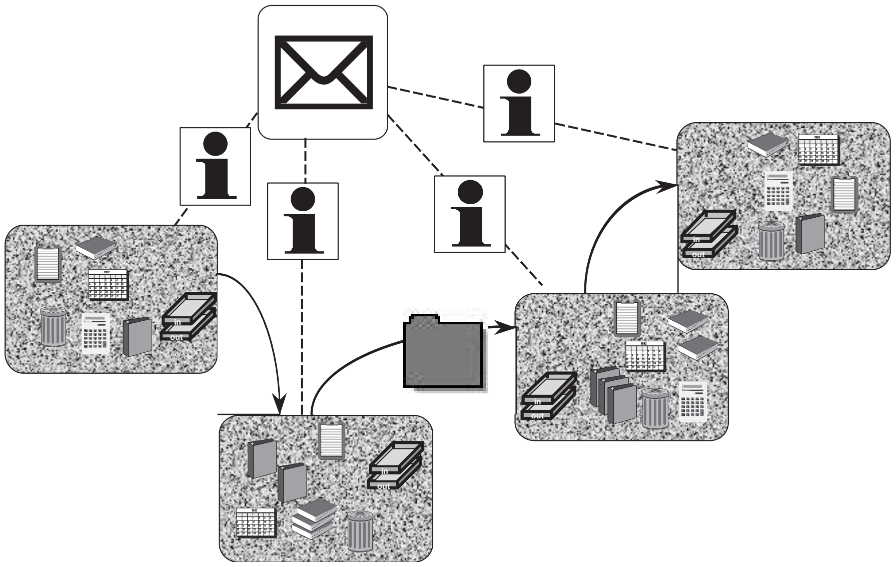


FIGURE 10.4

Using a mailing system to support cooperation between several workplaces.

Notice that the integration of such a mailing system into the work environment represents a relatively sophisticated concept for explicit cooperation, to the extent that this cooperation refers to the exchange of materials. On the other hand, the coordination of work is still based on conventions, and it is implemented by forwarding materials. Each of the cooperating participants has to have intuitive knowledge of the tasks involved and how they can be completed by use of a given material, because there is no such model in the system.

Construction Approach

Technical view on a mailing system

The dispatch automaton of a mailing system uses generic dispatch folders to transport materials between workplaces. The dispatch folders do not make any assumptions about their contents. Even if users can mark or label folders, there should be no need for the system to run consistency checks, because this would limit its use or add unnecessary complexity to the construction. Another property of dispatch folders is that they carry sender and receiver addresses, so that the dispatch automaton can respond to a failure. In the implementation, it is often meaningful to use existing transport mechanisms (e.g., for file transfer (ftp) or E-mail, and as a network protocol (TCP/IP) on a lower layer) as our system base.

In addition, we have to build a mechanism for role resolution, such as tables. Naturally, we could use more complex strategies with proxy control and load algorithms. Although there is no absolute need to implement such complex constructions, we should always keep things simple.

Finally, we intentionally do not want to use visible logging of the transport to register the whereabouts of folders at this point, because this aspect will be discussed as an independent concept.

Example

Let's return to our previous EMS example and assume that the device manager wants to buy a new computer. He places all the cost estimates that he has received and his recommendation in a dispatch folder. Next, he uses the mailing system to send this dispatch folder to one of his corporate managers. Once the corporate manager finishes checking the documents, she adds a note stating her approval. Finally, the corporate manager returns the dispatch folder to the device manager.

10.4 EXPLICIT COOPERATION MODEL: TRANSACTION PROCESSING SUPPORT

We intentionally use cooperation types commonly found in office environments to discuss the support of cooperative work. We also use these cooperation types when discussing the support of transaction processes in organizations. The example discussed in this section involves shared processes, which are supported by an explicit cooperation and coordination model in the system, in addition to exchanging material.

More specifically, we are interested in cooperation forms to support transaction processes that flow in separate temporal and spatial steps. It is characteristic for these forms that different persons, usually with different qualifications, cooperate to solve a task at different points in time.

EXAMPLE: LOAN APPLICATIONS

Let's use a simplified banking example to better understand the concepts discussed in this chapter. Let's further assume that you are an account manager in the loan department of a bank. Your department distinguishes between new and old customers, where new customers do not yet have a business relationship with your bank. In contrast, old customers are known to your bank. If you receive a loan application from an old customer, you already have some information to prepare for a meeting with this customer. For example, you may easily prepare several financing alternatives.

The bank employs other account managers, and the customers are allocated to specific account managers, for example, by customer names or account numbers.

When you process a loan application, you probably fill out the application form with some information, either before or during your meeting with that customer. In addition, you probably make notes about the meeting. Once you approve your customer's loan application, you need to obtain approval from other account managers or corporate managers.

This final approval is based on the loan application and meeting records you prepared. As soon as a second person has countersigned the application, the case is returned to you. You check the loan application case once more and send copies to the departments in charge, such as the loan and controlling departments. Payment against the loan application is effected over a setoff account, specially set up for this purpose. Loans can be paid out by the account manager or at a teller.

Table 10.2 shows how we can use a modified purpose table (see Section 13.8) to describe the most important activities involved in loan applications and the purpose and cooperation type of each.

Table 10.3 shows the dependencies between the activities involved in processing a loan application. The columns *earliest* and *latest* make statements about the time when these activities should be completed. The last column describes whether an activity is mandatory (M) or optional (O).

Decisions about loan applications are taken on the basis of a well-established procedure. The previous (simplified) tables were prepared on the basis of scenarios and interviews with account managers within a bank project. They reflect the (rarely successful) attempt to handle complex work processes, which recur on a regular basis but in different variants.

This type of transaction processing is also called *routinized cooperation*.

Important characteristics of routine cooperation

- In general, the cooperating persons know each other *personally* and have developed a certain cooperation *pattern*.
- Within this cooperation, the participants *exchange materials* as part of the aspired work result or for information, such as transaction files or folders, to allow spatial and temporal separation of a complete task.
- The *number of people* participating in this cooperation is *limited*. A major requirement for this cooperation is that all participants have a common understanding of their work and underlying rules.
- The *number of activities* to be completed is also *limited*, and both the content and purpose of each activity are known to all participants.
- The person who is actually handling a joint transaction has *control* over the progress of this cooperation. This person knows what has to be done next and which steps are taken by whom.

	Who	Does What	With Whom/What	What For?
1	Account manager	forwards	customer, other account manager	customers are allocated based on the bank's established conventions.
2	Account manager	prepares for meeting with customer	customer information and documents	to check basic requirements for granting loan.
3	Account manager	advises	customer	to be able to decide on loan approval.
4	Account manager	fills in	loan application	to have a basis for granting a loan.
5	Account manager	writes	meeting notes	for legal and administrative reasons.
6	Account manager	passes loan documents to	second account manager	for crosschecking (four-eyes principle).
7	Second account manager	signs	loan application and other documents	for final loan approval.
8	Second account manager	returns loan documents to	account manager	to inform account manager of final approval.
9	Account manager	sets up	setoff account	to have a special account for loan payout.
10	Account manager/ teller	pays out loan	over setoff account	to give customer the approved loan.
11	Account manager	checks and distributes	loan application	for further processing and control.
12	Back-office employee	checks	loan application, documents, and account	to control loan payments.

TABLE 10.2 A purpose table for credit application cases.

- We cannot identify or implement a *general workflow* in this transaction processing, though certain forms of cooperation have been established between the participants. This could be due to the following reasons:
  – Many activities can be done by one person at one workplace in parallel.
  – Results from previous activities are mandatory only for part of the activities.
  – Depending on the situation, some activities may be skipped, or special activities can be added.

Though routinized cooperation can basically be described, it will normally not produce an exact map of a concrete routine case. This is one of the difficult points of

	Activity	Earliest	Latest	Mandatory or Optional
1	Customer allocation	–	before 2	M
2	Preparation for meeting	after 1	before 3	O
3	Meeting with customer	after 1	before 6	M
4	Fill in loan application	during 3	before 6	M
5	Write meeting notes	after 3	before 10	O
6	Forward loan documents	after 4	before 7	M
7	Approve loan	after 6	before 8	M
8	Return loan documents	after 7	before 10	M
9	Open setoff account	after 1	before 10	M
10	Pay out approved loan	after 9	before 11	M
11	Process loan	after 8	–	M
12	Control loan	after 4, 9	–	O

TABLE 10.3

Dependencies in loan application cases.

achieving support for cooperative work, which is based on two fundamentally different design alternatives.

First, we can try to find a process description that implements the entire logics of a transaction. This description has to anticipate and handle all potential special cases that may occur in a specific situation. Experience has shown that such a complete process description is rarely possible or reasonable.

As a second alternative, we could try to make the cooperation model explicit and editable to coordinate cooperation between different persons. This alternative would let us cover situations that cannot be embedded in a standardized schema.

Process pattern

We opted for the second variant for our T&M approach. We call the underlying concept a *process pattern*, and will discuss it in the following section.

10.4.1 The Concept of Process Patterns

Process patterns represent the cooperation during transaction processing. They can be used by the people involved in completing a transaction. Process patterns can be described and modified, and they make the selected cooperation model explicit.

A *process pattern* is a material representing the regular flow within a routinized cooperation. This material supports the mutual coordination of steps and responsibilities when a transaction case is completed in several steps. A process pattern is based on conventions and the common understanding of all participants. The idea is to use process patterns not only as an explicit model for transaction processing, but also so that they can be modified to reflect current situations.

Process patterns are not designed for full control of a cooperative work process. Rather, a process pattern is a model of the *normal case*, which evolved in the course of practical experience. While a transaction case is being processed, the process pattern also represents the current process state.

First and foremost, a process pattern is a concept. This concept can be realized as routing slips, which will be described in the next section.

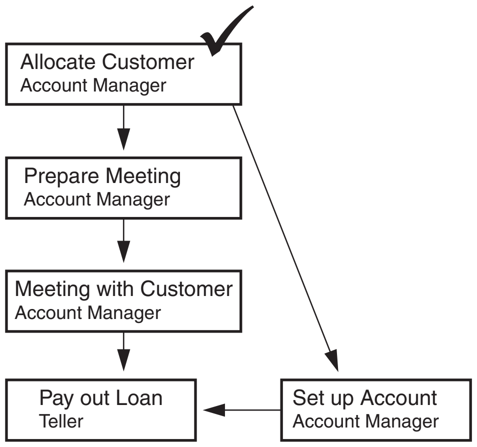


FIGURE 10.5

A process pattern for loan application cases.

Figure 10.5 shows part of a process pattern called `loan application.` Domain relations are defined for some of the activities involved. These dependencies are shown by directed arrows in the figure and should be interpreted in the sense of "should be completed before another activity."

10.4.2 Cooperation Medium: Routing Slips

We have implemented process patterns as routing slips in most of our projects. This is not a new idea, but it is simple and it works. Routing slips are normally used in connection with transaction files, which can be sent to several persons and coordinated.

A transaction file is a special form of dispatch folder and can be sent over a mailing system. It contains all the documents required to process a transaction case. A routing slip can be attached to each transaction file. This routing slip shows who should complete which activities within the transaction case and the receivers of the transaction file.

Cooperation Model

Characterstics of routing slips

In the T&M approach, routing slips are both cooperation materials and working materials. We can identify several characteristics of handling routing slips (see Figure 10.6):

- A routing slip is a work object created by a user for a specific transaction case. It includes information about persons in charge of activities, the flow of work steps involved, and the documents required.
- A routing slip can be removed from a collection of readymade routing slips for routine cases. If a new routine case comes up, then the user can add a prototype routing slip (similar to a template) to the collection.
- Users can change or adapt routing slips to the requirements of specific situations. Naturally, this does not apply to previously completed transaction cases, which are documented and cannot be changed. A user can only change responsibilities (e.g., when a person is on sick leave) and activities (e.g., when the sequence of steps has changed).
- A routing slip is also an instruction to transport the transaction file concerned. Based on the responsibilities noted on the routing slip, the mailing system sends the folder to the workplace in charge.

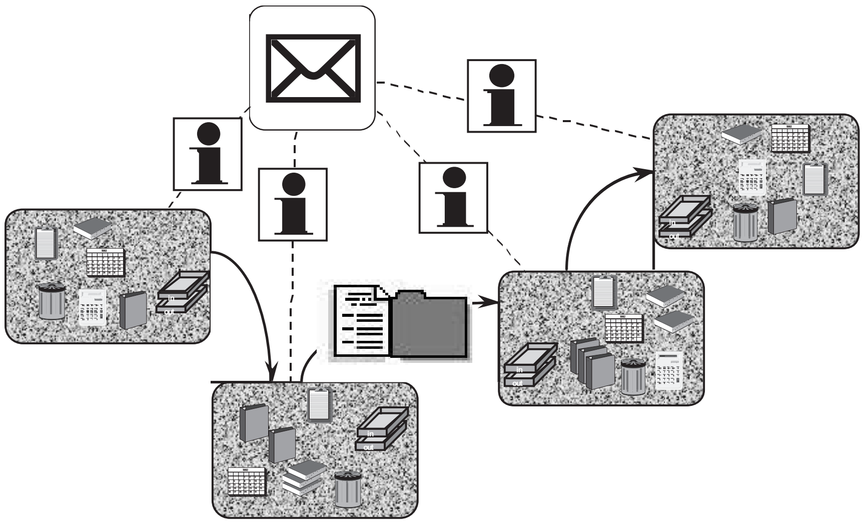


FIGURE 10.6

Using *routing slips* to coordinate cooperation between several workplaces.

- Users can use routing slips to coordinate their cooperation, that is, to inform all participants about the current state of a transaction case. Users can see who completed what task and what remains to be done. They normally tick off activities once they have been completed.
- Transaction files can be traced. For example, a user can request the mailing system to provide information about the whereabouts of a transaction file. In addition, the user can find out what steps within a transaction case have been completed. However, such a request says nothing about whether and how the contents of a transaction file is currently handled at a workplace.

CONSTRUCTION APPROACH

Using a mailing system and dispatch folders

The implementation of routing slips requires little technical effort besides a mailing system and dispatch folders. To function as transaction files, dispatch folders have to be specialized to a minor extent to accept routing slips instead of addresses. As mentioned earlier, a mailing system can be simply expanded by a protocol mechanism to document the transport of dispatch folders. If transaction files have unique identifiers, then these can be built on top of a follow-up mechanism. The question of who may use it and to what extent is not a technical but a domain problem. The desire to follow up on transaction cases should be weighed against the undesirable control of people's work by third parties.

Tracking mechanisms

Modifying routing slips

Similar factors apply to whether or not routing slips should be modifiable. In many cases, the application management has an interest in standardizing transaction cases to encourage uniform business processes. This desire has to be weighed against the flexibility required in processing transaction cases.

Designing routing slips

There is relative freedom in how routing slips can be designed. The only condition is that the mailing system should be able to identify receivers clearly as either people or roles. Sufficient information should also be available for cooperation and coordination, so that a receiver does not first have to check things in a folder. The principle mentioned earlier applies here too, that is that a user is free to see information about the contents and work steps pending completion. A mechanical consistency test between folder contents, routing slip contents, and tools used is not given in the standard version. We recommend this standard version for most application contexts in office environments. On the other hand, the mechanical check of folder contents or tools available at a workplace is very complex and does not contribute much to the routine cooperative work discussed here. The participants are in a position to evaluate these aspects easily themselves. Nevertheless, one of our bank projects required a specialization of transaction files so that they would know the important documents of a loan application. This means that simple consistencies can be checked during a transaction process.

10.5 REFERENCES

A. Krabbel, I. Wetzel, S. Ratuski: *Participation of Heterogeneous User Groups: Providing an Integrated Hospital Information System*. In: Proceedings PDC-Conference, Boston, November 13–15, 1996, pp. 241–249.

A paper on integrating users in the process of selecting standard software and an introduction into using cooperation pictures.

A. Krabbel, I. Wetzel, H. Züllighoven: *On the Inevitable Intertwining of Analysis and Design: Developing Systems for Complex Cooperations*. In: G. van der Veer, A. Henderson, S. Coles (eds.): DIS'97 Designing Interactive Systems: Processes, Practices, Methods, and Techniques, Conference Proceedings, Amsterdam, The Netherlands, August 1997, pp. 205–213.

Here we explain our ideas on developing cooperation support.

W. Prinz, S. Kolvenbach: *Support for Workflows in a Ministerial Environment*. In: Ackermann, M. S. (ed.): Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work, Nov. 16–20, 1996, Boston, Massachusetts, USA. New York: ACM Press, 1996, pp. 199–208.

Paper on advanced workflow concepts.

11

Interactive Application Systems and Persistence

In most application systems, materials have to be managed over an extended period of time. For this purpose, the most common core components of application systems are database systems with their special services, such as efficient data repository, and multi-user and security concepts.

For the T&M approach, it is important to find a domain-motivated architecture to connect an application system to a database system. Based on the T&M concept for workplace types (see Section 3.6), we introduce two important application types:

Persistence and workplace types

1. applications for expert workplaces, and
2. applications for functional workplaces.

From an application-oriented viewpoint, persistence seems to be of minor importance. The reason is that we expect the different technologies available to have little or no influence on the usage model. All we need is an adequate persistence medium to store our materials.

But on closer examination, persistence does matter. First, because a poor coupling of an object-oriented application with a database system will usually lead to annoying performance problems. Second, if we are not careful about the way we integrate a specific persistence medium, this will corrupt our software architecture and we loose important characteristics like structural similarity or maintainability.

This chapter, therefore, addresses readers who as application developers, software architects, or technology experts have to deal with the issue of OO application systems and persistence. Again, we use the structural elements of the pattern sections of this book for better readability of the chapter.

11.1 BACKGROUND: INTERACTIVE APPLICATION SYSTEMS

The architecture of application systems has to embed database systems within a general persistence concept. We will discuss this issue especially from the perspective of application systems developed by the T&M approach. We want to find an answer to the question of how we can overcome the paradigmatic conflict between object-oriented application software and relational databases.

It is important to understand that we do not favor one single solution; instead, we explicitly include the characteristics of an application in our search for a suitable architecture. Based on the T&M concept of workplace types, we introduce the two important application types for expert workplaces and for functional workplaces. For these two application types, we describe suitable solutions, discussing functional workplaces in connection to embedded systems (see Section 3.6).

Domain and architectural contexts

The solutions proposed in this chapter will be discussed on two levels:

1. Domain concepts for cooperative work (multiuser capability) for handling mass data and for working with complex materials are used.
2. Architectural concepts, including client-server architectures for persistence services, loading complex materials on demand, mapping options, and evolution of materials over the application life cycle, and connection to different databases.

Another important question is how far we can utilize the services of database systems (e.g., transaction concept, multiuser capability using locking concepts, load-on-demand in object-oriented databases), or whether we should implement these services in our application. There can be several reasons to opt for the second option, for example, to introduce “higher” domain concepts or a conflict between the object-oriented and relational worlds.

We will characterize the application domain and list criteria that allow us to estimate the cost of implementing a persistence concept early on in our project. Experiences from real-world projects have shown how necessary this is.

11.2 PERSISTENCE SERVICES

Connecting application systems to databases

This section describes two different approaches to connect an application system to a database:

1. Software registry
2. Generic persistence service

A software registry provides a domain usage model for jointly working with material archives at expert workplaces (see also Chapter 10).

The important features of a generic persistence service are efficiency and transparency for functional workplaces and embedded application systems.

11.2.1 Software Registry

Problem

We want to provide a domain-motivated persistence service. To solve this problem, we look for a suitable usage model to file and manage jointly used materials. Based on the T&M approach, we consider proven materials of the application domain and how they are handled, to better understand how we can model our application domain. The usage model developed to file and manage jointly used materials should produce a generic component that can be used in different application domains.

Background

Rather than identifying the requirements of many specific application domains ourselves, we analyze an existing generic concept that is commonly used for archive management—the registry.

Traditional archive management

Though most archive managements are based on outdated concepts, we can use the experiences gained in this field to develop a domain model and a software model for archiving and managing jointly used materials. On the one hand, these models should reflect the familiar use of proven objects and, on the other hand, use the potential of modern software to overcome some of the constraints inherent to physical objects. In other words, we combine a new technology with proven practical concepts.

The domain model has two aspects: In addition to the actual usage model at the individual workplace, we have to consider the cooperation aspect (in the form of a cooperation model) for our registry.

When designing the cooperation model, we have to bear in mind that documents will be used in the form of originals and copies. Hence, we combine the registry concept with the original-copy variant of documents, creating a simple cooperation model that uses the extended capabilities of modern software. The actual usage model takes the use of tables of contents into account, which contribute directly to technical requirements, such as, efficiency.

The Elements

One of the most important notions in managing joint documents is the registry.

A *registry* is the central instance for document management in commercial and governmental organizations. It forms the core of a network of information and interfaces to different independent departments.

A registry records each document once it has been created and filed in the registry. The whereabouts of a document and who removed it from the registry are known at any given time. The registry includes all documents not currently edited but required for transaction cases. To better understand the registry concept, we first explain the tasks assumed by the participating roles.

A *registrar* is responsible for registry management. The registrar coordinates concurrent access to the registry, ensures consistency of the documents managed, serves document requests, and accepts documents for management.

A registrar uses registration means

To meet its tasks, the registrar uses various *registration means*, including:

- *Inventory directory:* This is used to record each document managed in the registry. A document is characterized by information, such as author, registration date, and required retention period.
- *File logbook:* This logbook records all documents distributed. It can use placeholders instead of documents removed from the registry. Each placeholder should show the removal period and the current location.
- *Scheduling book:* The registrar uses this book to enter a date when documents should be resubmitted for follow-up purposes. These dates are normally provided by users.
- *Reservation list:* The registrar uses this list to enter user requests for documents.

Discussion

If we want to draw useful conclusions from traditional registries for our domain modeling work, we have to be aware of a number of seemingly obvious points:

Characteristics of a registry

- The registrar manages a *large number of documents*. We assume that these documents are not transient.
- Documents are managed and searched in file logbooks, which have good structuring characteristics. Many registrars maintain several logbooks in parallel to facilitate access based on different criteria.
- A registry contributes to *implicit cooperation*, based on conventions. This means that other users are not directly visible in the usage model; they can be seen indirectly by the user of a registry. For example, if a user removes a document from the registry, he or she merely leaves a *trace* in the form of a placeholder. The registry does not support explicit cooperation, as in a mailing system (see Chapter 10).
- A registry offers *registration means*, such as placeholder cards, to support coordination.
- A material can be exactly in *one location at a time*. A document is either on a user's desk or in the registry, but never in two locations at the same time. Before a document can be edited at a workplace, it has to be transported there.
- Documents managed in a registry can *relate*, that is, belong to the same business transaction case. To allow joint editing of documents, they are normally held in files or folders. The registry manages such business transaction cases as one *single* object.
- Documents can be copied; an original and its copies are not connected in any way, and are treated as separate materials. Changes to a copy do not affect the original.

11.2.2 The Basic Concept of a Software Registry

Problem

We can develop a matching usage model from our analysis of conventional registries and combine it with a cooperation model to implement it in our application system.

Background

A registry serves two purposes. First, with its registrar as the coordinator, it acts as a cooperation medium. Second, it serves as a managed container in which to easily file and find documents.

One important task is to integrate the possibilities offered by modern software into the design of a software registry. We want to utilize as many of the familiar interactions with a conventional registry as possible, while using the new technologies of modern software. This idea is aimed at designing a simple usage model for a software registry, which is intuitive and consistent.

Solution

A *software registry* is used by a *registrar* to manage a persistent container, or the registry. From the technical view, the registrar is a *service provider*, which supplies a specific *service* based on well-defined rules, that is, to maintain the registry consistently and handle concurrent access. A registry manages mainly documents, but the concept can be expanded to other materials.

11.2.3 Cooperation Model for a Registry

PROBLEM

We are looking for a cooperation model that explains in an understandable way how to use the fundamental means of parallel read-and-write access to a registry. Different variants of this general cooperation model should express the different degrees and kinds of parallel handling. These different model variants should be options offered by a registry component that can be configured to the requirements of a specific application.

BACKGROUND: ORIGINAL AND COPY IN COOPERATIVE WORK

The difference between originals and copies explained in Section 10.1 can be used as a sound basis for the cooperation model we are looking for. We analyze what it means for a document to be an original or a copy within the context of cooperative work. We can basically identify two common meanings of the terms *original* and *copy*, which are closely related:

1. We use the term *original* to define *legally binding documents*, which may be on special paper and include signatures and stamps. There can be several originals (e.g., contracts) or special copies (e.g., duplicates). An arbitrary number of copies can normally be produced from one original, and some copies may be notarized. All of this is done when the copies are issued, and subsequent changes require approval (e.g., by initializing copies). Duplicates are treated like originals. Changes to copies are normally meaningless.
2. We use the term *copy* to describe the *parallel use* or distribution of information. We produce copies of a text and distribute these copies, which serve as working documents. Users can add comments to their copies.

SOLUTION

We distinguish as follows when transferring the concepts of original and copy to a software registry:

Software registry using originals and copies

1. *Exclusive access* to a material for one user; there are *no copies*.
2. *Exclusive access* to an original; *copies* can be made.
3. Access is permitted to *copies only*.

Some application domains differentiate strictly between legal originals and copies, and the way they are handled. Variants 1 and 2 are particularly important for these application domains. Other domains may focus primarily on parallel use of information, so that they require computer-supported concurrent editing of materials. Variant 3 is of particular significance for these domains.

DISCUSSION: EXCLUSIVE ACCESS TO A MATERIAL; NO COPIES

If a material is removed from the registry, then this is obvious to other users, who cannot access it.

- *Modeling reason:* A material should be available and editable in one location only.
- *Application example:* There are things that cannot be simply copied at reasonable cost, or when working with modified or outdated copies can cause

problems. For example, a company's annual financial statement or a credit report should reflect the most current information from the original.

- *Consequence:* Users should be informed about who has removed a material from the registry, so that they can contact the current user, if necessary. Otherwise, there may be interruptions or errors in the work process. To avoid such problems, the registry provides information about the whereabouts of materials removed from the registry. Users can coordinate this process by E-mail or outside the system.

Discussion: Exclusive Access on an Original; Copies Can Be Made

Only one user can remove the original of a material from the registry. Other users are aware of that, but they can produce work copies for themselves. The registrar always keeps a copy of the original, so that more copies can be made and distributed. When the original is returned, it replaces the registrar's copy.

- *Modeling reason:* Work copies should be available even when the original of a material is currently being used.
- *Application example:* It is sufficient in many application cases to just have a look at a material. It wouldn't make sense to prevent this.
- *Consequence:* The users are responsible for coordination of concurrent changes. Changes to the original do not cause automatic changes to its copies. People who use copies can be informed as soon as the original is returned.

Discussion: Access to Copies Only

Each user receives only a working copy with a time stamp. The user may want to return the work copy for the original later on. In this case, the time stamps on the working copy and the original are compared. If the original's time stamp is more recent than that on the work copy, then the original has been modified in the meantime. The user is informed of the conflict and what caused it.

- *Modeling reason:* A user normally cannot have permanent exclusive access to an original (e.g., an account). Especially for read access, many users can have concurrent access. There is no automatic regulation for the small number of conflicting concurrent editing procedures.
- *Application example:* It is usually not a good idea to allow the originals to be turned over to a user when work is being carried out on an organization's customer files. Customer files are frequently provided as copies of currently relevant segments at workplaces. As the work procedure nears completion, it is compared with the original customer files. The user is informed of any discrepancies and must then decide on the basis of his or her professional expertise how these discrepancies can be solved.
- *Consequence:* A user is informed that a conflict occurred and can communicate with other users about how the conflict can be solved.

The term *original* takes on a new meaning when it comes to forms. Forms are distributed and edited as copies of form templates. For the most part, there is no problem with parallel work on a form as long as it relates to different business processes. An editable form template is stored in the forms archive to ensure that only the current version of a form is used once the template was modified.

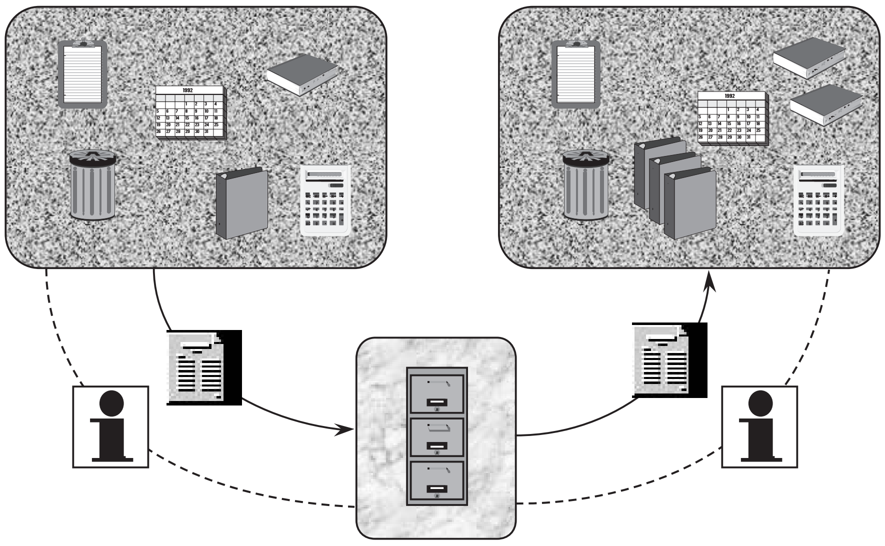


FIGURE 11.1

A registry based on a cooperation model.

TRADE-OFFS

We have just introduced three variants for the cooperation model of a registry. The strategy that a software registry implements depends on the application case.

User feedback

In all three cases, it may be meaningful to allow users to be informed when an original is replaced or updated. For this purpose, the work environment should have an independent generic messaging system, so that users can register for messages from the software registry. In addition, we should allow users to reserve documents.

Section 12.2.5 describes the design decision used in the JWAM framework for the copying behavior in a JWAM software registry. Figure 11.1 shows the cooperation model resulting from these considerations.

In this figure, a user puts documents in the registry from his or her personal workplace and removes documents from there. The registry provides information about users who removed documents. A user can see clearly that another user has removed a document or that another user is editing a document.

EXAMPLE

Returning to our EMS example, the device manager uses the room editor to update the equipment plan, because a new device was purchased for a workplace. For this purpose, the device manager removes the current equipment plan from the registry.

While the equipment plan is on the device manager's desk, the inventory list in the registry includes a removal entry showing when the device manager removed the plan. Users can only use copies of the plan, but they can see that the device manager is editing the plan.

This concurrent situation is characteristic for implicit cooperation. The cooperation is *implicit*, because another user, the device manager in our example, does not explicitly show to the other users. The device manager merely leaves his traces as a competitive user of a material. In this way, cooperation can be coordinated. For example, a user can call the device manager and ask when the equipment plan will be available.

11.2.4 Usage Model for a Registry

Problem

How can we support the management of a large number of documents, while at the same time facilitating the tracking of documents and information about these documents? In line with the cooperation model, we want to integrate the benefits and aspects offered by a software solution into a simple usage model.

Background

Using software to implement a registry has the following benefits:

- Software manages a large number of documents persistently in a minimum amount of space.
- Software lets users quickly access documents, regardless of both the user's and the document's location.
- Software provides tables of contents with several categories, which allow us to expand the wealth of categories by additional information (metainformation) without additional maintenance cost.
- Software can support requests to facilitate locating documents.
- Software maintains consistent storing of documents based on transactions.

Beyond these benefits, however, we also have to consider efficiency, for example, limit our tables of contents to keep them optimal for transmission over a network.

Solution

The registrar works on a registry. A registry is designed as a domain container. Like all domain containers built according to the T&M approach, the registry can create and offer a table of contents for the materials' it manages.

A sniffer

To allow different views on the registry contents or a targeted selection, we introduce the concept of a sniffer. A *sniffer* is an automaton that operates on the registry and accepts search criteria from users. A sniffer creates a directory of the materials' matching search criteria.

Discussion

Using domain containers

To implement the solution discussed in this section, we use a domain container. This means that the registry can be used just like all other containers and as a local workplace registry, independent of the registrar.

The table of contents created by the registry includes a list of metainformation of all materials managed in the registry. Depending on the document types used, we can supply different types of metainformation, so that individual tables of contents can be designed.

Using a sniffer

We use a sniffer to limit large amounts of materials. The sniffer automaton also contributes to efficiency. More specifically, if we have large data repositories in distributed applications, we do not have to transmit the entire table of contents over a network.

Documents are often filed for a business transaction case involving several steps. These transaction cases then, are normally represented in folders. For this reason, the registry should be able to create and find these folders, in line with the general definition of a database transaction. A transaction ensures that several actions on documents are grouped into an atomic unit as a single storage operation, which is then executed entirely or not at all.

COMBINING THE COOPERATION AND USAGE MODELS

When we combine the cooperation model and the usage model, we actually unify two different aspects in the table of contents. The table of contents provides information about the existing documents and about their status of availability, which means that it discloses their whereabouts within the application system. Concurrent use under these two aspects can lead to unexpected behavior on the part of the registrar. For example, while checking the table of contents, a user finds a document marked as available. When she then asks for removal of that document, she finds that it cannot be forwarded to her desk. The reason could be that in the meantime the document was removed *or is being edited* by another user.

11.2.5 JWAM: Architecture for a Software Registry

The registry concept can easily be made available for application development as part of a framework. Figure 11.2 shows what the architecture of a software registry for use in a distributed environment may look like.

Figure 11.3 shows the same architecture at runtime. We can see the components of the registry with a potential distribution of processes. In this example, a tool used by a user does not directly operate on the registry. Instead, the registrar on the server is

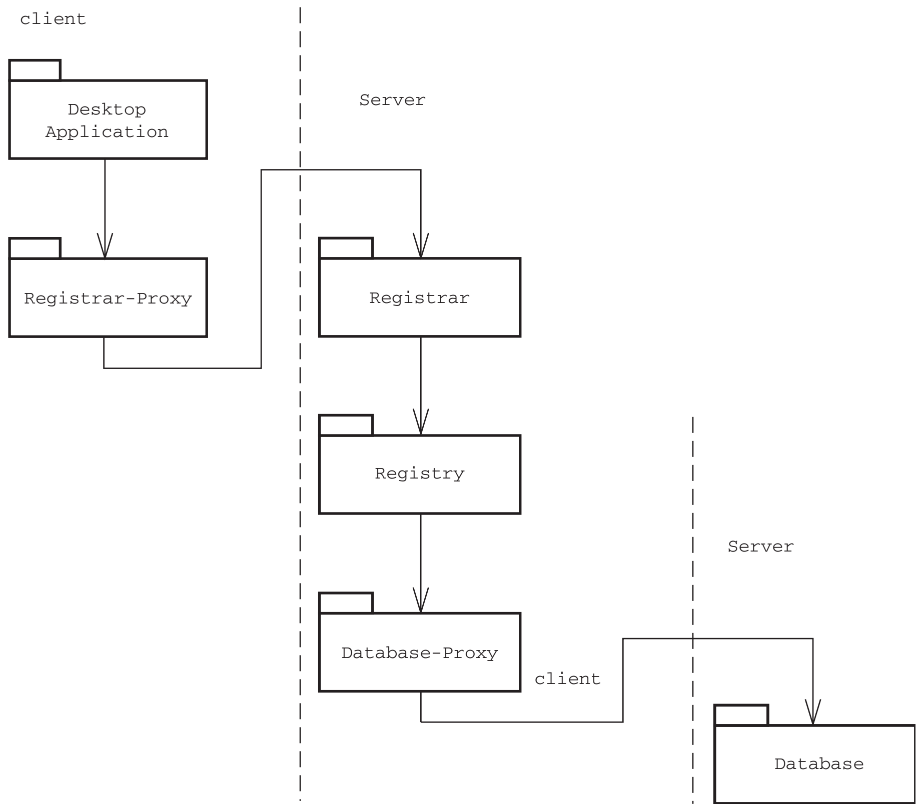


FIGURE 11.2
Architectural structure of a registry.

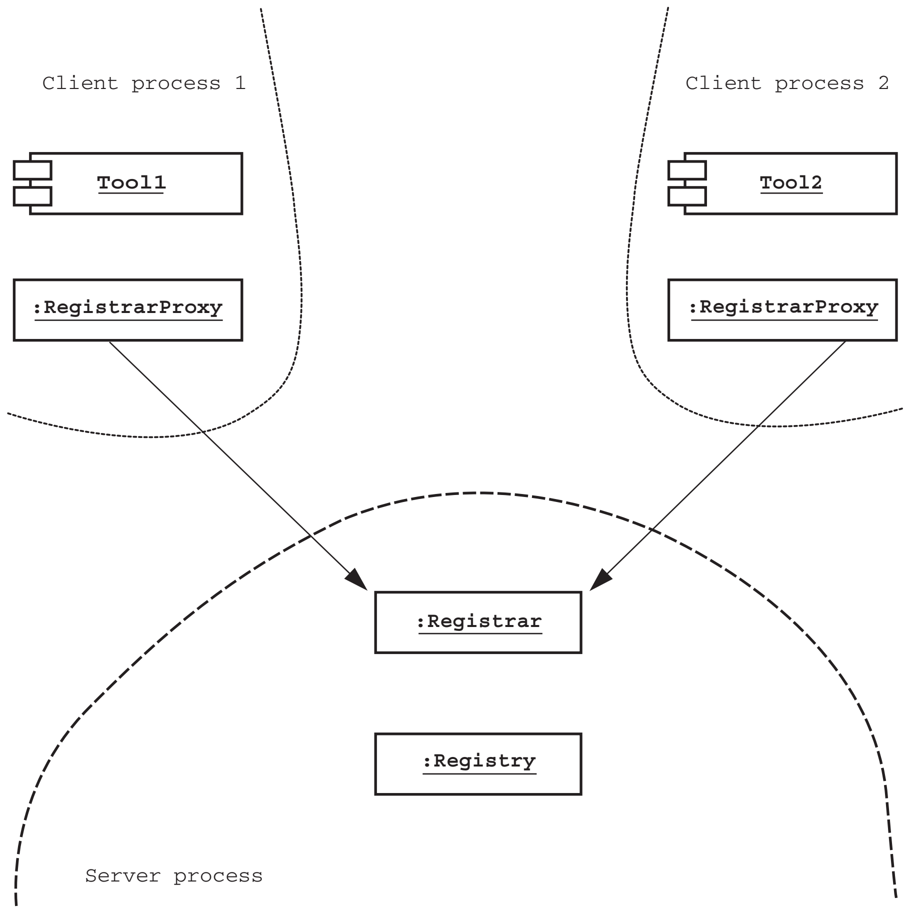


FIGURE 11.3
Cooperation model for a registry.

used through a proxy. Access to the registry is implemented by means of the *proxy* pattern (see Section 8.13).

Notice that we implemented this concept in the handling and presentation layer within the JWAM framework. Though the registry concept does not include any user interface elements, it influences the way in which the system can be manipulated. Figure 11.4 shows the most important classes of the registry concept in the JWAM framework. The framework classes are shown in gray.

As described in connection with the domain cooperation model (see Section 11.2.4), tools do not directly access materials managed in the registry. Instead, they request materials from the registrar. We implemented the registrar as an automaton based on the *proxy* pattern. The local proxy forwards requests for the registrar to the server, where concurrent access to materials is coordinated. The registrar proxy encapsulates the distribution mechanisms, so that tools running in different client processes within a distributed environment do not have to know the middleware used. The way tools interact with the registrar is motivated by the application domain. Figure 11.5 shows a part of the registrar's interface.

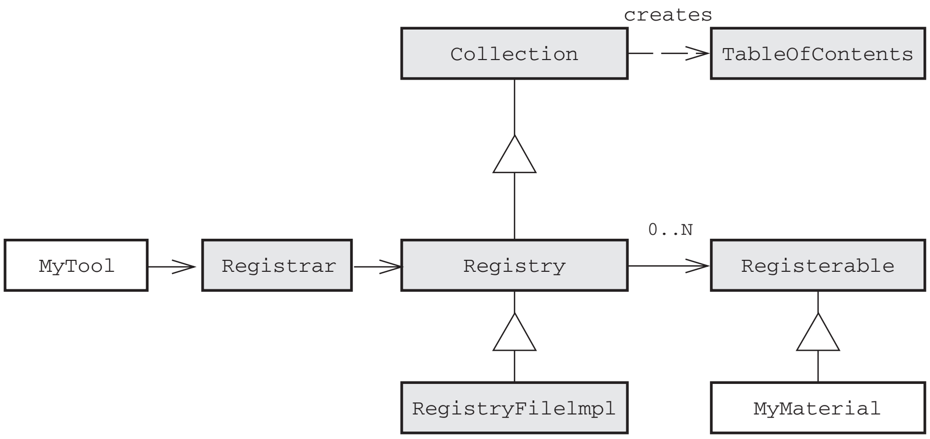


FIGURE 11.4

Classes for a registry.

FIGURE 11.5

Interface for a registrar.

```
public interface Registrar extends Sniffable
{
  public void add (Thing thing);
  public boolean isAddable (Thing thing);

  public void clear ();

  public RegistrationNumberDV registrationNumberByID
           (dvIdentificator id);

  public boolean has (RegistrationNumberDV regNo);
  public boolean has (IdentificatorDV id);

  public boolean isRegistered (Registerable r);
  public void register(Registerable r);

  public void replace(Registerable r);

  public Registerable lastRegistered();
  public boolean hasLastRegistered();

  public Registerable thing(RegistrationNumberDV regNo);

  public void remove(RegistrationNumberDV regNo);

  public dvTableOfContents tableOfContents (Class type);

  public boolean hasMissingCard
           (RegistrationNumberDV regNo);
  public MissingCard missingCard
           (RegistrationNumberDV regNo);
}
```

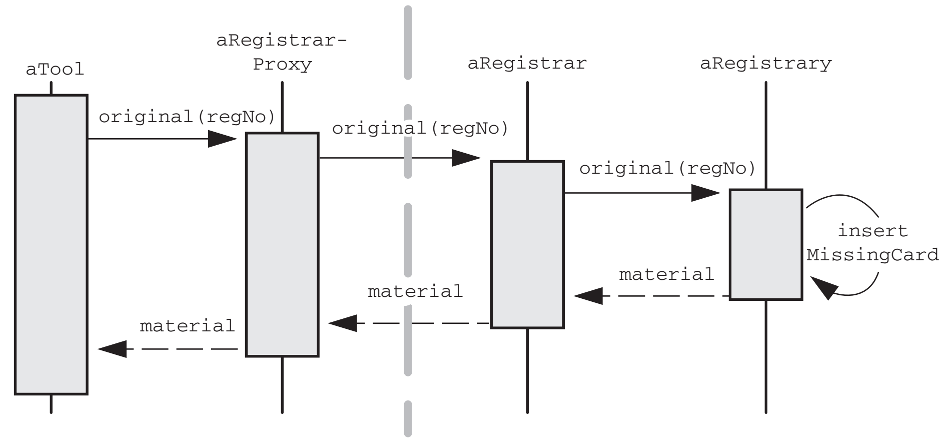


FIGURE 11.6

A registry returns a material.

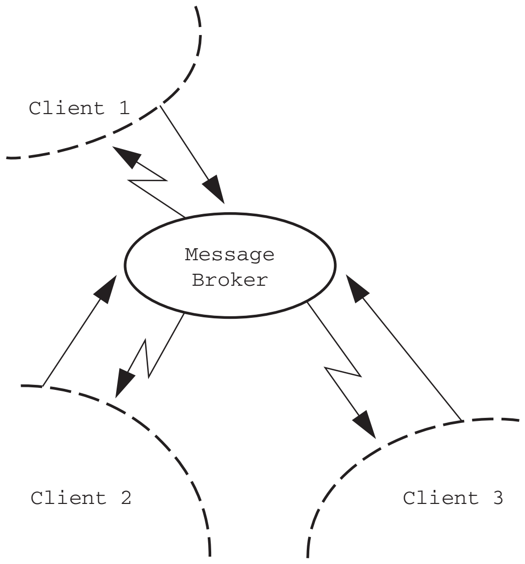
The registrar is the coordinating instance in the server process. Requests from tools running in different process spaces converge here, where they are coordinated by the registrar. The registrar returns materials, registers new materials, and puts back previously removed materials. In addition, the registrar manages the logbook and announces events in the registry to tools. The registrar uses a registry to manage materials (see Figure 11.6).

NOTIFICATION

It may be of interest to some tools to obtain information about actions in the registry. For this purpose, we can easily implement a message broker in the JWAM framework. Tools can register for specific messages with the message broker, so that they will be informed when events matching the requested information occur (see Figure 11.7).

FIGURE 11.7

Exchanging information through a message broker.



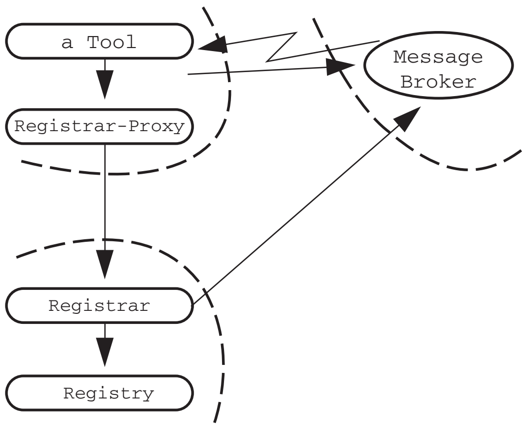


FIGURE 11.8

Runtime architecture using a message broker.

For example, a tool may be interested in knowing when a material is returned to the registry. In this case, the tool can register for the message `"ThingRestored"` with the message broker. In some cases, a tool may not be interested in all message types. For example, a tool that displays a copy may want to reserve the original. It can request a notification of when the original will be available in the registry. For this purpose, messages received can be limited by the use of clauses. In this example, the clause could define a material with a specific registry number. Since these clauses are maintained and checked on the server, this mechanism is effective in minimizing network traffic. Figure 11.8 shows the resulting runtime architecture. Note that the registrar can also send messages.

11.2.6 The Generic Persistence Service

Problem

Functional workplaces are often used to control technically embedded systems. In addition to basic problems that occur when modeling asynchronous processes, these embedded systems normally require high performance and flexibility of the persistence service used. On the other hand, the cooperation model is less important.

The overhead caused by the three-tier architecture of a software registry (client, registry server, database server) can be excessive. Also, the cost involved in making a registry server available can be considerable. We are looking for a concept to implement persistence in technically embedded systems.

Background

Technical persistence concepts

With the software registry, we introduced a persistence concept based on a domain usage model for jointly used materials. This usage model has proven effective in workplaces where users have a high degree of autonomy, such as expert workplaces.

Experience from our real-world projects has shown that this does not represent a universal solution to the persistence problem. Depending on the context (technology, application domain, workplace types), persistence concepts that are more technical have to be used. This applies particularly to the construction of technically embedded systems, such as laboratory information and control systems (see Section 11.4.2).

Functional workplaces

The workplaces we normally find in an embedded system are oriented mainly to the efficient use of functions, with little autonomy for the users. We have defined them as functional workplaces (see Section 3.6). The functional workplace in strongly automated technical processes is oriented to flawless and efficient completion of each task. We will show below that functional workplaces are important in our decisions for the design of a persistence concept.

Solution

We design a generic persistence service for a technically embedded system. The minimum functionality of this persistence service is to file materials over a lengthy period of time, and, for example, to return these materials upon request when the application system restarts. We will implement this generic persistence service as a technical service, operating on materials via a common aspect. This persistence service does not make assumptions about a usage model.

Discussion

The typical interface of a minimal persistence automaton that meets the interface `Storable` and its most important helper classes could look like the code in Figure 11.9.

FIGURE 11.9

Sample code for a persistence automation.

```
public interface PersistenceAutomation{

  void insert (Storable s);
  // stores a new material

  void update (Storable s);
  // stores an updated material

  void delete (Storable s);
  // deletes a material

  java.util.Collection select (Query q);
  // returns a list of materials matching
  // the predicate of the query
}

public class Query {

  Query (java.lang.Object basetype) {/* ... */}
  // returns a new query object; the query is evalutated
  // on basetype and all subtypes

  void addPredicate (Predicate p) {/* ... */}
  // adds a predicate
}
public class predicate {

  predicate (Slot s, Operator o, Integer i)
  {/* ... */}
  // creates a predicate for an Integer field

predicate (Slot s, Operator o, String st)
  {/* ... */}
  // creates a predicate for a String field

  // ... predicates for other database fields ...

  Predicate (Slot s, Storable st) {/* ... */}
  // creates a predicate for a reference field
}
```

FIGURE 11.9

(Continued)

Discussion: Data Consistency

The application systems discussed here are not single-user systems. The generic persistence service is often responsible for the joint use of materials in different work environments. In this case, the minimal interface introduced above is not sufficient. The joint use of materials means that we have to deal with concurrent access and solve the data consistency problem.

The ACID transaction concept of databases appears to be a suitable solution to the consistency problem. Here, the persistence service guarantees the ACID (Atomicity, Consistency, Isolation, Durability) properties of transactions for clients. As in the case of the registry concept, we apply the database transaction concept to containers (see Figure 11.10).

We now change the interface of the persistence automaton as shown in Figure 11.11.

Trade-Offs

The names used for the methods described in the previous section are similar to the (generic) SQL interface of relational databases intentionally. They give us a rough idea of how a generic persistence service for object-oriented systems can be added to conventional database systems in a relatively straightforward way. Other design decisions, including structural mapping, object identification, and so forth, will be addressed further in the following discussion.

In contrast to the software registry, we make no assumptions about how materials processed by the generic persistence service will be handled. There is neither a

```
public interface Transaction {

  void insert(Storable s);
  // marks a new material for persistent storing

  void update(Storable s);
  // marks an updated material for persistent storing

  void delete(Storable s);
  // marks a persistent material for deletion
}
```

FIGURE 11.10

Applying the transaction concept to containers.

```
public interface PersistenceAutomaton {

  boolean execute( Transaction t );
  // executes a transaction; the return value
  // indicates whether a transaction has terminated
  // successfully, i.e. without conflicts

  void reload( Storable s );
  // re-synchronizes the material with
  // the persistently stored vallues

  java.util.Collection select( Query q );
  // returns a list of materials
  // matching the predicate of the query
}
```

FIGURE 11.11

An interface for a persistence automaton.

cooperation model for access to jointly used materials (e.g., original and copy), nor a usage model (e.g., folders and tables of contents). The existence of concurrent users is obvious only where the `execute` method fails or where there is a `reload` method for resynchronization. The only organizational structure for materials is `CollectionOfStorables`, returned by the `select` method; it is the elementary solution.

The software registry requires a domain model that specifies the parts of a (normally complex) mesh of materials, such as assumptions of the software registry about the removal of originals or the production of copies. This is also obvious from the use of folders, which model a group of items that belong to one business transaction case. This means that we can hide the underlying technical transactions from the application developer.

However, if the persistence service we use has high performance and throughput requirements, then we may have to allow "manual" optimization of the technical transactions, for instance to improve the implementation and the conflict resolution.

Let's look at an example. Based on the structure of the application or empirical studies, we assume for the time-critical part of the application system that a very high number of transactions can run without conflicts. In view of optimal performance, it may be meaningful to do without a domain-motivated cooperation mechanism. Instead, we provide for subsequent handling of (rare) conflict cases. This optimistic conflict resolution strategy can be easily implemented in a persistence automaton, such as by calling the `reload` method for materials involved in a conflict.

The generic persistence service is a feasible alternative to a domain-motivated software registry, where performance and flexibility requirements would otherwise exceed the usual limits of interactive application systems. At the same time, a user's autonomy in completing their tasks should be of secondary importance. This applies primarily to functional workplaces in technically embedded systems. Figure 11.12 shows the architecture of a generic persistence service.

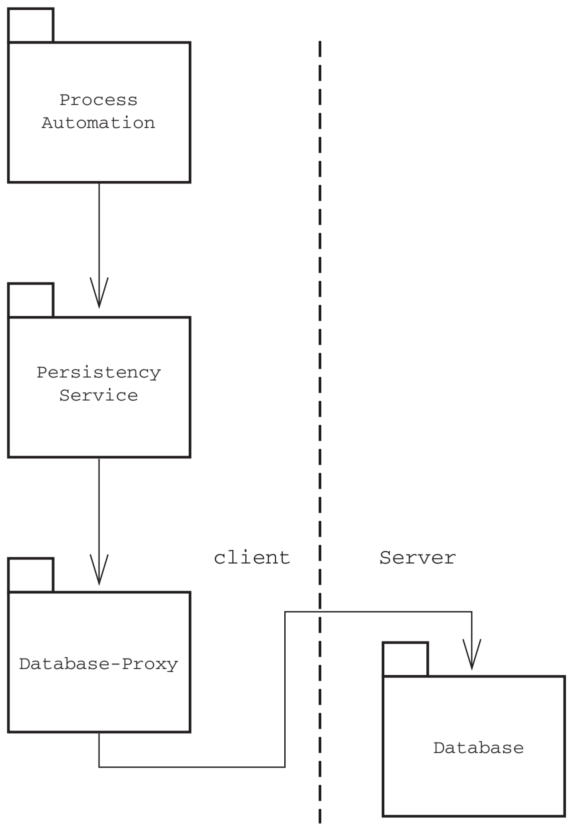


FIGURE 11.12
Architecture of a persistence service.

Notice at this point that software registry and generic persistence service are not conflicting approaches, that is, we could prefer either one, but we could also easily combine both concepts. If we combine them, we obtain the structure shown in Figure 11.13.

11.3 DESIGN CRITERIA TO IMPLEMENT PERSISTENCE

When implementing a software registry or a generic persistence service, we should carefully consider the following points, which will be discussed in the following sections:

- Client-server architecture
- Identifiers
- Technical data modeling and structural mapping
- Evaluation and data warehousing
- Load-on-demand
- Transactions and locking
- Class evolution
- Legacy databases

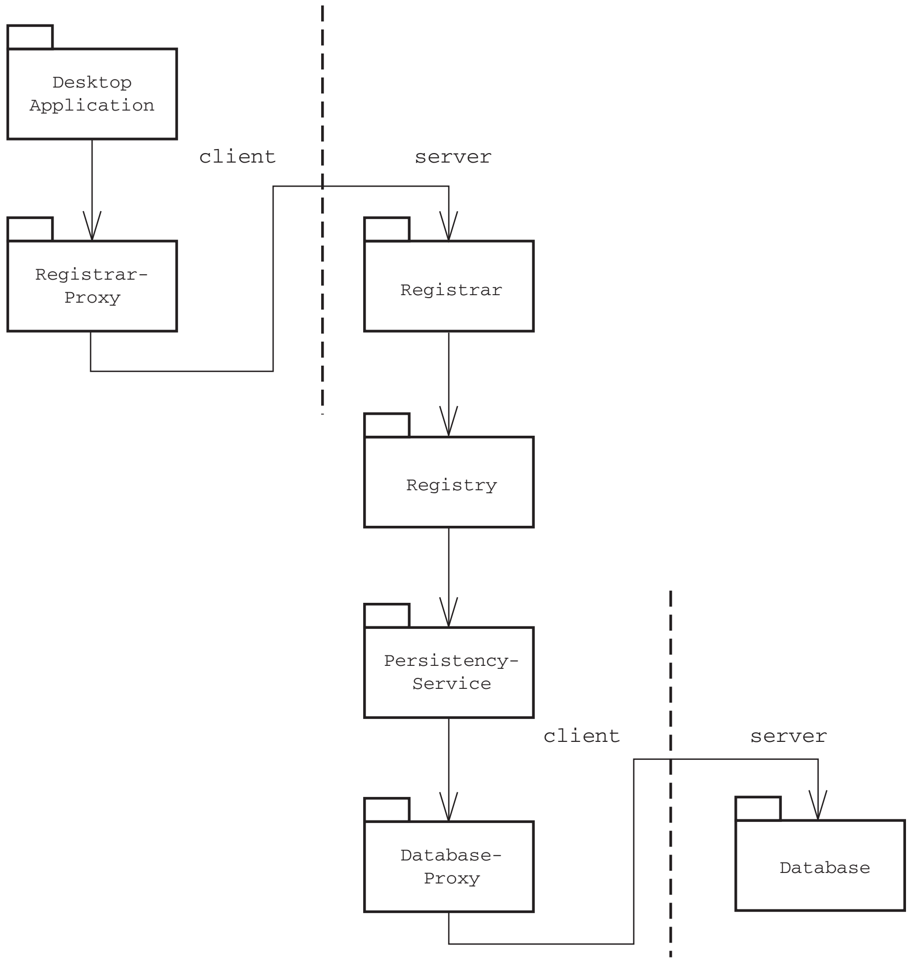


FIGURE 11.13

Combining software registry and persistence service.

11.3.1 Client-Server Architecture

Software registries and generic persistence services are normally used in multiuser environments and implemented over a client-server architecture. The persistence functionality described here can be distributed over clients and a server in different ways. We distinguish between two architectures:

Two architectures for distributing persistence

1. *Persistence-capable client*: In this architecture, the selected persistence mechanism is fully implemented on the client. In the simplest case, a file system (e.g., Network File System (NFS) server of the UNIX file system) or a commercial database server is used as server. The persistence mechanism is implemented using the programming interface of the database (normally in C for relational databases, or Java Data Base Connectivity (JDBC) and C++, Java, or Smalltalk for object-oriented databases).

   A typical application for a persistence-capable client architecture is the generic persistence service. But we can also use it for the software registry.

In this case, the registry server is integrated in the application, and its cooperation model is implemented over data stored on the server. This solution has two major drawbacks: the size of the software to be installed on the clients, and the client configuration. The reason is that drivers for the persistence service have to be installed and properly configured in each client.

2. *Persistence-capable server*: In this architecture, parts of the persistence mechanism are implemented in the application and others on a dedicated persistence server. In turn, this server acts as a client, using the services of another server. This additional server can be a file system or database, as in the above architecture, or the interface to a proprietary host system. Additional interprocess communication is required to let clients communicate with the persistence server. To implement such a communication mechanism, commercial middleware, such as CORBA or EJB (Enterprise JavaBeans) is normally used.

   Obviously, the software registry is a candidate for a persistence-capable server architecture. We implement the registry server and thereby the domain usage model for materials in the persistence server. In contrast, our experience has shown that the persistence-capable server architecture is rarely used to implement a generic persistence service. It is used in rare cases when the server's multiuser capability is limited, as is the case with a batch interface of a host system, so that the persistence service acts as a concentrator.

TRADE-OFFS

Persistence – capable clients

One major benefit of the persistence-capable client architecture is that its implementation is relatively easy. For example, we don't have to implement a separate multiuser-capable registry server. If the main memory of the registry server cannot hold the entire material stock permanently, and if there are high data security requirements, then the persistence-capable client variant is normally performing better than the persistence-capable server architecture. The reason is that the database is accessed without a detour over the registry server. In addition, the registry server has to do a costly things to ensure data security. It normally either uses the database's transaction mechanisms in the sense of a write-through cache, or protects itself directly against potential data loss. Several products introduced to the market more recently (e.g., EJB servers) include mechanisms to implement your own server.

Persistence – capable servers

The major benefit of the persistence-capable server architecture is that it is easy to implement domain cooperation models, such as a software registry. Another benefit is that the persistence-capable server architecture supports lean client applications limited to domain functions. In addition, we do not need the linked code to access a database as we do with persistence-capable clients, which is an important argument in view of limiting data traffic over networks.

11.3.2 Identifiers

Objects in an object-oriented model have a unique identity, regardless of the values of their attributes. The tuples of a relational model, which unfortunately are also often called objects, can be distinguished only by the values of their key attributes.

Object identity is asserted only within a process space in popular object-oriented programming languages, such as by their storage address in C++. If we want to extend the life cycle of a single object or a mesh of objects beyond the duration of the creating application, then we need a mechanism to secure the identity of objects, regardless of their current process space.

One commonly used solution specifies an additional attribute, a so-called object identifier (OID), for each persistent object. Each OID has a unique value across the entire system, which is specified when an object is created, and which cannot be modified. We normally define OIDs in the superclass of all persistent materials.

Implementing unique OIDs

There are several methods for implementing unique OIDs across the entire system:

1. One obvious solution uses a centralized service to assign an OID to an object. We normally use a database to implement this service, because many databases support a mechanism to create unique numbers.
2. Another solution uses random numbers for OIDs. With a current random number generator, the probability that a number will be assigned more than once is clearly smaller than the probability of a programming error that destroys objects.
3. Using random numbers for OIDs, we could optionally add more elements, such as the ID of the network card or a combination of IP address and process ID, to make the OID even "more unique."

Trade-offs

Though the above solution (1) is obvious, it has a number of drawbacks. First, our application programs always have to be connected to the centralized service when new materials are created. Today, this is not the case for an increasing number of systems. The cost to centrally create unique OIDs in large distributed applications can easily exceed the object creation cost, due to network latency, so that it can quickly turn into a system bottleneck. Second, people in distributed applications often have to work offline, that is, their systems are not permanently connected to a centralized service.

Random numbers (2) or combined random numbers and IP addresses (3) are serious alternatives. Java uses a library class (`VMID`) that can create unique numbers without having to access a centralized administrative instance. This mechanism uses a combination of IP address, process ID, and an incremental counter.

11.3.3 Technical Data Modeling and Structural Mapping

When opting for technical data modeling, we have to select a model for the persistence mechanism to store its managed materials. This means that we have to decide how to technically store and reconstruct material information.

There are several methods to address this issue, which will be briefly described in the following subsections.

Using Object-Oriented Database Systems

If we use an object-oriented database (OODB) system, we can normally use the domain class design and the implementation model for persistent objects of this OODB to design our technical data model. Most OODB vendors include special superclasses in their products, from which the classes concerned have to inherit their persistence properties (mix-in inheritance). More recent OODBs include post-processors that derive from the persistent superclass automatically and generate appropriate object code. Such solutions are suitable for programming languages like Java, because Java's bytecode is standardized. This means that we can make third-party classes persistent, even if they are available only as bytecode.

In contrast, if we want to implement the persistence service manually, then we have to convert the complex object structures serially into a data stream, and then make them persistent. Java's `Serializable` is not really suitable for this purpose, because individual handling of the byte stream can be very costly.

Using Relational Database Systems

Most large commercial projects use a relational database management system (RDBMS) to implement data storage. Normally, they opt for an RDBMS for strategic reasons, such as data security or compatibility with existing database applications. The question is how we can map the relationships of materials, like classification, aggregation, association, and inheritance, to the relations of an RDBMS. Several solutions have been proposed for this structural mapping.

Structural mapping

- Not decomposing the object structures at all is the approach that is closest to the object-oriented solutions described in the previous section. Instead, BLOB (binary large objects) fields are used to serially store objects in a relational database. An object's OID can be used as a key to retrieve that object. In this solution, the relational database simply acts as a file system that facilitates object retrieval extended by a transaction concept and other features (e.g., online backup, transaction logging, hot standby) for data integrity. Besides the BLOB entry, such object attributes are duplicated in table rows by individual programs that should be used by database queries.
- Mapping classes to relations is the method more frequently used in a relational database. The attribute values of the objects of a class are stored in the fields of the corresponding relation, with the following important details:
  - Associations such as use relationships between classes are mapped by means of foreign keys, maintaining the structural similarity.
  - The relational model supports as the types of fields of a relation only standard data types due to the first normal form. This means that nested table structures are not permitted. Class aggregation, or the structural composition of classes from attributes, which are themselves classes, cannot be mapped in RDBMS without losing structural similarity. Instead, we can use associations to simulate aggregation relationships.

Handling Inheritance in Relational Databases

A very interesting question is how we can map inheritance relationships between classes to relations. There are several solutions to solve this problem:

One relation for each class

- The most intuitive solution creates a relation for each class of the class hierarchy. Next, the attributes of the superclass are exclusively stored in the superclass relation and those of the subclass exclusively in the subclass relation. The attributes of an instance of a subclass have to be distributed over several relations. The OID serves as a common key to reconstruct a specific object from the database, using a join operation over these relations. The major benefit of this solution is the nonredundant storage of attributes. Another benefit is that superclasses can be queried efficiently.

One major drawback of this solution is that we always have to run a relatively expensive join operation when reconstructing objects from a subclass. However, we can reduce this cost, for instance, by optimized stored procedures. Another drawback is that, when storing subclass objects, there are several relational insert operations, depending on the hierarchical depth.

Replicating superclass attributes in sub-class relations

- An alternative solution also creates a relation for each class. However, in contrast to the above solution, the attributes of superclasses are replicated in the relations of subclasses. This avoids the cost introduced by join operations. On the other hand, this solution requires more storage space due to redundant attributes. In addition, each new or modified object can cause a number of relational insert or update operations. Creating subclass objects and modifying their attributes at runtime can be relatively expensive.

  We can see that, although this solution replicates attributes in the relations of subclasses, it stores objects in the relation allocated to the creating class. The major benefit of this model is that expensive joint operations are avoided. Another benefit is that we can also use simple insert or update operations to create and modify subclass objects. One major drawback of this solution is that all polymorphic query operations, where the class/relation to be selected is not static, must walk through all relations in the class hierarchy.

TRADE-OFFS

Which of these solutions we should use in a specific project depends primarily on the (expected) way that classes or relations should be accessed at runtime.

Commercial products more recently introduced to the market support database mapping for a number of object-oriented programming languages and relational databases. Most of these products use libraries and tools to support mapping. The best form of structural mapping depends largely on the application's characteristics. This means that, when selecting a commercial product, we should carefully consider whether or not the selected product includes the preferred mapping form at the required performance.

Though structure-preserving mapping of classes to tables represents a good basis for relational data modeling, it normally requires some improvements. In general, very small classes (e.g., domain value classes) are embedded to increase performance.

At least in an initial project phase, or when there are no strict requirements for data volumes and throughput, a project can often do without mapping tools. Instead, we can write special mapper classes ourselves. These mapper classes write objects into the relational database and reconstruct these objects on retrieval.

In contrast, projects with demanding requirements to data volumes and throughput should normally consider the structural mapping problem as a major risk, which should be dealt with at an early project phase.

11.3.4 Querying and Data Warehousing

Querying operative data

One of the most important aspects of designing large application systems are the expected requirements of users with regard to the querying of the operative data created and managed in the application system. Most practitioners assume that the operative data storage should ideally not be used directly for quantitative evaluations, because it is normally not optimized for set-oriented queries.

In a worst-case scenario, expanding the operative data model by redundancies to optimize queries can cause a total loss of domain-related structural mapping of the material design. If this happens, it seriously hinders the maintenance and further development of our software system.

Data warehouses

To solve this problem, we can use a *data warehouse* with separate data storages, where the data structure of each separate data storage is optimized to the expected evaluations. These data storages are synchronized at specific intervals (e.g., every night, weekly, etc.).

Since data warehouses are often used in combination with SQL for ad-hoc queries, it can even happen in object-oriented application systems that an OODB is used for the operative data, while the data storage for queries is transferred to a data warehouse based on an RDBMS.

11.3.5 Load-on-Demand

An object mesh can become very large so that, when using an RDBMS, it may be stored over several tables, which can cause inacceptable load times. In most cases, however, only a part is needed, rather than the entire object mesh. Figure 11.14 shows an example of such an object mesh.

If we wanted to edit `Customer` in this example, we would load not only `Customer`, but all of its orders and all articles referenced in these orders.

Using load on demand

To solve this problem, many OODBs offer a mechanism to *load objects on demand*. This technical mechanism works very well for many application cases, but can cause considerable problems in others, such as workplace systems. For example, a user can remove materials from a registry and place it on his or her personal desk. To ensure clean implementation of this concept, we have to define material boundaries. For example, if we use the original-copy concept, we have to lock the entire material mesh rather than only the part actually loaded into the client's main memory. Another problem occurs when a registered material references another material in the registry. If we use the load-on-demand mechanism, the referenced material is also loaded automatically on demand from the database. However, the registrar does not know about this access, so it cannot update its management information.

Not all OODBs are capable of implementing such a load-on-demand mechanism in multi-tier applications. Assume, for example, that part of a material is transmitted

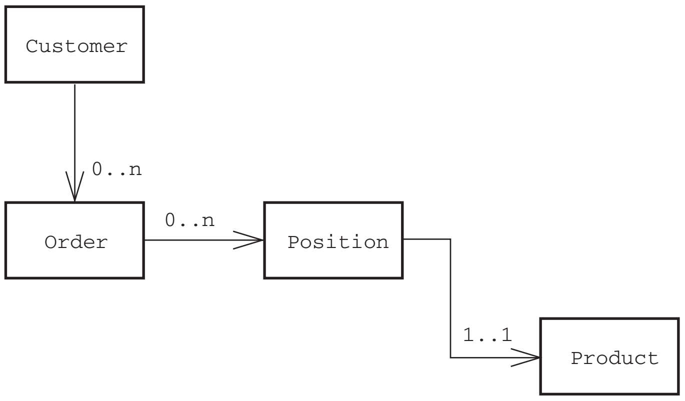


FIGURE 11.14
Object structures.

from an application server to a client. Let's further assume that a user activates a load-on-demand reference in this material. Since neither the database itself nor the drivers for accessing this database reside on that client, this access leads to an error.

TRADE-OFFS

Problems of load-on-demand

All of these problems can basically be solved. However, the cost involved is normally very high, at least if workplace systems have to be supported. It is often better not to use a technical load-on-demand mechanism and instead consider domain solutions early on in the modeling phase.

This means that we have to define clear material boundaries. In this respect, we can define two different types of references between materials:

References between materials

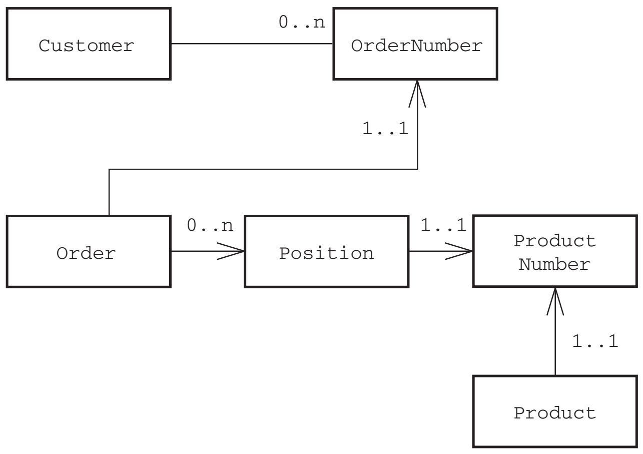
1. *Strict inclusion relationship*: One material is completely included in another material. There is no other material containing the same material. We can choose the use relationship between objects to model this kind of inclusion.
2. *Loose referencing*: To model this reference type, we use domain identifiers instead of technical object references. These identifiers are normally expressed as domain values or OIDs. When a tool that handles a material finds such a domain reference, then the tool has to resolve it directly. For example, the tool can use the identifier to request the referenced material from the software registry. The tool has to handle a situation where the requested material is not available, either because it is being used or it was deleted.

A simple example

One good example for using identifiers for domain references is a simple order-processing application: a customer object knows its orders. Each order includes a set of positions, and each position references a product. Between order and position, there is a strict inclusion relationship, which is modeled as an object reference. The situation is different between the customer and its orders, and between positions and products, that is, we use domain references. More specifically, we use order numbers for the first case and product numbers for the second. This example shows that we often find domain references used in real-world applications to specify loose referencing. For this reason, it is often a good idea to use the domain references of the application domain in system modeling, at least in the beginning. Figure 11.15 shows a class diagram of the above example.

FIGURE 11.15

Sample classes.



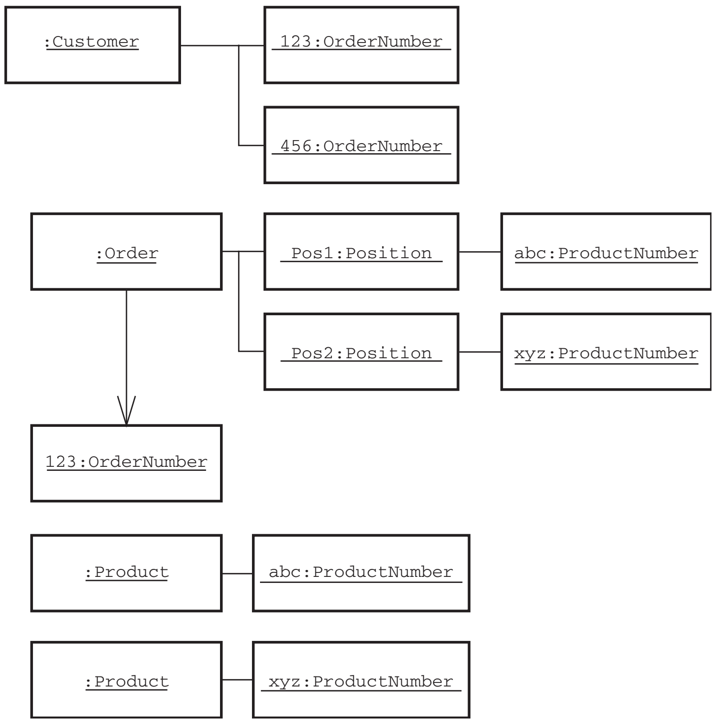


FIGURE 11.16
Sample objects.

At runtime, the example of Figure 11.15 results in the object diagram shown in Figure 11.16.

At first, this type of modeling with domain references may not appear object-oriented. Notice, however, that it provides a simple solution to the problems relating to load-on-demand, granularity of database locks and material transport over a network.

11.3.6 Transactions and Locking

Transactions

All databases include access control mechanisms; most use transactions and locks. A *transaction* encapsulates a number of database operations, which have to be executed jointly in each case. The database guarantees that the operations belonging to the same transaction run either together or not at all. *Locks* can be used to lock single units of a database (e.g., relations in an RDBMS or objects in an OODBMS). We normally distinguish between different locking strategies (optimistic, pessimistic) and lock types (read lock, write lock).

Using a database automaton

We have to determine the view we want an application developer to have on a persistence medium. If we use a database automaton (see Section 11.2.6), this view will be close to the database, which means that transactions are visible in the interface of that database automaton. This technical proximity to the database offers the benefit that its implementation is easy. The downside is a closer coupling between the database automaton and the persistence service. For example, if we want to replace the transaction concept in the database by a persistence service that doesn't support a transaction concept, then we have to implement the transaction concept as a part of the automaton, which introduces considerable cost.

Using a registry

Using a registry means that the transaction and lock concepts will be hidden from the application developer. In this case, application developers would use domain concepts (e.g., original-copy). Mapping these concepts to technical database features is not always obvious, but it is relatively simple. For example, a persistence medium without transaction support could easily be replaced by another one that does support transactions. This means that we could work with a prototype on top of the file system, which could then be replaced by a relational database in the productive system.

11.3.7 Class Evolution

Another important persistence aspect in object-oriented application systems is *class evolution*. In fact, one major benefit of object orientation is that the internal representation of a class can be changed easily without affecting the rest of the system.

As soon as objects are stored, we will have to deal with problems when we attempt to further develop our system. The reason is that if only the class definition is changed, then the modified class definition no longer matches the stored objects. This means that it is not sufficient to continue developing only the class. We also have to convert the data stored in objects. Though database structures can be changed manually, and special conversion programs can be written, both methods are expensive. In particular, this approach destroys a major benefit—easy modification. We have frequently observed that a project's productivity decreased considerably with the introduction of a database.

Consequently, we are looking for general mechanisms to support the evolution of persistent data. For this purpose, we could increase the functionality of a database automaton or of a registry. Either of the two would then verify upon system start whether or not the persistent class structure still matches the current class structure. If the two structures don't match, then our mechanism should adapt them automatically.

Java is one of the programming languages that include mechanisms for such an automatic conversion. However, they work only provided that we use the standard serialization. This means that it is suitable for storage in files and database BLOBS, but not for a full mapping on an RDBMS. Most OODBs offer conversion mechanisms similar to those of Java.

11.3.8 Legacy Databases

We seldom develop object-oriented software projects out of the blue. In fact, we normally find an existing IT environment that cannot be replaced from one day to the next. In addition, object-oriented and conventional systems will continue to exist for quite a while. This means that we have to link object-oriented and conventional applications to prevent an unhappy coexistence of two different applications for the users. There are two basic options for dealing with this problem:

Co-existense of OO and conventional systems

1. Object-oriented programs access existing data storages directly, bypassing legacy applications.
2. Object-oriented programs use existing database transactions as services to write and reconstruct data to/from existing databases.

Trade-Offs

The major drawback of the first solution is that consistency checks are redundant and may have to be maintained twice, that is, once in the object-oriented system and once in the legacy applications. Another drawback of this solution is that consistency checks often exist only implicitly in code in legacy software, and developers are not aware of it. This means that the consistency checks have to be reengineered from the code, which is expensive.

Consistency checks are normally maintained when an object-oriented application calls a conventional program. However, many conventional applications are not designed to be called by other programs. Often there is no way to pass parameters when calling a program. The reason is that domain functionality and dialog control are often strongly intertwined in these applications so that user interaction is required to let an object-oriented application control such a program. In many cases, conventional applications have to be refactored to a pure API.

The road we want to take depends largely on the size of the legacy applications and databases. Our experience has shown that direct access to legacy databases is meaningful for small systems, while calling conventional programs appears to be the better solution for large systems.

Regardless of whether we access data or programs, legacy databases are normally not sufficient to store all data required by an object-oriented application. It has proven to be a good idea in many projects to build a second database for the object-oriented system, in addition to the legacy database. This approach allows us to separate legacy and object-oriented application on the data level, which facilitates our development work. The additional cost introduced by this approach is normally acceptable, as long as there is no requirement for redundant data management. This means that this approach is useful particularly when data has to be copied in one direction only.

Finally, we also have to bear in mind that each change to jointly used data storages will cause additional cost for coordinating developers of conventional and object-oriented projects.

If the existing application systems are host-based, then we can store centralized and decentralized data in separate databases. In many cases, we will want to maintain object-oriented data in a decentralized location, such as on departmental servers, while keeping the legacy databases on the host.

11.4 REAL-WORLD EXAMPLES

This section describes real-world experiences with persistence, discussing some of the solutions that we implemented and the reasons why we opted for one solution over others.

11.4.1 JWAM Projects

We developed and supervised a number of projects based on the JWAM framework. This section describes our experience gained from linking applications to a database. Most applications were basically implemented as expert workplaces, supporting users in typical office work:

- search and edit documents;
- print documents; and
- respond to customer inquiries (e.g., on the phone).

In these projects, the major challenge was to design flexible workplaces supporting different work styles and customer requirements. Though we had to transfer data from legacy systems, there was no need for parallel operation and inclusion of the existing database structures, because new system parts were added to the legacy applications.

Though the persistence media used held large data storages, we observed that they were not frequently accessed. Normally, a user searches the database for a specific material and moves it to his or her desktop. The user then keeps the material over a lengthy period, editing it locally, to eventually return it to the database. This meant that we did not have to meet high performance requirements. It also meant that we were able to use a registry and did not have to worry too much about linking to a database.

Discussion: RDBMS Mapping

Many people argue in favor of using an RDBMS by saying that it can be easily combined with third-party products. For example, you can implement report generators directly on top of RDBMS structures. This is an enticing option, especially in Java projects where printing is supported in a rudimentary way. For this reason, we decided on complete RDBMS mapping in one specific project.

In the course of this project, however, this approach caused a number of problems.

Problems with RDBMS mapping

- For nontrivial materials, complete mapping leads to extremely complex structures in the database, so that loading materials slowed down dramatically. We then implemented load-on-demand mechanisms manually, because RDBMS does not support such mechanisms.

  Unfortunately, the load-on-demand mechanism we implemented led to other problems. Materials were loaded directly into the main memory, bypassing the material manager, which meant that we had to integrate an additional object management into the work environment. When a tool requests a previously loaded material from the material manager, then this material will not be retrieved from the database again; instead, only the reference to the previously loaded object is returned.
- The idea of being able to directly print from within the database turned out to be an illusion for several reasons.

  First, print reports now depended on the internal structure of materials. Every time the internal structure of a material changed, we had to adapt the reports involved.

  The database structure generated automatically was not really suitable for print list formatting, which means that it became unnecessarily complicated to create print lists. Eventually, we even had to expand our mapping to store information required for printing only to the database.

  In workplace systems, users frequently want to print materials that they handle locally on their desktops. Local materials are normally stored in a different way, compared to materials fully residing in the common registry. In the project discussed here, it was sufficient to store local materials in the file system. Only materials residing in the registry on the server were stored in the form of BLOBs in a dedicated database. The result was that materials currently on local desktops were not included in the print lists.

- Changes to materials led to even bigger problems. For example, we had to update the database structure for almost every change to a material. Initially, we tried to update it manually. However, we soon found that this was impractical. To be able to read objects from the database, we had to store administrative information together with the user information, which meant that all this information had to be updated. Automatic conversion was not possible, because the database used in the project did not support the SQL features required. Eventually, we found that the link options of Java's JDBC differed a lot, depending on the database vendor.

Discussion: Using OODBs

Problems with using OODBs

In the further development of the project discussed here, we evaluated several OODBs, only to find that, instead of the problems just described, we had to deal with new problems.

- All persistent objects must inherit from a "persistent" superclass. Unfortunately, Java allows only one superclass, and our domain materials already inherit from a superclass of the JWAM framework. For this reason, many OODBs include post-processors that let you manipulate the bytecode created by the compiler; for example, you can subsequently derive the material superclass of the JWAM framework from the OODB class `Persistent`. We then observed that the performance of each OODB we tested differed considerably. For example, only one of the post-processors we tried was able to convert the Java Development Kit (JDK)-defined classes (e.g., `Date`, `Vector`, `List`) into easily storable equivalents. In the other products, we would have to replace all container attributes by OODB classes. Naturally, this would have meant that our material model would depend on the OODB we use.
- Load-on-demand worked in a multi-tier environment only in one of the products we evaluated. And even with this product, we would have had to install the OODB drivers on each client workplace.
- Similarly to the RDBMS mapping, the load-on-demand mechanisms of the OODBs we tested bypassed the material manager.

Solution: RDBMS and BLOBs

For this project we eventually selected a solution that stores materials in the form of BLOBs in an RDBMS. More specifically, we created a separate table for each material type. This table included a column for material object IDs (OIDs) and a column for material data BLOBs. Java serialization is used to write and read material data to/from the BLOB field. This method allowed us to use the Java mechanism to convert objects for class evolution purposes. In addition to these two columns, the table included other columns for duplicated entries of material attributes, depending on the material type. These columns serve to search and select materials. Their values can be calculated from the material BLOB whenever necessary, and they are not required for loading.

This project showed that only a relatively small part of the material attributes is actually required in search-and-select activities. Consequently, the storage overhead caused by duplicate search attributes was not critical.

Eventually, we opted for the use of a report generator, operating on relational structures, to print materials. We used CSV (comma-separated values) files to define these structures. In this respect, we found it extremely useful that this data is written but not used to load materials. It meant that we needed only a small amount of administrative information.

Discussion: Experiences from JWAM Projects

This section summarizes the results gained from our JWAM projects.

- The expert workplace type has modest requirements for the performance of a database. The use of a registry has proven to be a good idea for this system type.
- You can replace the persistence medium used by the registry at little cost. In fact, the registry can encapsulate the complete persistence mechanism that we use. A simple solution to store objects (BLOBs in an RDBMS) is sufficient in most cases.
- In most cases, it is not recommended to use a separate database for different purposes (e.g., for material storage and printing).
- Class evolution can cause problems in application programming, unless we use a suitable mechanism to adapt the database structures.
- Many application projects use only a small part of the database functionality, such as store, load, and search. The use of transactions and locks directly by the registry is limited. The generic load-on-demand capabilities available in a database are not useful when we develop multi-tier systems with private work domains.

11.4.2 MedIS and SyLab

Background

MedIS GmbH is a German company that specializes in measuring instruments for cardiovascular diagnostics, using the latest technology. They are leaders particularly in the fields of impedance plethysmography, impedance cardiography, and photoplethysmography. The company sells diagnostic devices for doctors, as well as solutions for researchers and equipment manufacturers. The software for the company's laboratory information and control system (LIS) was developed with the T&M approach. The LIS SyLab system developed by MedIS is used primarily for the following tasks:

Tasks covered by SyLab

- A large number of analytical orders (approximately 40,000 to 50,000 per day) have to be acquired by use of different acquisition media (e.g., document scanners or digitizing tablets) within a short time (maximum four hours). These entries include patient information (e.g., name, date of birth, health care system, and clinical data) and are supported by OCR (optical character recognition) technology.
- Small bottles with blood or other body liquids are transported automatically from the sampling station to the appropriate analytical automatons.
- The analyses in these automatons are automated, but still controlled and monitored by people.
- The system supports laboratory staff and physicians in validating and interpreting the lab results. Finally, the system outputs results and bills.

Since all samples are transported under real-time conditions, and the analytical automatons have to be controlled and monitored, SyLab can be classified as a technically embedded system.

We used UNIX derivatives, that is Solaris and Free-BSD, to write the entire code for SyLab in C++. The development took almost fifty person-years and a period of five years. SyLab currently includes approximately 2,000 classes, distributed over a number of libraries in the underlying T&M architecture, technical and domain services, and approximately thirty different user applications. In addition, the system uses more than fifty different background programs to control the technical components connected to it. SyLab uses the relational database system Sybase as its persistence medium, and currently operates in the largest installed configuration, including 100 connected workplace computers and application servers for almost 200 distributed background processes.

Most SyLab workplaces are designed for efficient completion of functions (within the company's laboratory operations), giving their users relatively little autonomy over the way they complete their tasks. Most SyLab users are highly qualified staff, such as laboratory staff and physicians. The special characteristics of functional workplaces have a great influence on the decisions made for their implementation.

DISCUSSION: CLIENT-SERVER ARCHITECTURE

The client-server architecture of SyLab is primarily based on the relational database system Sybase, assuming the role of a server and several user applications or background processes as clients of the database server. The persistence functionality of all SyLab programs is implemented exclusively on the clients, using the generic persistence service described in Section 11.3.1. In turn, this service directly uses the database's API. There is no separate persistence-capable server to connect the database, such as a software registry. By our definition, this means that the SyLab programs are persistence-capable clients.

Using persistence-capable clients

We chose this approach both for its conceptual and technical aspects.

- Within the SyLab system, materials are handled either in asynchronous background processes, for instance by device drivers for analytical automatons, or by user programs of functional workplaces that operate in a set-oriented way. For example, most functional workplaces use statistic quality control to monitor and visualize the operation of analytical automatons. Subsequently, entire blocks of results validated in this way are released. It's difficult to find a domain-oriented cooperation model, such as original-copy, for this type of handling materials, and a cooperation model would be meaningless for background processes. In this project, we could not model private workplaces or conflict resolution strategies for qualified user actions. Users at functional workplaces normally handle large quantities of materials. The explicit representation of the underlying cooperation model and the user interactions related thereto, such as to handle originals and copies, is usually considered too ineffective or even disturbing in the work process. Instead, the cooperation concept implemented in SyLab is based directly on the state model (object-life cycle) of materials. A material can be changed by a background process or functional workplace when it is in a permissible state within its life cycle. In any other state, the material can only be probed, or may not even be accessible.

- On the technical side, efficiency is the main reason against the use of a three-tier architecture with a persistence-capable server in the middle layer. For example, the sheer quantity of analysis orders processed, as mentioned above, leads to extreme requirements for the performance of the persistence mechanism. A more or less rigid real-time control of technical components, for example, to transport samples and control analytical automatons, increases these performance requirements by another order of magnitude. These are the reasons why we chose persistence-capable clients to directly access the database server, especially for the components of the SyLab system with extreme performance requirements.[1] Notice that this decision also determines the persistence mechanism of all other components. For example, it wouldn't make sense to implement a three-tier architecture with a persistence-capable server in the middle when individual persistence-capable clients access the database server directly, bypassing the persistence-capable server; thus all cooperation concepts reside in the middle tier.

DISCUSSION: IDENTIFIERS

An efficient and highly available service to create unique object identifiers (OIDs) is important for the performance and operation of the entire system. If this service is not implemented efficiently, it can become a bottleneck for all applications that create persistent objects, due to frequent access.

In the SyLab system, this includes applications for order acquisition and result-producing background processes as well as various applications that store persistent complex object structures (e.g., job lists) to manage the actual transaction data.

Service availability

If we cannot guarantee high availability, then all applications have to implement special and normally costly mechanisms to cope with a service failure. Due to the normally related logical and technical problems, this additional cost is not justified. Our experience has shown that it should rather be invested in service availability.

For the SyLab system, we chose a very simple, performing implementation of a service to create OIDs. More specifically, a table called `ID` with only one integer field, `nextId`, was created on the database server. The field `nextId` is initialized to `0`, and subsequently contains the OID most recently generated. The ID table is accessed exclusively through the generic persistence service which, in turn, calls an SQL transaction, `sp_nextId`, encapsulated in a stored procedure. To increase performance, we previously created and precompiled this transaction on the database server:

```
CREATE PROCEDURE sp_nextId @nextIdParam INT OUTPUT AS
BEGIN TRAN nextIdTran
UPDATE id SET nextId = nextId + 1
SELECT @nextIdParam = (SELECT nextId from id)
COMMIT TRAN nextIdTran
```

The generic persistence service uses the stored procedure, `sp_nextId`, implicitly whenever a new object is stored in the database. From the perspective of a developer

1. Since even the best performing database server cannot guarantee response times in multiuser operation, the generic persistence service may be used only in asynchronous mode to guarantee the required response time.

of a SyLab client, the use of a stored procedure is a private implementation detail of the persistence automaton, that is, it is not visible.

DISCUSSION: TECHNICAL DATA MODELING AND STRUCTURAL MAPPING

When we decided to use a relational database system, in addition to the object-oriented approach, for the SyLab project, we not only considered strategic factors (compatibility to existing applications, license agreements). In fact, when we started this project in 1994, we found that the existing object-oriented databases were not sufficiently stable and performing.

Consequently, we had already identified the so-called structural mapping problem upon the project is start. The question was how the structure of persistent classes in an object-oriented system could be mapped to relations of an RDBMS.

Also, the pilot customer defined the additional requirement to use SQL queries to access all data in the SyLab system, without having to first use a data warehouse. For this reason, we had to do a relational decomposition of the class structures. We couldn't use a standard serialization or BLOB fields. Instead, we implemented a model that mapped the material classes defined in the SyLab system one-to-one to corresponding relations of the RDBMS. In this model, we used foreign keys to map associations between classes in structural similarity, while associations were used to simulate class aggregation.

BACKGROUND: THE MATERIAL STRUCTURE OF THE SYLAB SYSTEM

To explain the selected structural mapping, this section briefly describes the technical class hierarchy and the object structure of the most important persistent materials in the SyLab system.

`AnalysisOrders` are grouped by patient into a `Request` and processed together with a common `OrderNumber`. `Samples` representing the sample bottles received are allocated to a `Request`. In the course of processing each `AnalysisOrder` one or several objects of a concrete subclass of `Result` can be created. In addition, `Comments` can be added to a `Request` and to `AnalysisOrders`, `Samples`, and `Results`. These comments serve to document particularities of the object.

From the technical perspective, all of these objects are arranged in a tree, where the tree's root is a `Request`. The objects in the tree are linked over a joint superclass, `Node`. Figure 11.17 shows this structure.

At a certain point within the processing course, a typical request tree could look like the example in Figure 11.18.

DISCUSSION: DIFFERENT APPROACHES FOR CLASS MAPPING

Placing superclass attributes in super class relations only

To map the class hierarchy from Figure 11.18 to relations, we first selected the following intuitive solution from an object-oriented view. We created a relation for each class and placed the attributes of the superclass exclusively in the superclass relation, and those of the subclass exclusively in the subclasses relation. In addition to being able to store attributes without redundancy, this solution also complied with our assumption that there would be frequent polymorphic queries to the `Node`, `Result`, or `Comment` superclasses (e.g., "Find all results with order number x").

The persistence automaton implements potentially polymorphic queries in two steps: in the first step, it queries the superclass to determine the OID and type of requested objects; in the second step, it reconstructs the objects found one by one from

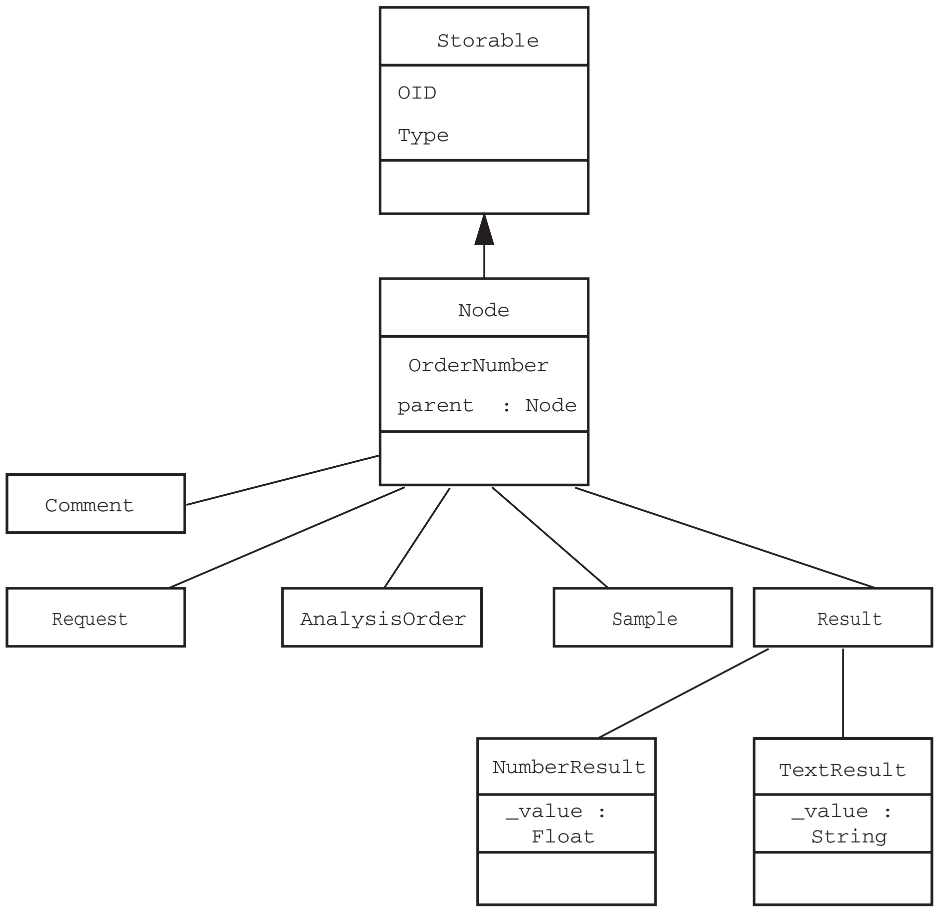


FIGURE 11.17

Request classes.

the database, using a relational joint operation and the OID as the key. The use of previously optimized stored procedures for join operations allowed us to achieve an acceptable performance for polymorphic queries in component tests.

The weaknesses of this approach emerged only in the course of the project when more and more applications were implemented and were evaluated in integration tests with respect to their (combined) database performance.

The actual number of polymorphic queries of the type described above was relatively small. Instead, there were many queries for types known in advance (e.g., “Find all samples currently in analysis automaton x”). This means that our two-step polymorphic mechanism was inappropriate for this type of query.

Another more serious weakness was the effect of the chosen structural mapping on the number of insert operations required. At least two relational insert operations were necessary for each object, depending on the hierarchical depth. So, creating an average request tree with seven `AnalysisOrders` and two `Samples`, in addition to the `Request`, translated into a total of twenty insert operations, which had to be grouped into a single transaction for data consistency reasons. However, our component tests

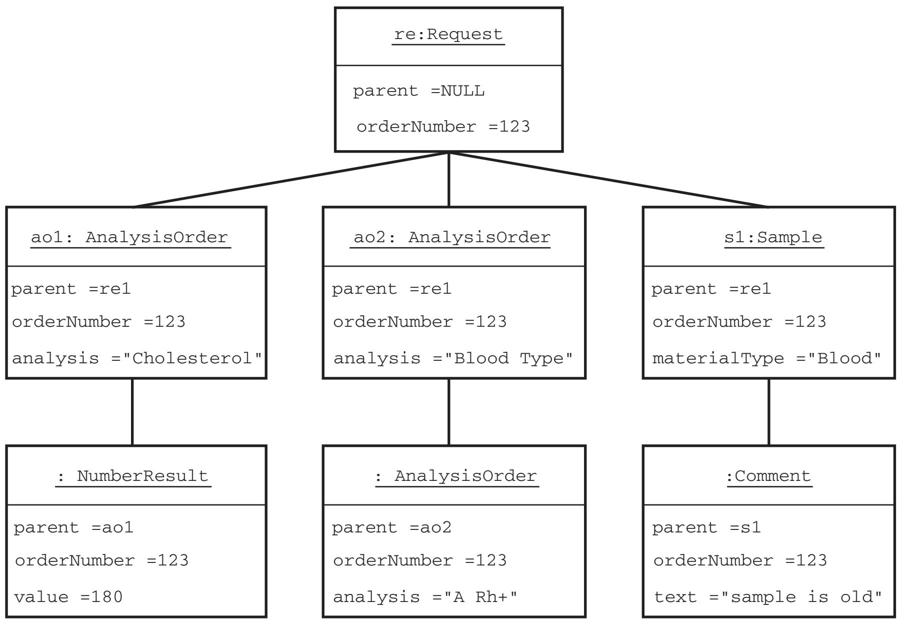


FIGURE 11.18
Request objects.

showed that the performance we achieved with this “request transaction” was still within the performance specified by the customer.

The real problems transpired later when we started integration and mass data tests: we observed aborted request transactions, deadlocks, and other database anomalies. Eventually, we realized that these problems were due to the fact that almost half of all insert operations were operated on the `Node` relation, growing dramatically so that it quickly turned into a bottleneck for the entire system. This problem became even more serious because moving specific common status attributes from `Node` subclasses to a common `Node` superclass, which was meaningful from an object-oriented view, caused an extreme update load on the underlying table.

In summary, we saw that the selected solution was not up to the requirements of a multiuser operation, and what’s more, it did not scale. We also learned that replicating superclass attributes in subclasses (see Section 11.3.3) was not the right way to go. Though it would have eliminated the (relatively harmless) need of relational joint operations for polymorphic queries, it would definitely have increased the bottleneck problem in the superclass tables, because each change to a superclass attribute required a number of update operations.

Storing objects in the relations of their creating class

The alternative we eventually implemented successfully replicated the attributes to subclass relations, while storing objects exclusively in the relation allocated to their creating classes. With this solution, the most important operations on persistent materials in the SyLab system, and first-time creation and manipulation of attributes can be easily mapped to insert and update operations. In addition, as we gained more practical experience with this solution, we found that a relational database system can easily handle polymorphic queries. With the structural mapping we selected for this project, these queries must visit all relations of the underlying class hierary. They always use the OID as the primary key and do not require expensive join operations. However, note that this holds true only provided that the part of the class hierarchy involved in frequent polymorphic queries does not grow much and the corresponding tables can be held in the database server cache.

12

The Development Process

As a professional product based on the division of labor, application software cannot be regarded independently of the process that led to the product's creation. This is a general insight of more than thirty years of software engineering. Looking at the T&M approach we can add: Software development is an intimate intertwining of an application-oriented software product and an evolutionary development strategy.

This chapter describes the fundamental characteristics and methods of evolutionary system development with prototyping (see Sections 12.1.5 and 12.3.2). We show what practical project planning and management can look like, and we explain how an evolutionary development process differs from conventional project models. Finally, we discuss how the approach we recommend can be interpreted within the Unified Process (UP).

We have been involved in numerous projects with very tight deadlines, limited resources, and complex development tasks. For some projects, we also elaborated and introduced complete, new process models. The experience gained in these projects provides the background for this chapter.

This chapter addresses not only project managers. It is essential for software developers to understand the mechanisms of an application-oriented software process, because this will guide the design and construction of an application system in a specific way.

12.1 BACKGROUND: EVOLUTIONARY AND TRADITIONAL PROCESS MODELS

We propose an evolutionary process model with prototyping and versioning that can be used to shape the development process. Instead of giving you a recipe for step-by-step implementation of a software project, however, we present guidelines that will help you help yourself. This means that a process model should not be thought of as a set of "how-to" rules, describing all the activities involved. Rather, our objective is to give you guidelines to work out a suitable approach for your software team and the project at hand.

An evolutionary development process cannot and should not distinguish itself from traditional methods by being interpreted to mean that a project can be managed by muddling through without plan or regulations. This would conflict with out professional claim for quality and lead to risks. To better understand the difference between our specific approach and traditional ones, we have compiled a number of topics from real-world project planning and management and use them in the "Discussion" sections of this chapter.

12.1.1 The Context of Our Process Model

Professional software development is organized as projects that can differ considerably in their orientation. These projects have different characteristics that determine the specific approach best suited for their needs and the methodological or technical support required.

Discussion

This section describes the concepts and techniques we developed and validated for process models with the following software project characteristics in mind:

Software project characteristics

- *Project goal*: Software should be developed in reusable components, as a collection of building blocks, or as a dedicated solution.
- *Project*: The project itself can be a new application, an improved variant of an existing application, or a reengineered software.
- *Application orientation*: Software should be used in one or more related application domains, which normally have a high level of domain complexity. Software should be usable over a relatively long period of time (one to several years) by different users with different profiles and qualifications.
- *Organizational context*: The development team is part of the user organization or works in a contract relationship between the contractor and the customer. A small software team normally includes three to five people. The team can scale up to several parallel teams with up to sixteen people each. (The mechanisms of formal user participation, extensively tried in Scandinavia and Germany, will not be considered here.)
- *Technical context*: Software is normally embedded in its environment, both from the technical and the social perspectives. It is a combination of hardware and software, and usually regarded as part of a landscape of heterogeneous software products.

12.1.2 The Process Model's Application Orientation

Andersen et al. proposed different dimensions for software projects (see Figure 12.1). We identify a strong similarity between the development and the management processes, which means that we see application orientation as the foundation of both processes.

Discussion

We can construct high-quality software products only if our development process is adequate. Using the author-critic cycle (see Section 5.3.4), we propose an approach that alternates between analysis, modeling, and evaluation steps within our development

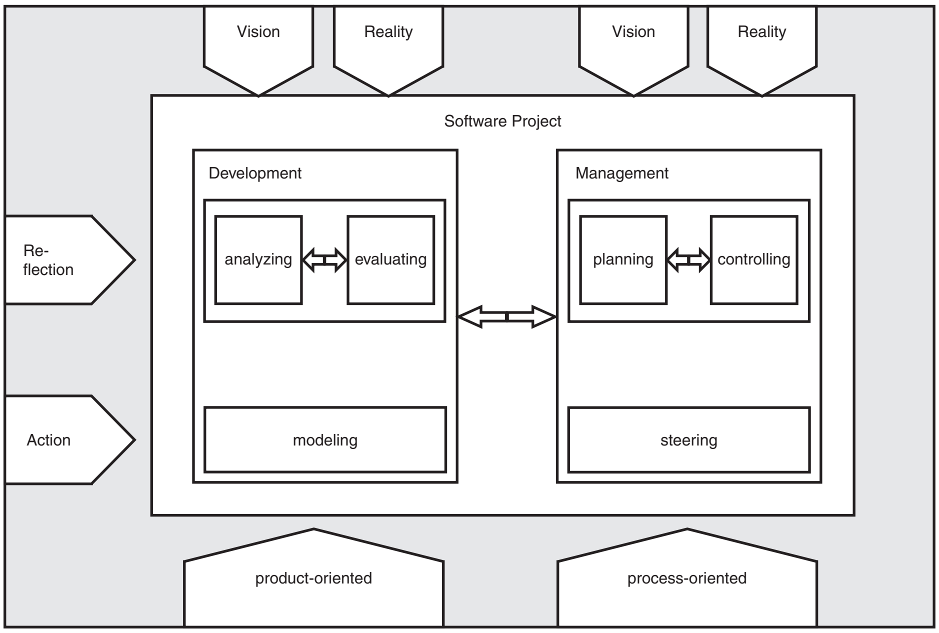


FIGURE 12.1

Dimensions of a software project.

activities. Sound understanding of the current situation is important for the development process and the target system. Section 12.1.5 takes a closer look at the relationship between actual state and target state modeling.

To better understand our discussion, let's look at a few important definitions:

The *development process* deals with a *software product*, involving analytical, modeling, and evaluation activities.

The *management process* deals with a *software project* aimed at making the development process easier to manage and plan, depending on the actual situation.

In addition to a suitable approach, application-oriented diagrams and document types are required to allow the participating groups to contribute their expertise and experience to the development and management processes.

12.1.3 The Classic Waterfall Model

Historically, the waterfall model was the first process model for software development and still shapes the thinking of many software developers and managers.

Discussion

The classic waterfall model (made popular by Barry Boehm) was the first linear stepwise model. It sees software production as a chronological sequence of self-contained activities. The numerous variations (see example in Figure 12.2) all require named and standardized development steps that are supposed to be executed one after

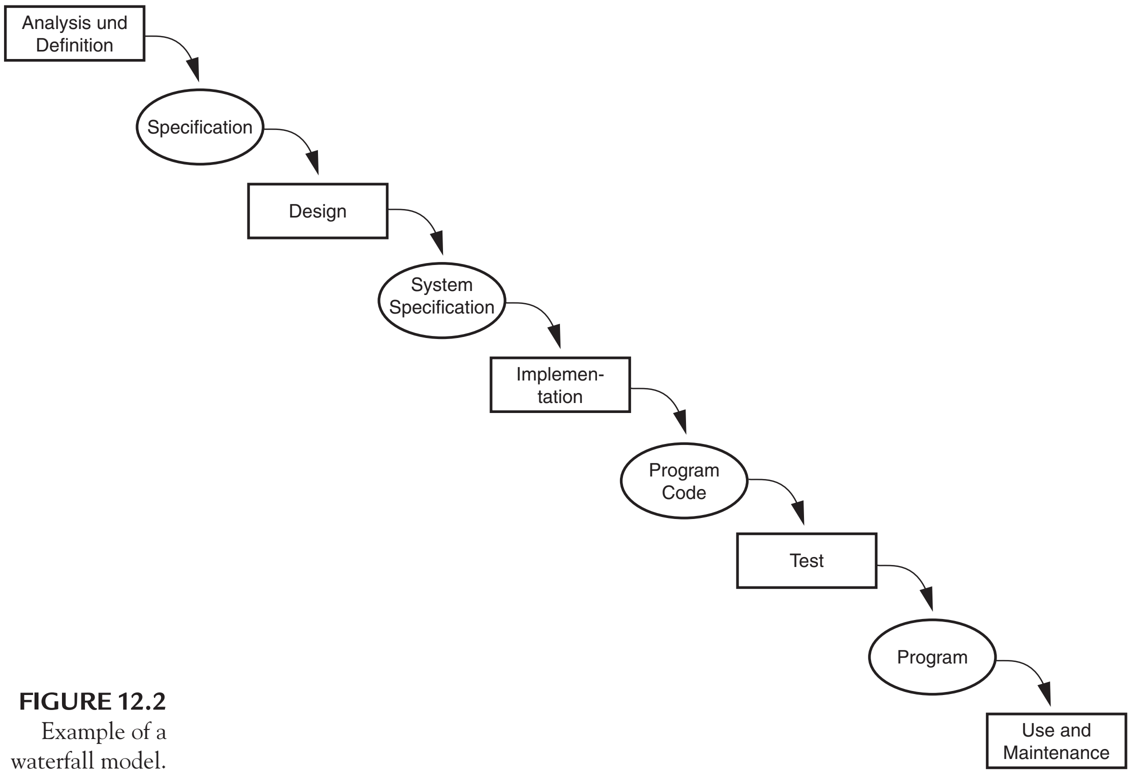


FIGURE 12.2 Example of a waterfall model.

another. This stepwise approach results in a number of documents (called "milestone documents") that should be specified in both their form and structure. During this approach, it is usually permitted and practiced to fall back to previous steps. However, taking a step or more back within this approach suggests errors and shortcomings that should be avoided in the course of an optimal project and should be minimized.

Weeknesses of the waterfall model

There has been a great deal of criticism of waterfall models (see for example Budde et al. 29, Parnas and Clement 85, Pomberger and Blascheck 96). The key problems with this model are that they poorly support application orientation and planning. We can summarize the following weaknesses:

- The activities of the process model are oriented to software technology rather than to the application domain's interest.
- The linear approach of this model is hard to maintain and plan.
- Essential requirements cannot be identified in advance and change constantly.
- Pure milestone documents are not reliable results, because their consequences are difficult to predict.
- The model does not consider developers' learning processes, and no prototypes are available.
- There is no systematic design feedback.

12.1.4 The Spiral Model

The spiral model (see Pamas 88) is regarded as an important improvement to the waterfall model. However, we think that it creates similar problems.

Discussion

The improvement represented by the more flexible spiral model over the waterfall model is also based on named and standardized development steps. However, these steps are repeated multiple times within a process, until the product is completed (see Figure 12.3). Though the spiral model includes a cyclical alternation between the activities involved and integrates prototyping to deal with the difficulties of identifying requirements, we think that its concept has the following flaws:

Flaws of the spiral model

- The model sees the object of a development process as a new and self-contained product, as does the waterfall model.
- The model forces you to run the activities involved one after another.
- The model separates software development from use and maintenance.
- The activities are oriented to technologies rather than the application.
- The model does not consider versioning or extensions.

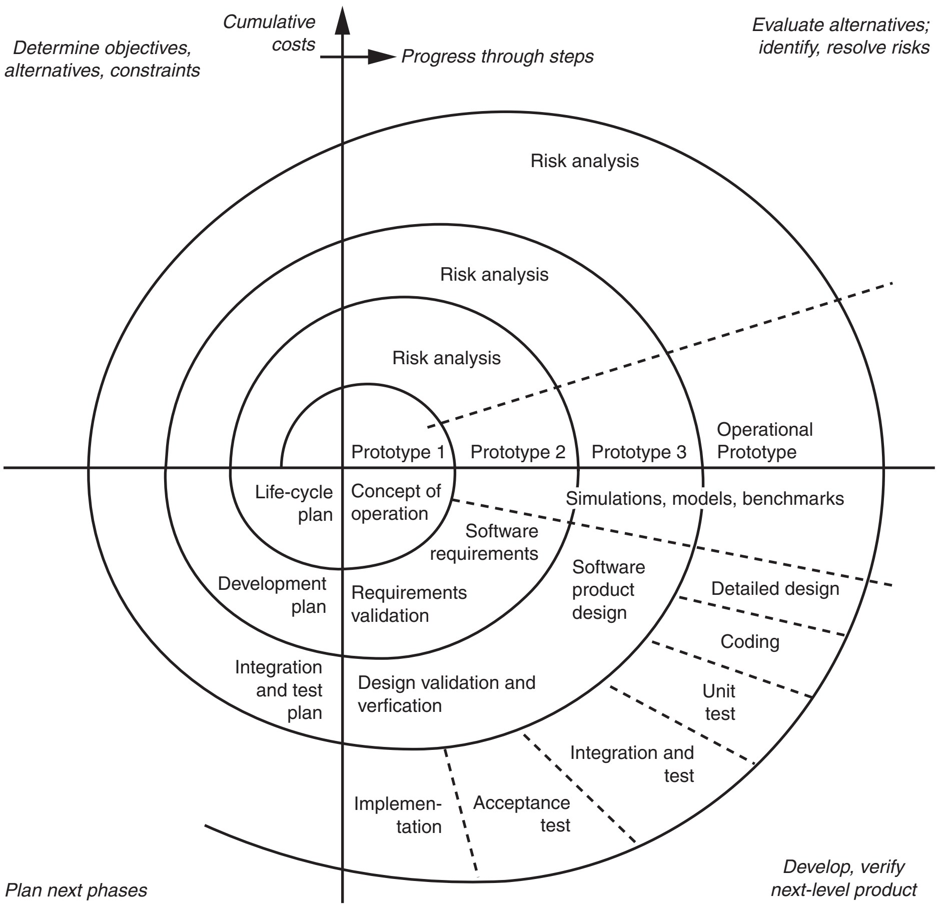


FIGURE 12.3
Example of a spiral model

12.1.5 An Idealized Evolutionary Process Model

This section formulates an idealized evolutionary process model, that is, an alternative to the waterfall or spiral model. It can be thought of as a fundamental reorientation of the software development process. Each project should reflect this ideal, depending on the individual case.

Discussion

Principles of the evolutionary process model

Our idealized evolutionary process model is shown in Figure 12.4. You will probably not be surprised that this model combines the basic principle of the author-critic cycle with application-oriented document types. Two major principles define this process model:

1. The general activities, namely analysis, modeling, and evaluation should be alternated as often and quickly as possible in each software project. This applies to both the development and the management of your project. Special attention should be paid to selecting suitable authors and critics.
2. All document types that we select for a project should be editable during the entire course of the project; there is no predefined processing sequence. Each document type should be selected in view of its purpose and how easily it can be understood by both authors and critics.

The development and the management process

We can identify the following requirements from the basic similarity between the development and management processes, based on our interpretation of general application orientation:

- *Document-based modeling*: The development process is based on documents representing a model of the application system (i.e., a "model-driven" approach in the sense of UP). This may not sound like doing something new. However, a

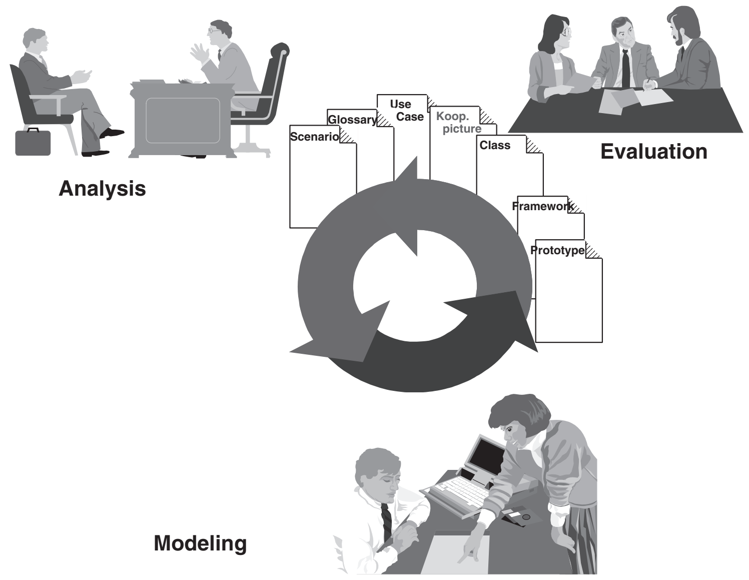


FIGURE 12.4
An idealized evolutionary process model.

closer look shows that those involved in the process create and evaluate only documents that represent relevant aspects of the application system. This means that each activity should have a purpose, and each document should contribute to the application system. All participants involved in the project should have a clear understanding of which document types are suitable for which target group and problem, and cooperate on this basis. We will come back to this issue in Chapter 13.

As one consequence, we do not use additional documents for the management process. The fewer dedicated management documents developers have to create, the stronger their commitment to the development process. Therefore, development documents should be systematically used in the management process. Naturally, this does not mean that you won't have to deal with planning documents in the evolutionary approach. We will discuss this issue in detail in Section 12.6.

- *Actual and target states adjustment*: Software development is not an end in itself; rather it is a service that should demonstrate its contribution to the corporation objective. We know that use contexts and domain-specific requirements may change considerably in the course of a project. For this reason, we often have to adjust our assessment of the current situation and the goals for the system under development. While traditional process models emphasize the target concepts, we stress the importance of documents representing the current situation (similar to UP). Target modeling requires the selection of appropriate prototypes (see Section 13.6 for more details).

  At the same time, application-oriented documents and prototypes form the basis of our management process. The more effectively they describe a current situation or vision of the system, the more likely that our management process will be able to check and review the project goals and allocation of resources for our project.
- *Constant feedback*: The document authors should receive an evaluation of how well their work can be understood and what use quality is expected. This is an important aspect of constructive quality assurance. It means that we assure the quality of a result during the entire construction processes. Quality assurance traditionally tends to be a control process, in which a software product is either accepted or rejected after a special test procedure, that is, when its development is complete. In addition to constructive quality assurance, constant feedback promotes the communication and learning process between the participants. Developers should be in a position to really understand the domain-related concepts and requirements. Section 12.3.1 describes how such feedback processes may look.

  For the management process, continuous feedback also means that we constantly check and revise our project plan, using mechanisms similar to those used in the development process. This means that we have to document the management process. We have to record project goals, distribution of tasks, responsibilities, and deadlines, in documents accessible to all involved. Moreover, all activities should be checked regularly by those in charge. As a result, we may have to introduce new goals and steps. This means that the management process is subject to author-critic cycles similar to those involved in the development process.

12.2 TOPICS FOR A DEVELOPMENT STRATEGY

Software projects normally raise similar questions: Which development activities are meaningful and when? Which types of programs can be developed in separate work steps? Can the project be organized on a decentralized basis? Who should we involved in the project team, and how should we organize the team? Which organizational form would be most suitable?

We have compiled the list of topics in the following section based on our experience with many software projects.

12.2.1 Sequence of Development Activities

Many critics of evolutionary process models argue that the sequence of development activities must always follow an inner logic, such as that design comes before construction. On the contrary, we argue that specific projects can and should specify a sequence of development steps, but not in a general process model.

Discussion

It may seem logical that requirements should be analyzed first and a design should be created before we build a product. Without discussion, software projects involve basic work steps (e.g., editing, compiling, versioning, documenting), handled in a particular sequence. However, the "who," "what," and "when" in a project do not follow a fixed (software-motivated) life cycle scheme, but rather should be oriented to the circumstances of the application domain. Many critics argue that there is a certain technical "subject logic" that by nature follows a waterfall approach. This is true for the microlevel, where it actually makes sense to think first what a piece of a program should do and how it should be structured, in terms of architecture, so that it can eventually be implemented and tested. However, these constraints apply less frequently on a broader project level. The Unified Process, with its parallel arrangement of activities, takes this into account. The following criteria are important when we try to identify activities and their sequence:

Identifying the sequence of activities

- The activities are not allocated to separate work steps or project phases.
- Each activity refers to a specific goal, that is, it is application-oriented, and takes the underlying technology into account (see Section 12.5.1).
- Modified domain requirements should continuously be taken into account to determine activities.
- Documents are not allocated to specific activities, which means that they will be updated whenever necessary, rather than "frozen."
- The learning process of all participants is supported by integrated project teams, but primarily by feedback from application-oriented documents and prototypes.

Example

In one of our projects, the developers decided not to take the conventional approach, that is, to begin with an analysis of the application domain. Instead, they constructed a technical prototype on a totally vague domain basis, showing little more than basic interactions at the user interface. Later on, they began familiarizing themselves with the situation and the requirements of their application domain by conducting interviews and attending lectures and seminars on the subject matter.

This approach was taken for the following reasons: First, it was the first time this team used Smalltalk as their development platform. Second, they felt that potential users in the application department didn't think much of the capabilities of their development department, because they had been disappointed with previous software projects.

For these reasons, our development team wanted to make sure that they mastered their technical tools before they addressed activities in that application domain, which was new to them. As the project progressed, this decision proved to have been right for that situation.

12.2.2 Objectifying the Development Process

We have said that software development is a document-driven modeling process. This statement raises the question of whether these documents may be the best prerequisite for an objectified development process. The aim is to substitute team members freely or to allocate analysis, design, and programming to different people. Our answer is that an objectified form of development is not feasible for most parts of an application system.

Discussion

Division of labour and depersonalization in software development

Objectification of the development process is often propagated as the software engineering idea, including the idea of creating software so that it can be produced by different people; in other words specialists with different qualifications and experiences (and different rates) should be used for the different sets of activities involved in a process model. For example, analysts would create the domain model, designers would produce the software design, software ergonomists would design the user interface, and programmers would write the actual program code. In addition, maintenance programmers and customer service staff would be available for post-development work. Another popular goal is to gain personnel independence within the groups involved; people who leave should be replaced without negative impact, and scheduling bottlenecks should be resolved through additional staff.

Project documents (e.g., milestone documents) should contain all information about the step currently worked on and the next step to ensure that this division of labor and objectification will work. For this purpose, the documents should be unambiguous, consistent, objective, and formalized to the widest possible extent.

We think that these are illusionary traps in traditional software engineering, connected to two problematic assumptions:

1. *Software is a solution for a given problem similar to solutions for problems such as mathematical equations*. This is true for a small number of cases only, where problems can be described precisely, as in mathematics.
2. *We can write unambiguous specification to have software built by different people and in a depersonalized process similar to a production line*. This assumption doesn't hold for most development projects.

We argue against this view with the philosophy that software development is first and foremost a communication and learning process between the people involved.

This is the only way that allows developers, who are not experts in a particular domain, to gain the knowledge required to fully understand the activities in the application domain and how they should be handled. Practically speaking we cannot let others do the learning for us. Developers will be able to develop high-quality application systems only if they understand the tasks and concepts of that application domain.

The system under development is not a "unique" solution to a well-defined problem. The groups involved have to negotiate the tasks to be dealt with in the future and ensure that these tasks are supported in the development process. Normally, we can specify only some aspects of this process in sufficient detail to produce documents that serve as a secure work basis for the team. The experience, views, and values inherent in each development project cannot be represented in documents. We summarize the resulting dilemma as follows.

The dilemma of software development

A software system should be formally documented, and these documents should be consistent, correct, and complete. On the other hand, both the goal and purpose of such a system cannot be described fully and objectively.

What impact does this dilemma have on cooperative software development? It would be naive not to allow a division of labor in a software project, expecting that the same people will work on it from the beginning to the end. In fact, this naive assumption contradicts our way of thinking. After all, we want to present the T&M approach as the appropriate basis for cooperative software development. This includes the fact that different people and groups are involved in the design and construction of system components.

The following section explains why the classification into S, P, and E programs proposed by Meir M. Lehman is helpful. Related to objectified development, this classification means that different program types can be described and implemented on the basis of a division of labor, with different degrees of success.

12.2.3 Lehman's Software Classification

The software classification proposed by Lehman is useful in describing the degree to which software can be developed based on written specifications. Lehman distinguishes between specification-type programs (S programs), problem-solving programs (P programs), and embedded programs (E programs).

Background

Lehman proposes the following classification for software:

- The characteristic of *S programs* (cf. Figure 12.5) is that there is a complete, formal specification describing well-defined problems and their basic solution. Examples of such problems include sine calculations and the Eight Queens problem. This means that we can normally define what an S program should achieve, regardless of a particular situation or use. The prerequisite for such problems and solutions is that they have to be generalized or abstracted before you can represent them formally in an objectified mathematical form. Using a mathematical method, we can then check whether or not the S program can be fully derived from its formal specification. This kind of verification can be achieved for a small number of well-known problems. Date-checking routines and model calculators to compute results are application-oriented examples of using S programs in real-world projects.

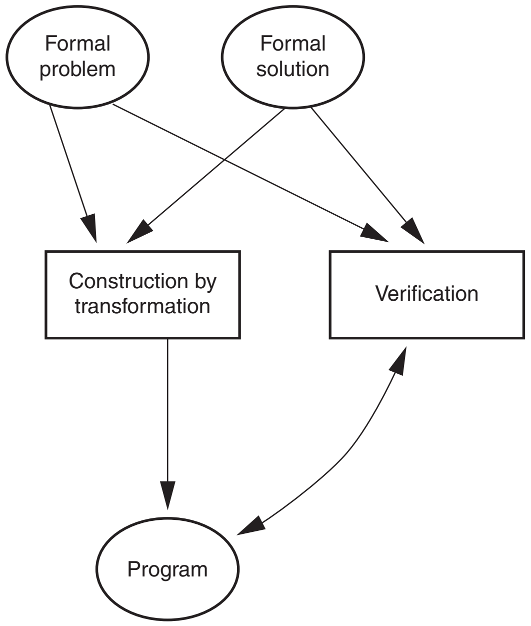


FIGURE 12.5

An S program in Lehman's classification.

- *P programs* (see Figure 12.6) are based on clear tasks, that is, they can be specified and solve a known problem. In some cases, P-type programs may even allow us to formally describe a task, which means that they relate to S programs. Chess and other games are good examples of P programs. They differ from the S program in how they solve a given problem, which must first satisfy the formal conditions of the task. For example, a chess program may not make a move that would be against the rules. In addition, we have to specify *how well*, or at what speed, precision or capacity, a P program should solve its task. This decision is normally taken by the developers, but ultimately evaluated and accepted by the users. For example, a novice will have different requirements for the performance and response time of a chess program than a master player. In the context of our projects, we often find P programs in place, for example to check the completeness and consistency of forms. A domain expert can easily tell what a consistent form is, and we know that an inconsistent form cannot be used officially, such as for contracts. On the other hand, when things should be checked and with what effort and precision is normally hard to define before the application is used.
- *E programs* (see Figure 12.7) are developed to support an application domain, for example in specific work situations. This means that a subjective aspect is already included in the conceptual definition of an E program. Whether and to what extent a state is considered a "problem" and what type of support is required depends largely on the observer's view. Consequently, it will be hard to find something like a self-contained and objective specification of a task or problem. This is the reason why solutions or algorithms are not defined abstractly for E programs; instead, they can only be evaluated and accepted by the participants. An office automation system is a good example. There is no

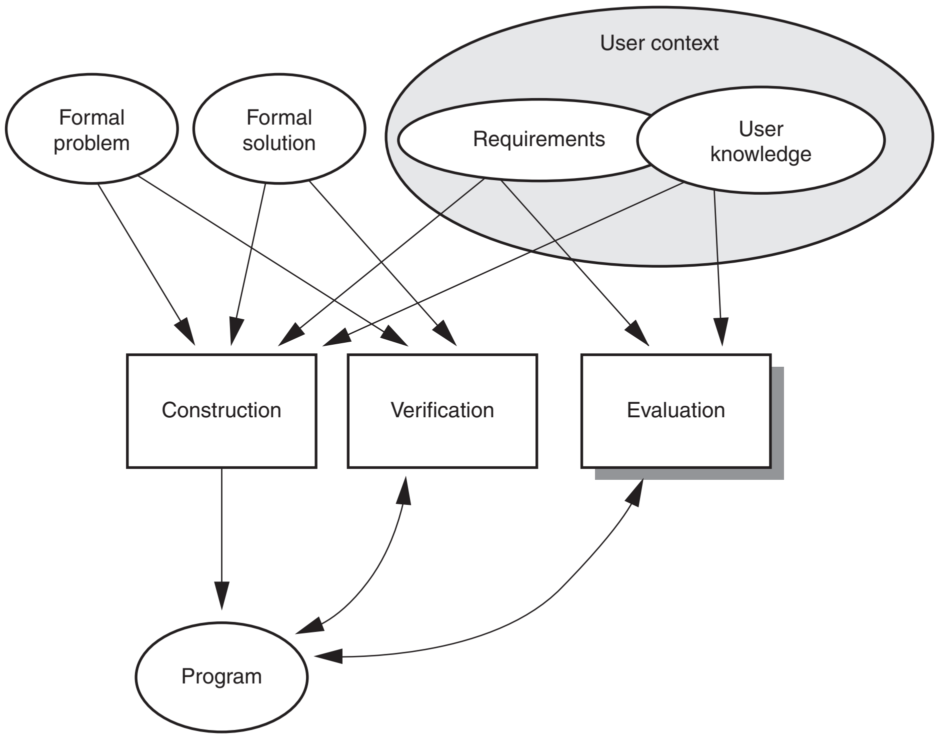
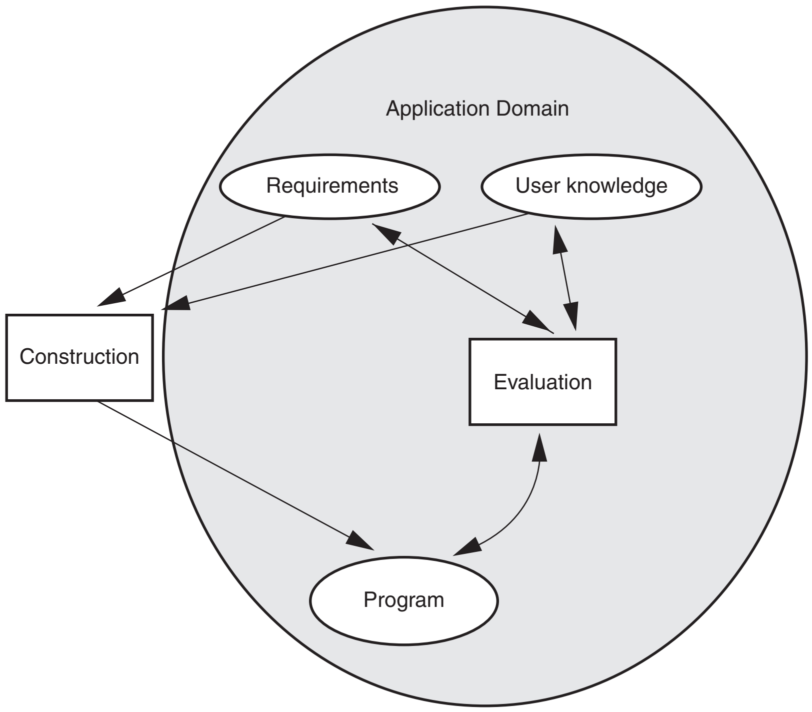


FIGURE 12.6
A P program in Lehman's classification.

FIGURE 12.7
An E program in Lehman's classification.



abstract way to tell when and in what situation such a system is a good solution for a problem; even identifying something as a problem depends on the participants and their tasks.

Correctness in a mathematical sense cannot be demonstrated for E programs. Whether or not an E program is right for a given work context and the people involved is a more important question. Most of the examples given in this book are E programs according to the classification here discussed.

DISCUSSION

Applying Lehman's classification to software projects

What is the meaning of Lehman's classification in relation to objectified software development? It primarily means that the program types can be described and implemented in different ways. In a traditional sense, S and P programs are well suited to specify an objectified basis. This is not the case with E-type programs, that is, we need to understand the parts of the context and work situation of an application system that cannot be formalized to be able to develop E programs.

In addition, the evaluation of S and P programs has different requirements. Although we can easily check S programs for correctness, this is difficult for some important properties of P programs. On the other hand, the evaluation of E programs is an ongoing process between all participants, since both the evaluation and the use may introduce changes to the requirements during our development process. In fact, new operative components and expansions change the team's and, even more so, the users' ideas about the application system. For this reason, it is important to achieve a high degree of continuity in the developer team to ensure evolutionary adaptation of the system as the requirements change.

12.2.4 The Cooperative Development Process

Application software is developed for different use contexts. The T&M approach normally supports different workplace types, as described in previous chapters. At the same time, it is important that existing components should be reused for workplaces in our system under development. This means that we have to deal with issues related to distributed development.

DISCUSSION

S, P, and E programs and the author-critic cycle

Lehman's classification of the development process in S, P, and E programs is helpful for finding an answer to the question of which elements we can create for an application in separate steps or by different teams within our development process. Suitable system parts are those we can specify as independent domain or technical services in the context of a workplace concept. When we have to develop a new service, then we need to clearly understand the application context. This is the only way to implement a service independently. One example from our own projects is the implementation of a workplace system, which was connected to a large host-based customer management system, in separate steps.

However, we have to understand that the implementation of system parts by independent work groups always represents the starting point for subsequent joint design decisions. We have to bear in mind that each new system element changes the context that integrates other elements. The E program concept shows clearly that it is not enough to identify a service and implement it to complete a development process.

We also need to do evaluations and obtain user acceptance to complete the cycle, since only the actual users can finally approve the system elements and services. This means that our understanding of E programs is based on the interplay between construction and evaluation. It also means that we need author-critic cycles throughout our development process to define and evaluate subtasks constantly.

The classification into S, P, and E programs also shows whether and to what extent external cooperation partners or subcontractors can adopt components. In fact, this classification shows that S program can generally be defined as separate tasks. For example, an application component for a banking application can be specified with reasonable complexity to calculate different types of credit ratios.

P-type programs

P-type programs are suitable for loose cooperations, but the cooperation partners should have sufficient expertise, and the components should be integrated and evaluated in regular intervals. A good example are the elements used for chart analysis in a bank's securities management system. This involves balancing the financial parameters against usability and manageability requirements.

E-type programs

In contrast, complex E-type application systems require a very high level of constant coordination and feedback in a project team and with users, so that they are hard to develop in distributed projects. Therefore, although system parts of E-type application systems can be developed as independent services by different people "in-house," it is rather unlikely that they will be suited for subcontracting to third parties.

An important management function in each development organization is to identify separate S and P elements within the entire application system under development. These elements can then be implemented jointly by cooperation partners and external contractors in separate steps. However, there are always risks inherent in this approach. Experience with formal specifications has shown that the effort required to create such a specification usually exceeds the implementation effort. For this reason, the corporate management should be careful to avoid having the preparation and assessment of external orders consume more internal resources than absolutely necessary, in comparison to developing the system part in-house.

If an organization has exhausted its possibilities for outsourcing the development of system elements, and there are still scheduling and manpower bottlenecks, then the organization could choose an insourcing option (see the following Section).

12.2.5 Organizational and Domain Integration

Project teams should be formed so that there is a high level of personnel continuity and different expert knowledge. Minimum staff fluctuation in a development process is a prerequisite for the learning processes involved, and it helps build a solid basis for systematic work based on documents. The domain integration establishes the application orientation and should reflect changes in that domain. The options available to use external developer capacities could also be evaluated against this background.

Discussion

We think that the ideal of objectifying development documents as a basis for project organization is not a good idea, because objectification should not become the primary principle of a project. In practice, dividing a project into separate self-contained steps for independent teams does not produce the desired results. Instead, we encourage the use of *integrated developer teams*, composed of domain and technical staff members, and strive for *continuity of personnel* within a project. This means that the same members of all groups should ideally work in the project team throughout the entire project.

This integration and continuity should extend beyond the individual project. Our project experience has shown that a development culture should unfold that influences the entire software development, and it should not disappear once a project is officially completed. For example, we see an *architectural group* (see Section 12.2.6) as a catalyst for a common development culture. It can ensure continuity beyond an individual project, based on its experience and knowledge of the context.

Of course, not all team members are normally able to handle each task in a project with the same skill. There is always the likelihood that some expert team members won't be available for the entire project. Nevertheless, we should try to solve the problem; purely technical skills are not sufficient to develop application software on the basis of written requirements.

Organizational Integration

Organizational integration is one solution to solve the continuity problem. The "relay principle" is applied to projects and "project families." When athletes run in groups during a relay race, they make sure that a pole will be handed from one runner to another. Similarly, each project allows sufficient time for "overlapping" staff when personnel changes occur. During this period, the departing team member and the new employee exchange information about the project directly. This is a primary prerequisite for a good understanding of the project and development documents.

Insourcing

A valid mechanism within organizational integration to work with external partners is *insourcing*, or local presence.

Insourcing means that external partners cooperate with the development team. The prerequisite is that the development style of these external partners is compatible with the method selected. The people concerned should not be substituted at will, nor should they be given other tasks. We have acquired some useful experience in various projects, where external advisors, who were not familiar with the application domain and the T&M approach, were used for programming support. The external partners worked in pairs (see "Pair Programming" in Section 12.3.4) with two or three team members, helping them to solve existing construction problems.

Pair programming is a good means of transfering the knowledge of external experts into a project.

Example

A Smalltalk development environment was supposed to be applied to a banking project. The team involved was not familiar with Smalltalk. Although the members of the team received training, it soon turned out that problems were occurring—particularly with the use of the GUI tools and visual programming components. To solve the problems, a six-month contract was signed with a consulting firm that had already carried out various Smalltalk projects with this environment. A consultant was available to the project three days each week. Each day the consultant worked with a team member at the person's workplace. The in-house person would explain the current construction problem, which the consultant then solved jointly with the team member. This resulted in a dramatic reduction in the development time for the application components and a considerable increase in construction knowledge in the team—without the need for writing complicated specification and requirement papers and the formal acceptance of externally created components.

Coordination Phase

If a local presence based on the insourcing principle is not possible, then preplanned and continuous coordination is essential between participants. It should be noted that communication media such as E-mail, telephones, and video conferencing are usually an unsatisfactory substitute for direct cooperation between people. A modified form of insourcing can be helpful, with an experienced team member working for a substantial period of time on the team of the external partner. Our experience with this approach is, however, not as good as with actual insourcing. Problems often occur when the two projects subsequently start to proceed differently. Different views about the need of feedback cycles and the significance of documents have a particularly detrimental effect. Subcontracts to partners in the form of so-called "fixed-price" projects in this connection have a universally negative effect. We found that there is always a disparity between the expectations and conceptions of the client and the readiness of the contractor to meet these requirements. As a result, both parties are dissatisfied with the project result, and an atmosphere of mutual distrust instead of a cooperation will prevail during the course of the project.

The down side of fixed-price projects

Domain Integration

The *domain integration* has to be added to the organizational integration of personnel. This integration relates to the developer organization structure and the work style in the application domain.

What has happened in many developer organizations is that the organizational structure of departments and domains, such as sales and central and distributed development, has sometimes created considerable friction in projects that work with application-oriented and evolutionary project strategies. Thought should therefore be given to avoiding this separation at the project level and within a project family. The specializations required of individual team members and the different knowledge each of them has to contribute must be used in a beneficial way in domain-integrated teams.

In contrast, we still find a department-oriented project organization, where certain activities or project types are only handled by the employees of a specific department, which does not only apply to development. Experience shows that this approach is also important for bug fixing and further development.

All-rounders versus specialists

Projects in which interactive workplace systems are being developed are those that particularly end up in a dilemma. On the one hand, in accordance with the principles of eXtreme Programming, all team members should be able to master all aspects of a project. On the other hand, due to the growing number of complex technologies, it is necessary that different types of knowledge about technologies are represented in the project team. XP designates the role of a consultant here. Even independently of XP, individual team members will specialize in areas like interactive software development, distributed systems, networks, databases, and mainframe applications. Although it is sensible to have specialists for the different technology and application subjects, it is important to disseminate knowledge about these subjects to all team members in a project so that they can use it meaningfully in application development.

Technical integration in a developer organization must be appropriate to the domain integration. There are obvious trends, at least in the financial services sector. The common factor is an application orientation that today is referred to as *customer orientation*. What is meant is the tendency in businesses that deal directly with customers to move away from the traditional separation of business sectors and instead be customer-oriented, thus providing more comprehensive individualized services. In other words, the processes and the organization of a company or enterprise should be oriented to the customers in order to guarantee maximum customer satisfaction. In addition to the generalization this requires, the signs are that specialization in the consulting and marketing areas will always exist for dealing with very demanding services and products.

Customer orientation

The "natural" allocation of projects and development activities along the traditional product lines and departments of the user organization must therefore be rethought. In Sections 12.2.6 and 12.7, we describe how this affects the design of applications software. In summary, we have to form integrated teams to guarantee consistency in the development documents and eventually the usage quality of the application system. These teams should consist of members with domain and technical expertise.

EXAMPLE

The Bank example

Customer orientation in banking is shown through the merging of savings and credit customer account services, tellers, and the securities business in service centers and the imminent integration of cross-selling products (e.g., insurance, property, mortgage services). There is a trend that new workplace types, such as professional customer advisory services, standard consulting services, teller services, and self-service facilities, emerge in the customer area. On the other hand, the traditional product-oriented or business line separation may well be retained in the back-office area in the near future. In the middle, we find controlling and monitoring activities, where customer-centered and product-oriented access probably needs to be combined. These different trends are leading to different use contexts in which customized workplace systems are being developed. Yet these systems must be developed with the same domain and technical basis. Here, the architectural concepts explained in Chapter 9 can help.

The best place to start a family of projects is with the customer-oriented workplaces, because this is where the new orientation is most obvious and is the easiest to evaluate its repercussions on other areas.

12.2.6 Developing an IT Organization

Evolutionary system development based on the T&M approach usually has consequences for the developer organization. If several object-oriented projects or whole project families are to be organized in one company, then it appears reasonable to use frameworks or component technologies. This requires further changes to the organizational structure, and consideration should then be given to establishing an architectural group and a team for product planning.

Zuli-12.qxd 20/8/04 7:46 AM Page 410

Discussion

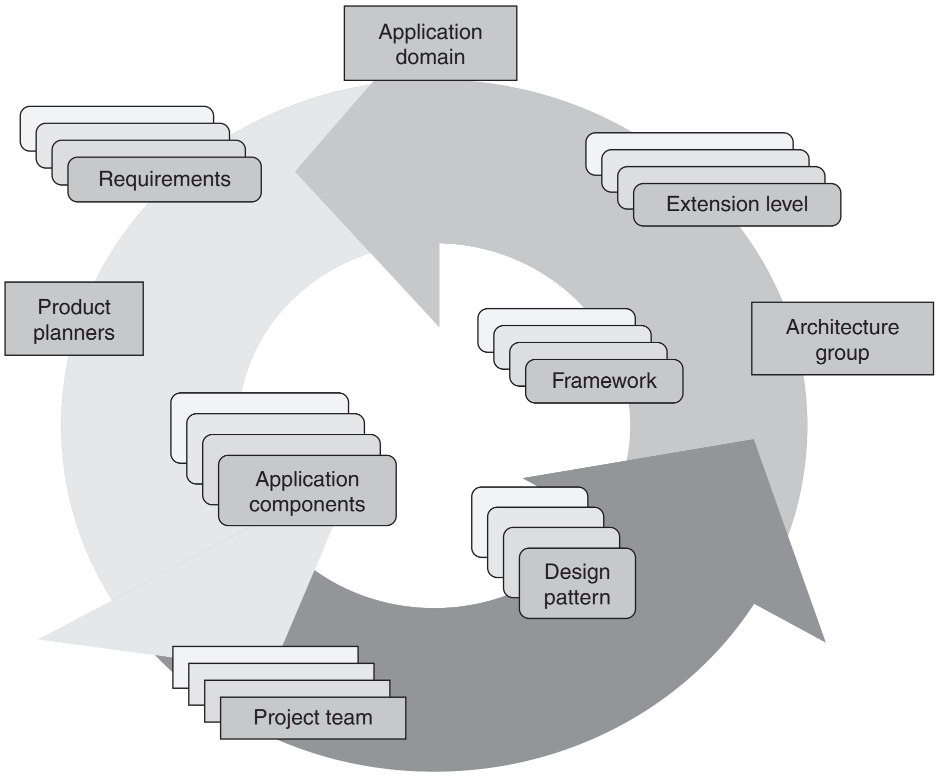
The establishment of application-oriented projects for the development of an interactive workplace requires changes to the developer organization. Our demand for technical and organizational integration can seldom be implemented smoothly into existing organizational structures. In practice, the mere question regarding which department is to provide the project management and how team members from various other departments will report to this management is enough to cause difficulties that can hamper the progress of a project.

Using frameworks and component technologies

Our concept of a core system with extension levels (see Section 12.7) transcends these organizational issues. Today, such a system with its different workplaces and components should actually be constructed through the use of (application) frameworks and component technologies. Section 12.7 describes the relevant domain and technical concepts. This section discusses the consequences for the organization and the management of the total process. It should be noted that only very few developer organizations have made the transition from conventional to application-oriented and evolutionary software development. The following observations are based on experience and provide some clarity about the trends described.

Figure 12.8 shows the interplay between design patterns, frameworks, and application components in the development of an application system with different extension levels. The framework architecture is the key to the technical and domain-related

FIGURE 12.8



Cycle of an application-oriented software development process.

integration of the application system. Different actors (or "workers," in UP terms) are necessary to put this into practice.

The role of the *application domain* as the central domain instance for application development in its different aspects is described extensively in this book. Here we want to emphasize that the application domain fundamentally defines the requirements of a system and establishes its usage value.

Project Team

The development of the application components that make up a system's extension levels is carried out by *project teams*. Together with the product planners, the application software developers form the core of every application project. Individual experts that come directly from the application department and one or two software architects are part of this core. These integrated teams design and implement the respective application systems or components. The principle of continuity in personnel is applied. This also means that a high percentage of the project members work on only one project and continue doing so. Having a person work on several projects at the same time has proven to be counterproductive.

Along with an understanding of the domain and tasks and requirements, a solid technical foundation is necessary for project work. It is mainly the application developers who need a grasp of these fundamentals. But it is also important that the other team members have an elementary technical knowledge.

The project work encourages further development of frameworks and components and the formulation of new patterns.

Product Planners

The design of an application system with its components and extension levels depends not only on actual requirements of the application domain. What is also important for the organization in which the developers work are the strategic decisions taken during the development of a product line. The responsibility for this lies with a department or with a group of people we call *product planners*. These product planners coordinate the development of new application components and the use of different extension levels, based on domain requirements and corporate policies.

Consequently, this group finds itself in a position of conflict, balancing user requirements on the one hand and the technical feasibility and the strategic concerns of the developer organization on the other. These team members need to be well-qualified in various fields. The ability to communicate is the main prerequisite for merging the different interests. Domain knowledge is obligatory. In addition, a solid understanding of the technical concepts of object-oriented application development is needed. Accordingly, conceptual patterns represent an important element in the language of product planners. It has also proven useful if product planners have some basic programming experience and are even able to construct presentation prototypes themselves. The existence of these capabilities noticeably helps to improve cooperative work with developers and architects. Lastly, it helps for this group to have a feel for company strategy and management capabilities.

Architecture Group

A framework-based architecture is the backbone of the type of application development described here. The conceptualization and development of this architecture is the responsibility of, what Ivar Jacobson has called, the *architecture group*. As described in Section 9.3.7, the architecture specifies the basic technological concepts, the principal tool construction, and, above all, the main domain concepts of a system. From its view of the entire system, the architecture group has to initiate the domain-oriented coordination between the projects through the main concepts of the business domain (see Section 9.2.3). It has to ensure that these concepts form a consistent foundation for the entire system.

Software architects combine technology and domain knowledge

Software architects represent the software-engineering viewpoint in application development. However, they also have sufficient domain knowledge to find the abstractions needed for the further development of frameworks and the "distillation" of domain components. To ensure that their knowledge and experience is reflected in the project work and can constantly be updated there, architects also always work as senior consultants in application projects. They are used there during design and implementation as well as for selected management tasks. However, they should not function as project managers. The domain-oriented architects are assisted by selected technology specialists, recruited for the implementation and further development of the technology basis across all platforms.

Tasks of the architecture group

The architecture group is responsible for the architecture management of the whole project. Architecture management can be divided into a domain part and a technical part. Both parts form the architecture's core. This system core is the prerequisite for constructing a family of application systems and enabling the reuse of concepts and components. The technical architecture management concerns itself with those requirements that concern the constructive basis for the domain-related projects.

The cross-project tasks of the architecture group include glossary management (see Section 13.4), **plus** the management of cross-project concept models and the entire glossary. This cross-section function is an important prerequisite for presenting users with a consist domain-specific taxonomy that extends across all integrated application functions. It requires project members to acquire a common work language that can be applied to several projects. Glossary management monitors the consistency of the terms used in the work language, building a foundation for a coordinated domain-related architecture.

Another important responsibility of the architecture group is to maintain close cooperation with product planning. As goals and priorities are established, the architecture group has to clarify which development options are possible considering the current state of technology and architecture as well as the current project activities.

Qualification profile of the architecture group

This discussion shows clearly that high demands are expected from the architecture group members. Only developers who have extensive experience should be part of that group. In addition to excellent software engineering knowledge with a solid theoretical or conceptual basis, they also need sufficient domain knowledge. Communication capabilities also appear high on the list of priorities. Of course, this describes an employee profile that only a very few are able to fit—or as someone recently quipped, "software gods." But we want to make clear that the requirements for application-oriented software development using frameworks and component technologies are ambitious. Each developer organization should therefore examine whether existing personnel resources can meet such a challenge. In summary, Figure 12.8 shows a cyclic process that integrates different participants and development results.

12.3 QUALITY ASSURANCE IN THE DEVELOPMENT PROCESS

As a professional product created by cooperating groups, the quality of application software cannot be regarded independently of the process that developed this product. We therefore present different measures that can guarantee the quality of the process. These measures "build quality" into the product.

The proposed measures influence different groups involved in the development process. Direct user integration, prototyping, and informal reviews are carried out by project team members in conjunction with the users. Formal reviews, pair-programming, and refactoring are techniques used within the actual developer teams.

12.3.1 Direct User Integration

In the T&M approach the author-critic cycle (see Section 5.3.4) is the key for integrating users. As a feedback technique, the author-critic cycle encourages communication and the learning process between the groups participating in the project. Thus the quality of model elements can be checked and improved, if necessary.

Discussion

The author-critic cycle

The author-critic cycle demands constant alternation between analysis, modeling, and evaluation. During these cycles, project members design models by writing documents, diagrams, and program code. The authors of the different models must receive feedback about how easy their work is to understand and about its level of domain-related quality. This is important for constructive quality assurance. Critics enable an author to improve the quality of his or her models and to add new model elements. The general principle guiding the author-critic cycle says that authors and critics should be different people. The best experts available should be selected for critics roles. In the sense of usage quality the critics often have to evaluate not only the domain-related correctness of a model but also whether the modeled system part provides appropriate support for day-to-day work. This kind of assessment can usually be best made by the users themselves.

Application-oriented document types

For cooperative work with users, developers select those document types that directly relate to the users' work, such as scenarios, glossary entries, cooperation pictures, purpose tables, and, to some extent, visions (see Chapter 13).

Other representation means, such as most UML diagram types of the software model and the program code, cannot be assessed by the users. These model elements are authored by a project member and then turned over to technical experts, such as database administrators, or other project members who act as critics. Again, critics are used to improve the quality and open the door to overlooked issues.

The author-critic cycle should consist of short time periods to allow authors and critics to be in constant contact. These short cycles should ideally be repeated until all participants have reached a common understanding of the current problem and its potential solution. This enables model elements to be developed in an evolutionary way and to achieve improved quality at the same time. Of course, we have to plan these cycles within a realistic time and resource frame (see Section 12.6).

12.3.2 Prototyping

Prototyping is the key feedback process for system evaluation, involving both users and developers.

Different tasks within our software development process may be supported by prototyping, including project initiation, application domain analysis, and the design and construction of the application system. One or several kinds of prototypes (see Section 13.6) will be well suited for such tasks.

Discussion

Prototyping has been a well-known and proven technique in software development for a long time. It is something that cannot be dispensed within an application-oriented approach, because there are few other means available for users to evaluate a system under development. Yet simply programming an executable piece of software is not sufficient.

Prototypes address problems

Prototyping should always be related to a specific problem—the one that the prototype is supposed to deal with. The problem must be defined clearly before the prototype is constructed. This prevents a situation where aspects for which the prototype was not designed are evaluated after we built the prototype. If the problem is not carefully defined, there is the danger of "muddling through." In other words, we build executable software versions and, if people like them, they are accepted as a success, but if they are not well received, they are discarded with the attitude "it was just a prototype." Different kinds of prototypes address certain problems (see Section 13.6). For example, no meaningful performance tests can be carried out on pure presentation prototypes.

Different kinds of prototypes are normally built during the entire development process. In large projects, we use the whole spectrum of prototypes, depending on the problem. However, this means that prototyping is not simply reduced to just another phase in the development process.

Prototyping, as we see it, is strongly focused on domain-specific problems and the usage quality of the software system. But this should not rule out the importance of the software engineering aspects and the demands on the architecture. Therefore, a methodological approach should be applied to allow for refactoring in the prototyping process. This means that existing functional prototypes have to be revised from a software engineering view to create a solid basis for evolutionary prototype development towards the target system (see Section 12.3.5).

In contrast, presentation prototypes are normally handled as "disposable" prototypes and retain that character. In real-world projects, we often find that an attractive user interface tempts the corporate management to turn a presentation prototype into the future system platform. Developers should always make clear that presentation prototypes are only design drafts. They demonstrate what the domain analysis and design have produced, but they are not developed for a specific target architecture or for technical quality. Presentation prototypes support the essential learning and cooperation processes in system development and the development of domain-related system visions.

12.3.3 Reviews

Informal reviews are events that are not bound to formal guidelines. We include road shows and user work groups in this category, that is, meetings that provide an opportunity for project ideas and results to be discussed in larger groups. In contrast, formal reviews are carried out in fixed settings and based on certain rules. The following subsections discuss these forms of the author-critic cycle.

ROAD SHOWS

One way to make the project ideas and results accessible to a broad interest group is through road shows. These events are informal to the extent that no uniform rules dictate how to prepare or conduct them. Most road shows are organized in the application domain, so that other people who are not direct users can learn more about the project work. Often, it is also useful to hold road shows within the IT department, for example to invite other developers interested in the project topics. New object-oriented projects in particular tend to create a mixture of interest and suspicion among other developers. Road shows normally represent a good opportunity to motivate these people.

The project team usually decides whether documentation material should be distributed to the participants of a road show before the event takes place. During the event, partial results are presented and discussed. The feedback that project members receive from such discussions can be beneficial for their project work. For the project setting, it can improve the flow of information and consequently often results in higher project acceptance.

USER WORK GROUPS

User work groups are normally composed of a selected number of potential users. These users assume the role of the critics, while the project members are the authors. The project team invites users to these work groups with the objective of obtaining comprehensive feedback about a project result in an early phase. It is important that this feedback comes from a group of users and not from an individual person. Anything that is unclear due to different situations in user organizations can be discussed and clarified in a larger group. Therefore, the important thing is not to have an individual document, such as a scenario, validated by a user, but instead to identify and possibly consolidate different views on a topic.

Creating cooperation pictures and evaluating prototypes

A user work group should be arranged as soon as an application domain model, including scenarios, and a concept model are ready. These meetings should then be used to create and discuss cooperation pictures to generate a common view of the tasks and processes involved in the particular domain. Similarly, each functional prototype should be presented to these groups so that hints and critical comments can be obtained for further development.

FORMAL REVIEWS

The concept of formal reviews has established itself as a feedback technique in many developer organizations. As the name already implies, this is a review type involving a fixed sequence of activities, in contrast to informal reviews. Formal reviews allow project members to obtain in-house feedback from other colleagues not involved in the project. It is important to guarantee independence between the authors and their critics (reviewers). Consequently, only those reviewers to whom the producers do not have a dependency relationship should be allowed to participate in formal reviews.

Rules for formal reviews

The detailed procedure of formal reviews varies according to the organization and its culture. The following rules have proven useful:

- The documents used for a review should be distributed to participants at least one week before the actual review date.
- During the review, all reviewers should first present their positive and then their critical comments, and all comments should be documented.
- A review is conducted by a review manager who has the responsibility to ensure that there is no discussion of content and that the only questions raised are those concerning the comprehension of critical comments.
- Unlike road shows, there are normally no system presentations in formal reviews. It is assumed that all reviewers are familiar with the documents distributed before the event.
- When the minutes of the review are available, the project management meets with the taker of the minutes and the review manager to elaborate a catalog of measures derived from the review. Notes are also taken at this meeting. These notes are then used in planning future stages and iterations.

12.3.4 Pair Programming

eXtreme programming techniques

Pair programming is an aspect of eXtreme programming that can considerably improve the quality of software development. With this technique two programmers work together at one computer in order to complete a programming task.

With pair programming each pair has two roles. One partner (the driver), namely the one who operates the keyboard and mouse, thinks concretely about how, for example, a certain operation should be implemented. The other partner (the reviewer) instead focuses on the design and implementation strategy. The reviewer controls the syntactic and stylistic aspects and decides whether the chosen approach is promising or whether the problem can be solved in another way. Each pair is formed according to the problem at hand and the availability of staff.

The roles (driver/reviewer) in pair programming should be changed as frequently as possible, up to several times per hour. This allows the work carried out during a day to be very concentrated because of the constant change in emphasis in what each partner is doing.

"Local arrangements"

The physical arrangement of desks and computers is very important for pair programming. It has proven helpful to have the computer used jointly by both programmers placed at the corner of a desk (see Figure 12.9).

In such an arrangement, the two programmers can alternate their roles quickly, because they both have direct access to the keyboard. Alternatively, both can also sit normally at a desk. However, the seating position should not influence the role distribution. The programmers should be able to change roles without changing their seats.

Discussion

Advantages of pair programming

Pair programming has several advantages:

- It can improve the quality of the source code, because two people work together. There is a greater chance that concepts and programming conventions will be maintained. Formal and semantic errors are usually discovered right away.

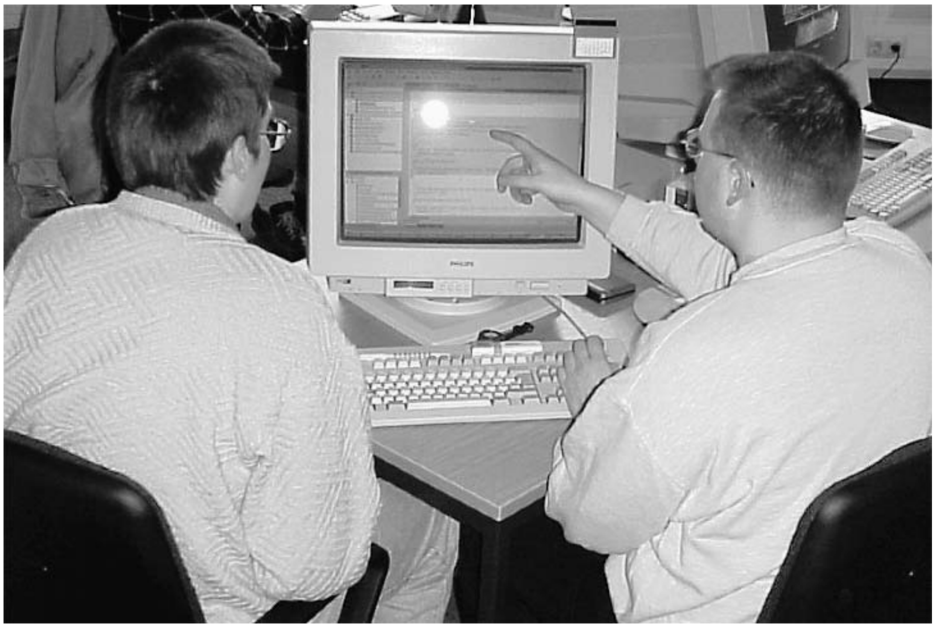


FIGURE 12.9

Example of a pair-programming session.

- When pairs change systematically, knowledge about the overall system is dispersed throughout the team. The departure or unavailability of a developer thus has no serious effect on project progress.
- The developers frequently question design decisions. Any blocked thinking or dead ends are avoided in development.

Pair programming for team training

In addition to these advantages, which mainly apply to homogeneous pairs, pair programming can also be used for team training. For example, an experienced programmer works with the new team member in pairs. Two things are important when using pair programming for team training:

- New team members should have good basic programming knowledge; this is particularly important for retraining in a new technology. Without minimum qualification and experience, the gap between experienced and novice team members is too great, with the result that the inexperienced person does not understand the work at hand and is usually too timid to ask questions.
- Experienced programmers should keep an eye on the training task assigned to them. It should be made clear that this task does not focus on development work.

Training in pairs is efficient, but it requires a high degree of patience and discipline from the experienced programmer. We have successfully used this approach in projects and found that the technical and domain knowledge of new team members was quickly brought up to the level of the other members.

Pair programming develops its full potential when used in conjunction with refactoring (see Section 12.3.5), design by contract (see Section 2.3), test classes (see Section 12.4.2), continuous integration, and collective ownership. Continuous integration simply means that sources that have been changed are integrated as quickly as possible. Integration should take place several times a day during the construction phase.

Collective ownership means that each developer may basically change all documents and source texts of a project at any time. The overall project knowledge required for this can be disseminated effectively in pair programming.

12.3.5 Refactoring

Refactoring is meant as an improvement of the internal structure of a software system. This should not change the observable semantics of the program to the outside.

Refactoring is seen as a disciplined approach that allows code to be cleared without building new errors into the software. Refactoring produces a subsequent enhancement of software of design.

When programmers are given the task of writing program extensions, it is up to them to check whether this extension would be easier to implement if the existing program were first restructured. Programmers should continue asking themselves whether it is possible to make a program easier even after a modification has been made. Refactoring takes place in very small steps. In principle, each individual refactoring (e.g., renaming a class, shifting an operation to a superclass) can be carried out in a few minutes. Large refactoring jobs should always be decomposed into small refactoring jobs to allow operative intermediate versions to be integrated periodically.

Discussion

Refactoring is time-consuming by nature. Time pressures in a project often lead to decisions to leave a system alone or to work around a problem. Nevertheless, refactoring should always be considered if a system potentially has a long life span, if it has to be reused and should remain capable of further development. The advantage of refactoring is that it prevents the feared deterioration of a system's structure. The question of whether refactoring would be sensible should be raised as soon as source code is duplicated.

Benefits of refactoring

Refactoring has the following benefits:

- Improved software design
- Comprehensible software
- Easier error location
- Shorter development times

At first glance the last point may cause some surprise. However, if you think about it, you realize that a good design helps to quicken the software development process. After all, the goal of developing something quickly is the reason for making a good design. Without a good design, development may proceed quickly for a time, but sooner or later a bad design will slow down the development. It takes a lot of time to find and correct errors. Changes take even longer because one first has to understand the code and then find the places where the code being changed was duplicated. New elements always require more lines of code because places that were provisionally patched have to be changed several times over again. A good design makes sure that software development will not slow down over time. Refactoring thus helps to ensure that software can continue to be developed quickly, because it prevents the architecture of a system from deteriorating. Through refactoring the design should in fact improve with time.

Like pair programming, refactoring has attracted a good deal of attention in the object-oriented community in recent years. Historically, refactoring has been used primarily in Smalltalk programming. In his book *Refactoring: Improving the Design of Existing Code*, Martin Fowler compiled a catalog of refactoring procedures, which supply simple step-by-step directions for improving suboptimal designs.

12.4 QUALITY ASSURANCE IN CONSTRUCTION

Besides the factors that assure the quality of a software product in the process, there are procedures and techniques that relate directly to the product itself. Design by contract (see Section 2.3) plays a key role here. It is used as early as in the domain design and is reflected in the concrete programming.

Along with design by contract, testing can become an important form of constructive quality assurance. This is where design by contract and aspects of eXtreme Programming can be merged.

Lastly, we take a look at the state modeling for the design of those classes that incorporate the concept of a business process. Design by contract is also important in this connection.

12.4.1 Characteristics of OO Testing

Compared to classic imperative programming, object-oriented (OO) programming has some structural and dynamic particularities that have to be taken into account in designing our tests. Testing object-oriented programs is an extensive topic, and we can only provide an overview here. We refer our readers to the seminal work of Binder for more detailed information.

Encapsulation

The smallest constructive units of an object-oriented program are classes. A class has operations that are different in character from the classic subprograms in modules. In class construction the specification (interface) is separated from the implementation. The specification is visible as the interface of operations to the class's client. However, an operation can be implemented by more than one procedure in different classes.

The procedures differ from classic subprograms in that their coupling (over jointly used objects or through mutual calls) is much stronger. Due to a number of dependent elements, the complexity inherent in classes is higher than with typical functional modules.

A class not only encapsulates a number of procedures, it also normally contains data that models the state of an object. The state of an object is determined by the values of its attributes. Most of these values are references to other compound objects, rather than primitive data. Moreover, polymorphism can be used to bind differently typed objects to identifiers, resulting in a large state space.

In summary, this means that the smallest testable unit that makes sense in object-oriented programming is an operation in the context of a class.

Information Hiding

We can use the principle of information hiding (see Section 2.1.6) to encapsulate the implementation of the operations of a class. This is a valuable principle in software engineering, but it makes our testing of classes more difficult. In our experience, pure black-box tests cover only one-third to half of the states or execution paths that a class can have, because the test can cover only the structure visible externally.

When developing tests, however, we often need to know the specific state space that an object can have, the embedding structure of a class, and the resulting dependencies. Consequently, we need direct access to the encapsulated state of an object. A class should offer a sufficient number of access functions to allow tests to identify an object's state.

Complexity and Dependencies

Object-oriented application systems are constructed from objects. Objects communicate with one another and influence each other's state through the exchange of messages. Whether and how an object will react to a message is defined by its own state or by the state that it can observe on another object. This means that objects are able to form a time-dependent network of communicating units with several "entry points" at runtime, but without a central entity that monitors the control flow. This clearly complicates the testing of control flows in such systems. For example, Binder proposes higher techniques, such as the use of stubs, mock types, and dummies, to deal with this problem. These techniques let you create a test context for the object under test, without the need to reconstruct the entire system environment.

At a semantically higher level, design patterns are helpful to identify dependencies and communication relationships between classes and test strategies.

Inheritance and Polymorphism

Inheritance and polymorphism (see Section 2.1.20) make testing more difficult. As a result, the specification and implementation can end up being distributed over several classes. In particular, the structure of the source text no longer reflects the control flow, which cancels out one of the key arguments in favor of structured programming. "Where are all the places an implementation is used?" and "Was an operation redefined somewhere?" are some of the typical questions in connection to object-oriented testing.

The principle of information hiding is largely eliminated in an inheritance hierarchy. Unless visibility is restricted, a subclass has unlimited access to all features inherited from its superclasses. Consequently, all features of a class and its superclasses have to be retested. This is the only way to ensure that no unexpected side-effects can occur in the context of a new class.

Semantic differences between inherited and redefined operations are not seldom and may surface only in specific contexts. Even if the pre-conditions and post-conditions remain textually equal, they can have different meanings in the superclass and a subclass.

Abstract classes

Abstract classes are a special case in that they won't let you instantiate test objects without implementing abstract operations. This means that we cannot test them dynamically, unless we do some additional implementation work. All operations implemented in subclasses must be checked against the specification of their redefined abstract operation.

Polymorphism requires special consideration in connection with inheritance. Many parameters and return values of an operation reference objects. In statically typed languages, they can reference objects of all polymorphic classes, and testing has to ensure the compliance of these classes with the specification.

This may lead to a situation where errors occur within a class, *A*, which uses an operation of class *B* that was redefined in a subclass C. In practice, design by contract can help eliminate these problems to some extent.

12.4.2 Testing OO Programs

The special structural and dynamic features of object-oriented systems have some consequences for testing. We will take a look at the class test, the integration test, and the regression test. We conclude by explaining the concept of test cases and test classes.

Testing Classes

We said before that the smallest construction unit is a class.

A *class test* checks the operations and interaction between operations of a class. It most closely corresponds to the unit test in imperative programming.

The basis for a *unit test* is a precise specification of the expected behavior, for example, by using the design-by-contract principle (see Section 2.3). The post-conditions are then the different kinds of external behavior that are checked by black-box tests. These tests are constructed in the form of separate test classes.

An *integration test* is used to locate errors in the "using" rather than in the "used" code. Typical errors that the integration tests look for on the operation call level include call of a wrong operation, incorrect use of a correct operation, and unexpected results.

The large number of relationships known (use, inheritance, association, polymorphism) increase the significance of integration tests for object-oriented programs, for example, to identify cyclic dependencies.

Inheritance and polymorphism

The use of inheritance and polymorphism means that inherited operations are also tested, along with redefined and new operations in subclasses. If an operation was not redefined, the test from the superclass can be used without modification. A redefined operation must first satisfy the tests of the superclass. If the reimplementation of an operation weakens the pre-condition or strengthens the post-condition, then additional tests are needed to check these new boundaries.

If a new class that redefines operations is introduced into a class hierarchy, the test has to find in client classes all locations where that new dynamic type can occur. These classes should then be tested using the new dynamic type.

In practice, there are problems in instantiating the objects of a subclass, so that they can be bound to the superclass tests (number, type, kind of test).

When building complex classes, it is normally a good idea to specify the behavior in a state diagram and use test cases to check for potential state transitions. This requires the use of *gray-box tests* that know the internal representation of the states.

Gray-box tests

The problem of information hiding in the gray-box test can be reduced by implementing classes with two interfaces. One is the public interface, which is subject to the black-box test. The object's state is defined through private attributes that can be probed or changed by protected operations, which can be overwritten in the subclass (see Section 2.7).

In Java, this inheritance interface can be declared as `protected`. It is also visible for classes in the same package. However, the test classes then have to reside in the same package that includes the classes under test. This is particularly suitable for testing auxiliary functions of complex algorithmic operations.

The construction of class teams and tool classes leads to complex control flows. Different design patterns that regulate the control flow are examples of patterns that are used (e.g., *mediators*, *events*, *chains of responsibility*, *adapters*, *template methods*). These are often based on abstract classes and their specialization. Testing the correctness of the control flow is problematic.

The following strategy is recommended for gray-box tests: The tested classes are implemented and modified in a derived subclass in such a way that we can track the control flow (additional collector parameter in the operation call). Stubs should be developed for the classes used. In the test class, the network of the tested class and its stubs is instantiated and linked together. When operations are called, the test class passes a collector parameter, and each successively activated operation or stub registers with this parameter. The test class can then check the resulting list.

Tim Mackinnon et al. propose the concept of mock types. Based on this concept, the state of a class should be manipulated and probed exclusively through protected getter and setter methods. This makes tests of the state model more reliable. In addition, it reduces the chance that modifications of the underlying implementation of the state model in subclasses jeopardize the correctness of inherited methods.

Classes should be organized in subsystems that are relatively autonomous in the way they provide their services and are loosely coupled with other subsystems. This reduces the number of control flows to be tested and the possible side-effects.

Integration Tests and Regression Tests

The classic waterfall model (see Section 12.1.3) commonly used in software development distinguishes clearly between integration and regression tests.

An *integration test* runs during the development phase based on unit testing and is used to find errors in the interaction between new classes or subsystems when they are integrated into an existing system.

***Regression tests* are traditionally used during maintenance. Whenever a change is made to the code, a regression test should ensure that this change has not caused any errors.**

The boundaries between these two kinds of tests blur in evolutionary object-oriented application development. The same also applies to class and integration tests. This is due to several reasons.

First, it is difficult to localize the effects of changes to the code (polymorphism, control flows); second, a system develops during several evolution cycles, which means that "maintenance" is necessary early on in the process. For refactoring to be consistent, tests must constantly ensure that the changes implemented do not have any effect on the system functionality.

Test Cases

For constructive quality assurance it is important that programs are constructed and tested in parallel. Test cases can be written on the basis of domain-specific analysis and design documents (see Chapter 13) and contracts.

Tasks and business processes are described in scenarios and system visions. This reveals the activities to be supported by the application system. Application tests can be conducted on the basis of these descriptions. The situation is similar with screenplays (see Section 13.6.1), which describe a realistic mix of activities at a workplace. Screenplays can be the basis for end users to run application tests. Business and use case diagrams that clarify the overall domain-related scope of the application under test are another means that can be used for these application tests.

Feature acceptance tests can be specified effectively on the basis of detailed system visions (e.g., tool visions) and contracts for the classes.

Test Classes

A method for systemizing testing known from eXtreme programming specifies a separate test class for each class. It also recommends that test classes be written before the classes that are being tested. It will then be clear which problem is being solved under which conditions before a class is actually implemented. It also ensures that statements are well covered.

It has proven to be sensible to allow testing and programming to follow in quick alternations, so that each new class and operation is tested immediately. At the end of the day, all test cases for an integration should run successfully. Testing should thus become a basic attitude in the development process.

Changes will not be safe unless we have test classes, particularly when refactoring (see Section 12.3.5) is involved. Since refactoring changes the inner structure of software, while the interface remains the same, your tests should run without error after each refactoring step.

Systematic use of design by contract does not make test classes superfluous. While assertions of the contract specify only a part of how an operation is used, such as “extreme cases,” the test class can include other important states and parameter values. This increases the level of test coverage.

In addition, a protocol is used implicitly as the basis for each contract but is not checked explicitly, because each contract relates only to one operation. Critical or illegal call sequences can only be checked in test classes. In summary, test classes offer the following benefits:

Benefits of test cases

- The source code is well suited for further development, because changes can easily be checked for correctness.
- The debugging times are reduced, because errors can be localized more quickly.
- Interfaces become simple, as each programmer prefers to test simpler interfaces.
- Test classes show the use intended for a class by the developer and can be interpreted as part of the documentation of the source code.

A suitable tool like JUnit should be used to automate testing.

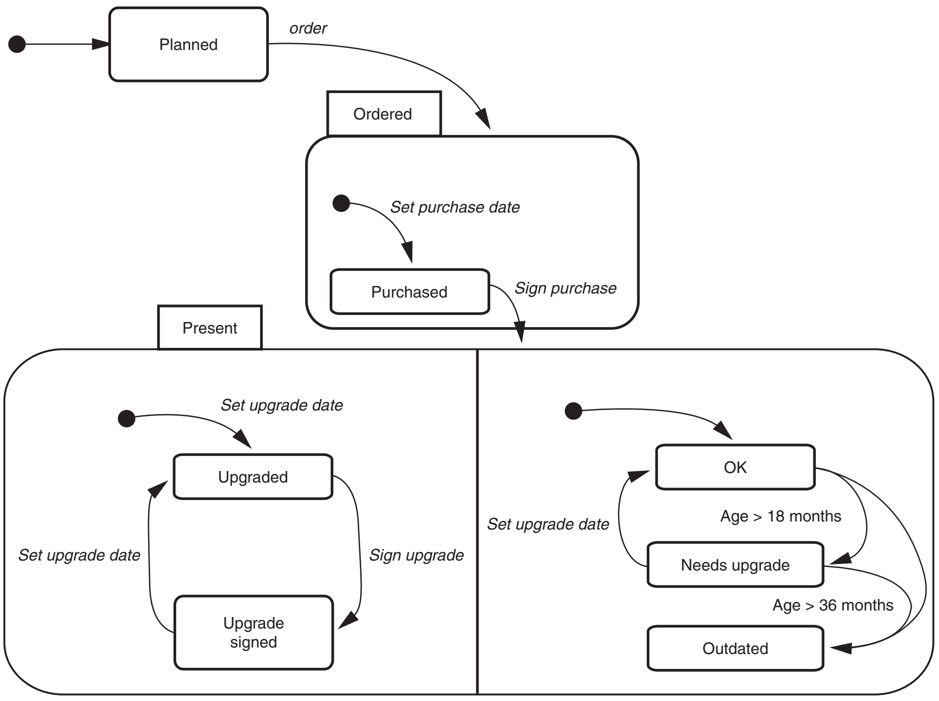
State Modeling

State modeling is originally either a task in the development of technical systems or a technique in database design for documenting changes of object-like items in database tables. From the object-oriented view, an object actually has a state (the respective bindings of its attributes) that can change while the object identity is preserved. We therefore do not initially need an explicit state model.

This changes when we have to model a business process or workflow that relates to a material, such as a set of documents. In the course of a workflow, the material often adopts domain-specific and defined states. From the outside these states can be observed at the interface. For safe handling of material it is important that certain operations are only allowed in appropriate states. Note that these state transitions are triggered by operations; in other words, they are not set as attributes directly by the developer. It therefore makes sense to elaborate a state model at least at the design level and to clarify the states and transitions through operations on the basis of a finite automaton. State charts have proven an effective representation means (see Figure 12.10 and Harel 87). An explicit implementation of a state model, such as using state patterns, is useful for critical objects.

When state transitions with objects manipulated in business processes are modeled, the domain knowledge is transferred to the objects. These objects become more than just data sets. Pre-conditions and post-conditions can also be derived from the different states. It is clear in each state which operations are permitted and which are not. Furthermore, for each operation it can be identified from which source states (relevant for the pre-condition) it is transferring the object and to which target states (relevant for the post-condition).

FIGURE 12.10 Connection between management activities.



12.5 PROJECT MANAGEMENT

The literature often sees project management as an independent task that is separate from actual software development. We have found that this is not a good approach for application-oriented software development based on the principles of object orientation. This section describes the relationship between application-oriented software development and project management.

12.5.1 Fundamental Activities of the Management Process

In the evolutionary model, the different activities of the management process are closely oriented towards the activities of the development process. The basic principle of the author-critic cycle applies here as well. Analysis, modeling, and evaluation become planning, steering, and control. In terms of content and scheduling, each project must be planned, organized, and controlled.

Project Planning

The objectives envisaged for a project and its various stages (see Section 12.8.2) must be worked out and documented in a comprehensible way. The domain vision for the system is the driving factor in a software project. Management must ensure that this domain vision is compatible with the strategic goals of the company as well as with the available resources and the technology being used. At the level of individual tasks, it has proven useful to allow team members to work out the fine planning themselves.

Project Steering

Process-controlled management style

Supporting management style

There are many ways to manage different project activities. To simplify matters, commonly used management styles can be divided into a process-*controlled* or *supporting* style. A process-controlled management style is characterized by orders from above and detailed job specifications. It corresponds to the traditional hierarchical military management style. In contrast, the supporting management style more closely matches our evolutionary approach. Here the project manager takes on the role of coach, spokes person, and service provider for his or her team along the lines of Tom DeMarco and Timothy Lister. The project manager creates the opportunities and conditions that provide the team with an optimal way to work.

Based on eXtreme Programming, we recently tried “pair managing” in large projects and project families. In pair managing, two project managers of equal rank work closely together in the steering of a project. Along with receiving better feedback on their own work, another pleasant side-effect of this approach is that the project managers still have the time and opportunity to get involved in the content of the project subjects.

Project Control

A careful assessment of the situation is an essential starting point for each cycle in the management process. The current situation in the application domain is as important as the situation in the project itself. As a result, for the control aspect it is of prime importance to evaluate the domain documents and compare them with the current situation. The planning documents are checked to ensure that decisions are followed and activities put into action correctly. Here the separation of roles between authors and critics plays a particularly important part. Real-world projects have shown that it is useful to have team members plan and run project meetings related to these topics.

The Interplay of Management Activities

Figure 12.11 shows the connection between the fundamental activities of the management process. For our purposes, it is important that none of these activities may be viewed in isolation or without influence on the other activities. To ensure that this interplay is maintained and reproducible in its consequences, we propose a quick alternation between planning, steering, and control in our evolutionary process model.

Conversely, it has proved detrimental to have these activities scheduled as far apart as suggested by the classic phase models. In these models, the "early" steps are directed towards planning, while the management carries out the steering function within the plan over a long period of time, and eventually executes the necessary controls at the end of the project. The large number of milestones and milestone documents have a formal control and monitoring function but give no real insight into the project.

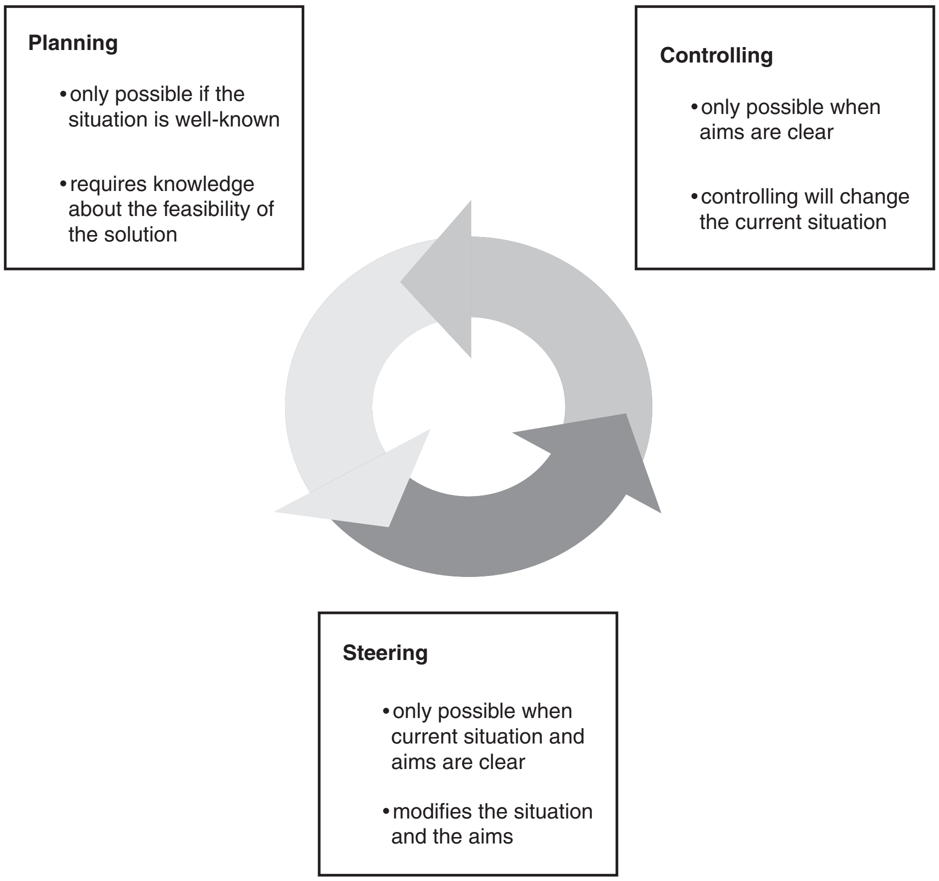

Planning
• only possible if the situation is well-known
• requires knowledge about the feasibility of the solution
Controlling
• only possible when aims are clear
• controlling will change the current situation
Steering
• only possible when current situation and aims are clear
• modifies the situation and the aims

FIGURE 12.11 Connection between management activities.

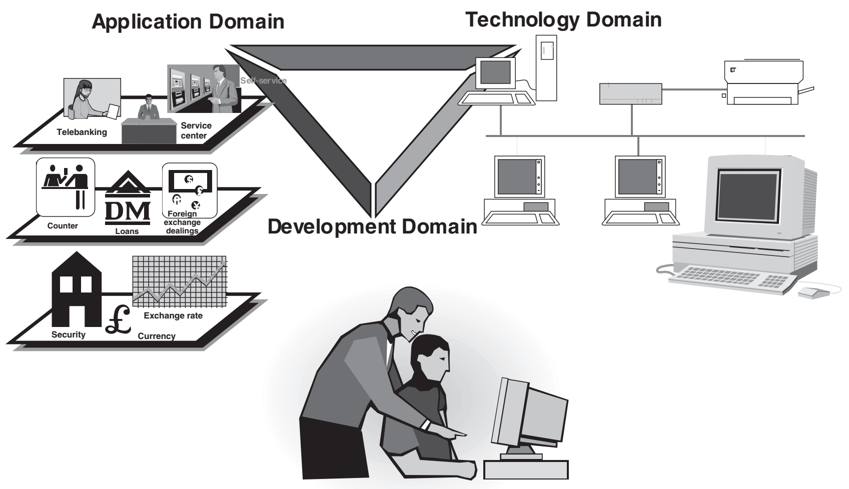


FIGURE 12.12

Contexts of the management process.

12.5.2 The Contexts of the Management Process

The management process refers to different contexts: application domain, development domain, and technology domain. All three contexts have an effect on the development process and therefore must be taken into account by the management process (see Figure 12.12). The following subsections discuss the three contexts for our management process.

The Application Domain

The application domain is where the future software system will be *used*. In concrete terms, this can be an organization, an organizational unit, or a workplace type. The application domain determines largely how a project will be oriented and executed.

The size of the application domain

From the application developer's perspective, it is desirable to keep the application domain as small as possible to reduce the complexity of the project. For example, we would not recommend building software support for an entire hospital as the application domain. The tasks and processes are so diverse and complex that they would extend beyond the scope of any project. On the other hand, the application domain has to be large enough so that its context is still comprehensible. For example, it wouldn't make sense to reduce the application domain in a hospital project to the work of a single ward nurse, because her work can only be understood and modeled in cooperation with doctors, administrative personnel, and other nurses.

Who is the user?

When the application orientation is chosen, we have to answer the question of who is considered a user. In addition to the "direct" users, the people who will use the application system, there are probably other people affected by the results of the application system and modified work processes, so that they too should be integrated in the process. A good example is the issue of "customers of customers" in banks. Naturally, we will want to integrate the bank's employees and its customers, as both groups are normally the direct beneficiaries of our application system.

The Technology Domain

The technological environment of a project determines which architectural concepts, design guidelines, technology models, and concrete technical components for the system base can be selected and used in the development process (see Section 9.3.7). The assessment of the technology domain is a separate and ongoing task, often underestimated by the management. The issue here is which new technologies and means are necessary for implementing the system and whether new technologies should be employed for strategic and technical reasons. A rule proven in our projects says that only one fundamentally new technology should be introduced per project. We realize that it is not always easy to stick to this rule.

Selecting technologies

The reason is that technologies are no longer selected by the corporate management alone. For instance, as a result of the recent popularity of Internet and Java technologies, many application projects were experiencing major "external" pressure to use these technologies with relative disregard of project objectives. We can therefore assume that the technology domain will increasingly develop its own dynamic that influences project direction. It is important that at least the type of technology is made known as early as possible in a project since this often affects the approach taken and the modeling.

The Development Domain

The development domain is the organizational context in which a project team works. This can be an in-house IT department, a contracted software house, or a vendor of prefabricated software components. We do not take into account developers of standard software products that work for an "anonymous market," because a fundamentally different approach is followed in many of those areas.

Neglecting the development domain doesn't pay

The development domain is often given too little attention during planning. Yet the objectives of the development team members and their strategic orientation are very important. If the use of new technologies is being planned, then a provision has to be made for further courses and training of the team. However, this usually involves more than just new development tools and programming languages. What is often overlooked is that good developers are the most important resource of any IT company. The more qualified these people are, the easier it is for the IT department to handle the central task of management.

A big problem is created when project teams are simply made up of the people who just happen to be available and no clarification is provided as to the domain-related and technological profile of a project and who can match it. We deal with the special importance of domain knowledge in Sections 13.1 and 13.2.

12.6 PROJECT PLANNING CONCEPTS AND TECHNIQUES

12.6.1 Project Calibration

Many books on project management and the Unified Process use an "average" project, a project that develops an application system from scratch, as their starting point. A closer look shows that these books present a general pattern for project planning and execution. We are able to differentiate this project pattern for the T&M approach. The key dimensions are runtime and project scope.

Classification of Projects

The following classification provides guidance on what is needed to plan different types of projects:

Large projects

- *Large projects*: At the top end, we see new developments or the reorganization of complete application landscapes. An example would be the redesign of a complete banking system for distributed and multichannel applications. This sort of project involves a family of systems with thousands of classes that are revised over a period of several years, gradually replacing a complete existing application. The planning and organizational effort for such an undertaking will incorporate all techniques described in this chapter.

Medium-sized projects

- *Medium-sized projects*: At the middle level, we see the implementation of new workplace applications and more substantial application components. The scope of such projects lies in the new development of more than one hundred nontrivial classes, and the time requirement is at least six months. Here trained teams may work in a more or less familiar environment. The planning required is still considerable, but due to the partially calculable risk involved and the experience of the team, the number of very complicated or "heavyweight" techniques can be reduced or only used in isolated cases.

Small projects

- *Small projects*: At the bottom level, a minimum planning effort is required to implement individual components and small applications. The tools of eXtreme Programming in their "pure form" usually suffice. The short project runtime of a few months and the scope of less than one hundred classes justify these simpler planning techniques.

Consequently, the first (logical) planning task for every project based on T&M is calibration. This means the selection of meaningful planning and control means that match the concrete project.

Dimensions of Project Calibration

The key dimensions of calibration have already been mentioned.

- The scope of the project is related to its runtime and the size of the team. The longer a project runs and the more people needed to participate, the more complicated is the planning.
- The scope and complexity of the product: the more design and construction units the system being developed comprise and the more nebulous the domain-related and technological requirements are, the more extensive is the planning.

The following can serve as modifiers of the dimensions just listed:

Important aspects of project calibration

- *Strategic importance of a project*: The more strategic importance a project gains, the less management can deviate from specified objectives or identify trade-offs for the performance characteristics of the application system. The planning effort should be increased accordingly.
- *Team experience*: The more experienced a team is in the area of application development in general and in the existing domain in particular, the better it can assess and organize a project. Explicit planning techniques can then be used on a reduced basis.

- *Stable application context*: The more precisely an application domain can be modeled and the more stable the application context of the future system is, the less the deviation between the actual and projected state in the respective iteration cycles. This can also produce a noticeable reduction in the amount of planning required.

The difference in the amount of planning required is reflected in the precision of the planning numbers and the scope and quality of the pure planning documents. This has little effect on the quality of the application system.

12.6.2 Project Goals

It is fundamental that the management of a development process elaborates the aims for a project. As a vision of the future system, the project goal is the driving factor in the project. It will change during the course of the development project but must be outlined at the beginning so that the participating groups have the proper common orientation.

Formulating Goals

The main questions that must be dealt with when goals are formulated are:

- What is the meaning and purpose of the software project? In other words, which corporate objectives will benefit from the development project?
- What goals are to be achieved in the application domain? Do they have to be achieved all at once, or can steps towards achieving them be identified?
- Which information technology tools are suitable to meet the goal and to what degree? Which alternatives (e.g., which organizational measures or available standard systems) are conceivable?
- Who determines when the goals have been achieved? Is it one person or interest group or several?
- What kind of time frame is required to achieve the goals? How flexible is this time frame?

Based on the specified set of goals, we plan the development project using plans for the different stages, iteration plans, and base lines starting timewise from "back to front," that is, from the time targeted for project completion back to the time the project began. The basic idea is to work out the steps necessary to reach a subgoal.

Example

The Bank project

At the beginning of a bank project, we were able to identify the following aspects for the general aims:

- The corporate customer business of the bank was to be restructured in such a way that a profit of $1 million would be produced per year in this line of business.
- As part of the restructuring of the corporate customer business, the core processes were to be supported by an interactive workplace system. The planning and design of this workplace system was to be handled initially by the in-house IT department. If necessary, suitable software available on the market would be bought as an alternative.
- The board and the department management wanted to use reviews to determine whether their objective was achieved and what contribution the

software system made towards it. In addition, the management of the central development department was to check the technical quality of the software system.
- In terms of the timetable, the plan was to complete the restructuring of all business in this area, along with new IT support in all branches, within two to three years.

The subgoals were broken down as follows after discussion with application management:

- In domain terms priority was given to restructuring the loan business.
- Within the loan business balance sheet analysis was seen as the area that could profit most from IT support.
- The developer team received instructions to design a workplace for corporate customer advisors that initially would cover balance sheet analysis.
- The time frame for the first pilot system was specified as one year.

12.6.3 Decision Principles

Along with a shared vision of goals, a common repertoire of principles governing decision-making is helpful in the planning and execution of the individual steps. Such principles can affect risk handling, the form of cooperation in teams and with outsiders, and the technical aspects of the implementation.

Discussion

A team that works and makes decisions on the basis of shared principles is more flexible in delegating tasks. Observance of these principles ensures that the results follow a uniform view and structure.

While clearly defined project goals provide an orientation of the meaning and purpose of a software project, well-defined principles ensure that a uniform picture is created of the approach taken and joint actions. Typical principles at the overall project level include:

Principles of decision making

- "Buy before make."
- "Ensure platform-independence."
- "Reduce the total cost of ownership."

Although these principles are open to interpretation and thus are not solid conditions, they provide the scope of decision-making for a project. They can often be derived from IT or corporate strategy.

Other principles consider the special situation of a project in the context of political forces and technological requirements. Examples include:

- "Risks first": Complete the riskiest tasks and evaluate the biggest technical issues first (as recommended by the UP).
- "Water principle": Avoid confrontations with suppliers and plan defensively to avoid obstacles on the path.
- "Look ahead but do not rush ahead": Concentrate on essential requirements. Do not implement project parts until concrete requirements exist and are needed.
- Use of other eXtreme Programming principles.

Guiding metaphors and design metaphors (see Chapter 3) that also highlight a common view for project planning are supportive for the definition of goals and principles.

12.6.4 Project Establishment

Each new project starts with an initial meeting. How well a team will be able to work together is often already determined at this meeting. Establishing and initializing a project explicitly makes it clear that the right start is important for how the project proceeds.

Discussion

The start of a project has a particular impact on its success. If all the participants are given the impression that they are taking part in a viable project with clear objectives and that the project will be jointly executed by a team, chances are high that it will be completed successfully in the timeframe planned.

Establishing the "rules of the game"

A project is explicitly "established" so that a high level of motivation can be achieved by the participants. The fundamental principles governing the project are clarified while the project is being established. The project establishment is often initiated in a kick-off meeting, where the project goal is defined or explained in detail. Existing deadlines are clarified, the infrastructure is fixed, and some initial rough planning takes place.

It is a good idea to clarify cooperation rules and discuss the project culture. The project culture includes informal things like dress codes when dealing with customers and users, project dinners, and dealing with deadlines. The actual project content plays no role in this; only the project goal can hint at the content.

Along with the planning activities, project establishment involves defining the level of cooperation that exists between project members. To motivate team spirit in a group, DeMarco and Lister proposes arranging joint activities such as excursions and dinners.

12.7 STRUCTURING A PROJECT BY SYSTEM DECOMPOSITION

As part of the process of defining the goals for a project, clarification is needed about which part of the system should be realized by the project. It is rarely advisable to implement an entire application system with all its envisaged functions in one project. Instead, the target system is divided into subsystems that are realized in a step-by-step process. We present the concept of a core system with extension levels and special-purpose systems.

12.7.1 Core System and Special-Purpose Systems

The evolutionary development of application systems does not only mean that software is implemented in close author-critic cycles. With large application systems it is just as important that a complete system be developed in manageable units. The concept of a core system and special-purpose systems provides an initial guideline for the domain segmentation of large and complex application software.

The Core System

An open *core system* can be defined once general objectives and subgoals have been finalized. The core system should be coordinated with representatives of the departments concerned. This agreement on a core system is an important factor in the success of complex development projects.

A *core system*

- **is a productively used part of the total system.**
- **fulfills prioritized subgoals of the project.**
- **supports the key tasks in a department (e.g., a product domain; see Section 9.2.2) or in closely cooperating work groups.**
- **includes components (tools, materials, and automatons) to provide a set of elementary services that can be used for further development.**
- **deals with acute demands (e.g., in accordance with legal and business management requirements).**
- **supports the incorporation of special-purpose systems.**

Special-Purpose Systems

A core system can be enhanced with special-purpose systems.

A *special-purpose system*

- **includes components to support largely independent tasks.**
- **interfaces only to the core system and not to other special-purpose systems.**
- **can be developed independently and in parallel with other special-purpose systems.**

Alternatively, special-purpose systems can be developed by other manufacturers or can be bought as off-the-shelf components. The time sequence for introducing the different special-purpose systems is determined by the users. Consequently, application orientation is an important aspect of project planning for these systems.

Discussion

The basis for specifying a core system and optional special-purpose systems is a common understanding of the key tasks and division of labor in the application domain. Decomposing the target system into different components can take place while the project is being established. It can also be entrusted to a higher-level committee that allocates subtasks to the individual projects. It has proven useful in new projects to partition the overall system into the core system, special-purpose systems, and the extension levels within the context of a "pilot project." In addition to decomposing the system in as short a time frame as possible, the "core" team also builds a presentation prototype. This prototype serves as the design outline for the actual project work.

Domain knowledge and task orientation are crucial for system decomposition

The specification of the core system and the special-purpose systems is therefore a task that is outlined but cannot be completed in its entirety at the beginning of a project.

Example

The core system in a hospital project essentially consisted of patient administration and billing, the corresponding design of workplaces at the wards in the medical and nursing areas, as well as basic services for ward communication with different service suppliers, such as radiology, laboratory systems, and kitchens (see Figure 12.13).

Zull-12.qxd 20/8/04 7:46 AM Page 434

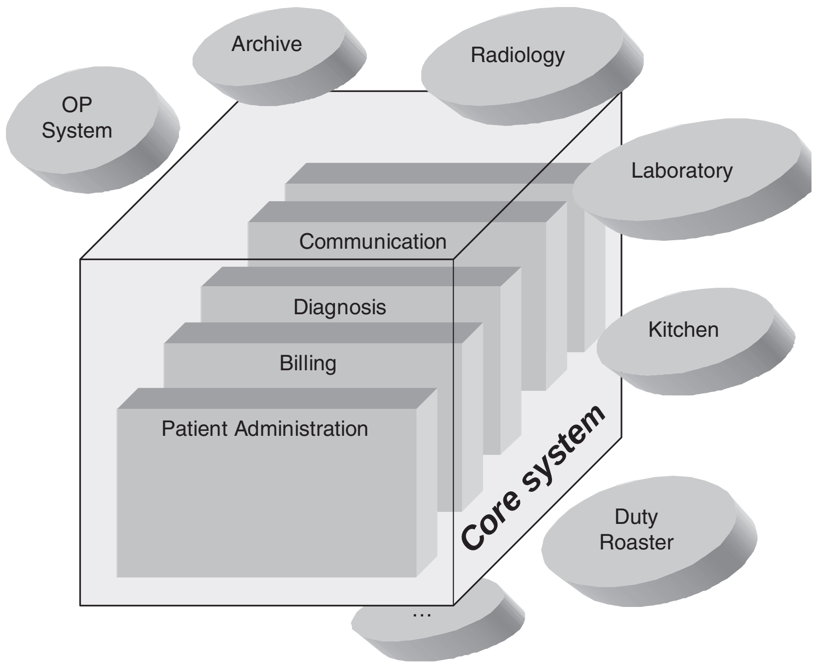


FIGURE 12.13

A core system with special-purpose systems.

Providing these basic services as separate components for the cooperation between wards and function areas could ensure that the cooperation and coordination between ward workplaces were uniformly designed. Since new legal rulings require a strong link between administrative and medical data, we saw the need to integrate patient administration and billing with the clinical areas in the core system.

Due to the large number of requirements from the different departments and areas, we found that there was a need for a core system enhanced by several special-purpose systems. The introduction of the core system contributed greatly to involving all participants. They agreed on shared project goals and the different extension levels. This provided a good basis for the planning and execution of the hospital project.

12.7.2 Core System and Extension Levels

The close connection between different departments and tasks usually makes a core system very complex. As a result, consideration is given to decomposing the system further, and, consequently, to the domain-motivated sequences for the development and planning process. We therefore divide complex application systems into a core system and extension levels.

Minimal Core System

When a core system cannot be realized "in one piece" due to its domain or technical complexity, then we have to divide it into *extension levels*. First we will define a minimal core system.

Identifying the minimal core system

A *minimal core system* comprises the minimal domain and software functionality that an application system should offer as services. It is the smallest autonomous and meaningfully executable part of the entire target system.

The following questions are helpful in specifying a minimal core system:

- Which task in the application domain has the greatest significance?
- Which task can be supported by a limited number of tools, materials, and automatons with minimal technical complexity?
- Which software support will bring the greatest benefit for the users?

A minimal core system also often has strategic importance, because it usually is the first real object-oriented application system, at least within the context of this approach. The benefits of a "quick win" should therefore be exploited. With a minimum of effort it is possible to demonstrate a major advantage through the introduction of a minimal system in the application domain. At the same time, the special character of the T&M approach should become clear through some well-designed tools with matching materials. Of course, the basic idea can easily be transferred to pure Web applications.

The Bank example

Example

In one of our projects, sections of a bank's loan department were to be restructured. We developed an object-oriented workplace system, which was the first of its kind in this bank application. Initially, there were many reservations about the project, particularly in the IT department. In search of a suitable minimal core system, we realized that the mainframe interfaces would soon not be sufficient for interactively processing loans. In talks with the loan department, we found that many loans were being paid out on the basis of handwritten forms. So we built a small electronic payment system with resubmission and copying functions. Although the tool did not offer any specific functionality for dealing with loans, it elegantly supported some of the tedious work in the loan department. The users were greatly in favor of similar support for the other segments of the loan-processing workplace.

Extension Levels

Once we complete our minimal core system, we have to analyze how the services designed for the entire core system are logically based on one another. To this end, we have to group these services and arrange them in successive extension levels of the core system.

First, let's look at the definition:

An *extension level* includes components that support related tasks and activities. These components should logically depend on one another, or on an existing extension level, but not on anything else.

Analyzing the logical dependencies between the components within extension levels, we can identify the domain-related sequence in which we can build the components of the system. Then we have to check if this sequence is feasible technically. Once we finalize the planning of extension levels, we can define the users and their tasks to be supported by the extension levels. At the same time, we have a schedule for providing the technical infrastructure of the application.

DISCUSSION

It is important for the concept of the core system and extension levels that we decompose the system according to a domain view, which is comprehensible to users. This integrates users and technical departments into the planning process.

A careful definition of the minimal core system with its extension levels provides developers and users with more clarity in recognizing and planning the consequences of iterative system development and deployment with the resulting intermediate organizational forms. At the same time, the extension levels support the task of planning accurate time horizons for a project and assessing the effort involved.

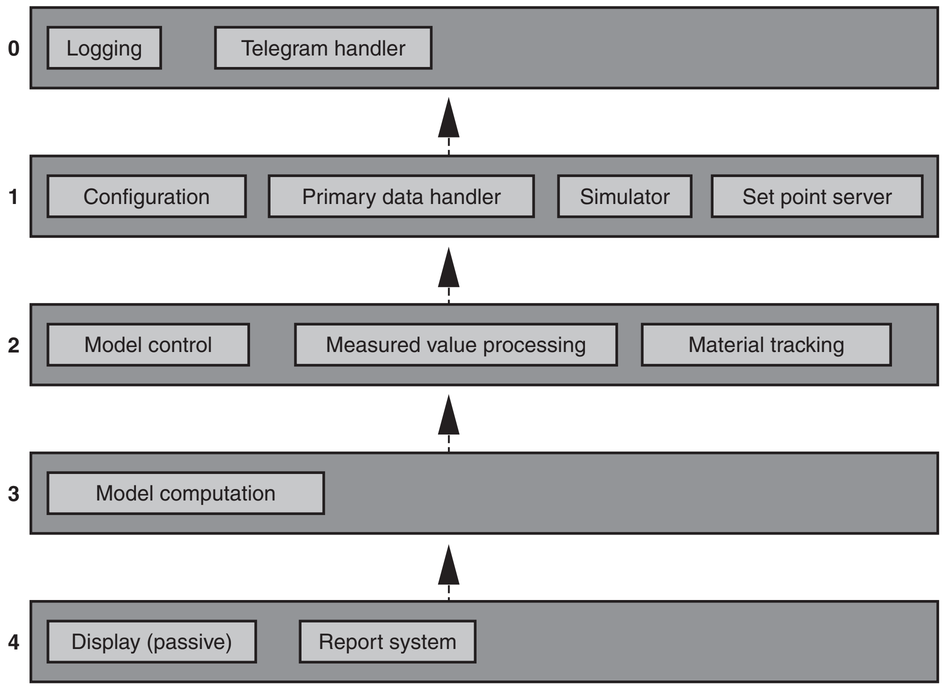
Ideally, extension levels are selected so that they can be deployed as executable versions or components, before the entire system is completed. At the very least, one extension level should always be realized as a verifiable prototype.

EXAMPLE

The decomposition into extension levels is obvious in domains with a traditional separation of different businesses (such as banks with active and passive business, securities trading, tellers, etc.). Figure 12.14 shows that this type of system partitioning can also be extended to domains with a totally different structure. The core system shown here was planned for the process automation software of a hot rolling-mill in five stages. The software components of the core system (level 0) are needed for the components being constructed in extension level 1, for example, the "primary data handler" is based on the "telegram handler." The component for model computation being developed in extension level 4 cannot be developed without the components of extension level 3 (reading value processing, model control, and material tracking).

FIGURE 12.14

A core system with extension levels.



12.8 SCHEDULING AND TASK PLANNING

How to approach scheduling and task planning

We have a list of the techniques and approaches for scheduling and task planning recommended for different types of projects and systems:

- On the basis of set goals and a partitioning of the system into a core system and extension levels, we recommend a process model and a time estimation for each of the system components or extension levels being developed. For the planning approach, we suggest a division into project stages and base lines.
- For very large systems or systems that are partitioned differently, we integrate project stages and base lines with the general concept of cycles and iterations of the Unified Process.
- At the micro level of large projects or for small and very flexible projects, we propose the planning instruments of eXtreme Programming.

12.8.1 General Rules for Time Estimates

An overall time estimate has to be presented for large projects. For large and complex projects, this time estimate can only be very rough and only provide an indication of what is feasible in terms of time. However, the estimate can refer to figures from previous experience or general criteria.

Discussion

An initial time estimate involves sketching the time frame in which the components of an application systems could possibly be realized. A discussion of the different imponderabilities in an application-oriented software project, such as those discussed in this book, should make one thing clear: It is not possible to present an accurate overall time plan of a large project. However, "rules of thumb" based on figures from past experience can be used for a rough estimate:

Rules of thumb for time estimates

- Each prototype development cycle takes between one and three months. The question, therefore, is how many prototype versions should be and can be developed.
- When using XP techniques, we calculate between three and six months for a release cycle with a delivered system version.
- Interviews and scenarios form an essential basis for every development project. Approximately one day should be planned for the interview team per interview, including the time needed for evaluation. Scenarios have to be written, read by the interview partners, and then jointly discussed. A scenario with an experienced interview team takes about one work week. Therefore, we have to estimate the number of interview teams required to produce scenarios, how many interviews and interview partners we will need, and to what extent the work can be carried out in parallel.
- The time needed for the design and construction of a workplace system can be estimated largely on the basis of tools, materials, and domain values. For example, for trained developers building software along the lines of the T&M approach based on a validated domain model, we calculate half a pair-day per average domain value, one to three pair-days per material, and two to five pair-days per typical T&M tool.

- Similar experience also exists for domain service providers and the connection to ERP systems. We calculate two to three pair-days for a domain service provider of average complexity, and one to two pair-days for database mapping. In our experience, host developers, who are usually relatively good at this, are the best at calculating the additional time needed for adjustments on the host system side.
- The organizational installation and field test of the pilot system also need a certain period of time that cannot easily be reduced. One should count on one to three months here.

Problems in estimating time

These simple rules of thumb (modified through the reader's own project experience) can be used to estimate whether the time expectations of management or the user organization are at all realistic. The time estimate obviously becomes less accurate

- when there is little familiarity with the respective application domain.
- when there is little familiarity with the type of application system being developed and the methodology and technology used.
- the more system components and extension levels are incorporated into the estimate.

This form of time estimation should not be confused with the normal scheduling in a project. It serves as a kind of feasibility test to determine whether a project can be carried out at all in the desired form.

12.8.2 Planning the Project Stages

If an application system has been suitably partitioned and a rough schedule planned, then the actual project planning is carried out on the basis of project stages. These stages define manageable project segments. They also link constructive quality assurance with an application-oriented approach.

PROJECT STAGES

In the Sections 12.7.1 and 12.7.2 on the core system and its extension levels, we stressed the importance of decomposing a complex application system into manageable design and construction units. If such a unit is defined as a core system or an extension level, the development process has to be planned in more detail.

We execute projects on a goal-oriented basis. When a goal for a project exists, we can decide which means, or which application system components, will contribute towards achieving this goal. It makes sense to define manageable subgoals and to work through them in a project stage.

***Project stages* represent important events that are "visible to the outside" in a project. They are specified with a view towards the overall goal and the project stage goals derived from it. They serve to define the scope of the system under development. A project stage is linked to an operative version or a prototype of the system under development, and it defines the date when the project stage goal should be achieved. Base lines (see next section) are used to plan each project stage in detail, supporting a precise control of the project. The evaluation of a project stage often leads to revisions in subsequent planning phases.**

Discussion

Planning backward

The completion date for a system under development is usually based on a rough time estimate for the project and management's own ideas on the subject. The basic idea of planning based on project stages involves starting from the projected completion date of a project and working *backwards* according to the different stages.

Using the goal as a starting point, we define the project stages working in the reverse. The idea of project stages is comparable to the Unified Process. In so far as operative versions are being developed, these in turn are usually realized in substeps as the core system or the extension level of an application system, and these substeps are revealed to all participants. The same applies when building "only" a prototype. A project stage therefore produces a result visible to all participants, and at the same time, it serves as a control instrument for further planning.

Finishing a project stage

Each project stage is concluded with an event, where a prototype or the system version is presented and evaluated. On this basis, the participants decide whether the respective project stage goal has been achieved. It should be noted that different kinds of prototypes are used in the respective stages, depending on the questions being dealt with.

There is another important difference from traditional management techniques: All documents generated to that point are checked for the current project planning. It is therefore common that feedback on a user interface prototype will clarify for users and developers whether the domain analysis with scenarios was sufficient or requires further detail. System visions are also repeatedly examined for their feasibility.

This examination can be a formal review with users or a simple presentation given to a project committee. The important point is that key groups and decision-makers are informed of the project's progress and made aware of possible problems.

Evaluating a project stage

At the conclusion of each project stage, the project plan is revised in terms of content and schedule. Various aspects are taken into consideration:

- Has the goal set for the project stage been achieved? If not, is it still basically achievable?
- What is the significance of the result of the project stage for the next stage? If the result was worse than expected, what effect will it have on the next project stage? Are modifications needed to the distribution of work, or is there a risk of not meeting the next stage's goal?
- What are the chances of still achieving subgoals or the overall goal with reduced means (i.e., system features) if the result for a project stage is unsatisfactory?

If revisions to the plan are necessary due to the evaluated results of a project stage, we again focus on the overall goal. Project teams should stick to this goal to the extent possible, and when problems and timing bottlenecks occur, consider whether the goal can be achieved using fewer or other resources. We have always had the same experience in projects: From the view of the users, a "thinner" application system with less features than the ones originally planned, or a reduced extension level, can well be an accepted and almost equally welcome solution to their current problems. A reduced system version delivered on time is preferable to a long delay in almost all cases.

The "size" of a project stage

Project stages should normally extend over a period of between six weeks and three months. A large number of short stages is not easy to manage and creates a hectic situation. On the other hand, planning is difficult if the intervals are long, and they also create problems for the author-critic cycles.

Team planning

In practice, it has proven useful to have the team discuss the planning of project stages. In projects with small teams, such plans can also be worked out jointly. With large project teams, the plan can be prepared and then discussed in a project meeting, revised if necessary, and adopted. In view of the application orientation we want to achieve, users should also have an influence on the plan. All deadlines that affect users should at least be agreed upon with them.

Example

Figure 12.15 refers to the example of setting goals (see Section 12.6.2) and shows how these goals can be implemented. This includes the definition of a minimal core system, which is now planned with the help of project stages. Our approach of organizing this plan from back to front is again important here. In this example, the completion date for the core system was set for 02-15-2002. All other deadlines are calculated from this date backwards.

Background: Project Stage Plans and Conference Planning

Why do we plan stages from back to front? There is a simple reason: We often find that some conventionally planned software projects exceed their time budget by far. In contrast, we have observed that most conferences actually take place exactly when and where they have been planned long before the actual events. Conferences can obviously be planned according to a precise date.

Why planning a project like a conference?

What is the big difference between conventional project planning and conference planning? We think that this difference has little to do with the fact that conferences are not software systems. Conferences are simply planned differently, namely, from back to front. Once a program committee has agreed on the location and date for a conference, then certain intermediate results or stages are tied to certain dates. For instance, if the conference proceedings are to be distributed among the participants, they have to be available in printed form at the organizers' two to three days beforehand.

FIGURE 12.15 Example of project stages.

Goal:	Implementation:	When:
Steering committee accepts reorganization concept	Presentation prototype plus scenarios, glossary, vision of account manager workplace	31.03.01
Central development accepts architecture	Breadboard with server and rel.DB	16.05.01
Steering committee accepts reorganized business process	Kernel system for the account managers workplace tested in one bank branch	30.09.01
Steering committee accepts feasibility of concepts	Extension level 1 Account manager workplaces in three banks in test operation	01.12.01
Controlling accepts system	Extension level 1 documented and tested	10.12.01
Extension level 1 accepted by steering committee	Evaluation workshop with the user Extension level 1 ready for roll-out	15.02.02

This means that the print masters have to be at the printer, for instance, one month before that date. This date, in turn, determines when the conference proceedings have to be edited for content, and so on. This is how program committee meetings, deadlines for completed papers, and the dispatch of call for papers are planned and scheduled. Any planning deviations are always noticed very quickly while a conference is being prepared. Organizers can then take suitable actions to ensure that the conference is not jeopardized. For example, lectures are deleted from the conference program or other speakers are invited on short notice when illness or another unforeseen circumstance prevents a scheduled speaker from participating.

On the other hand, we could argue that when we organize conferences we have a much more accurate idea of the steps and results required and how they depend on one another. However, our experience has shown that there is nothing against using these fundamental planning principles in software projects.

12.8.3 Using Base Lines for Detailed Planning

Planning according to project stages is still so rough that it is difficult to identify any work packages or responsibilities from it. Base lines are therefore used for fine planning within a project stage.

Base Lines

Planning based on project stages provides no information about the distribution of tasks and responsibilities. In our experience, this type of finely granular planning usually becomes necessary and reasonable at the level of the current project stage. This is where we use base lines.

***Base lines* are used in the detailed planning of project work within a project stage and in constructive quality assurance. They are defined by the development team and describe tasks relating to qualitative or quantitative document and code states. Base lines define a checkable result. Someone is responsible for each base line, which is validated on a regular basis. The respective developers estimate the work involved and the time required.**

Discussion

Base lines need to be checked

A base line primarily describes the quality expected of the next document or program. Ideally, documents and programs should be edited until the desired (and checkable) state is achieved. The same applies when events are specified in base lines. It is important that a base line identifies who will be responsible for ensuring that it is achieved. Though this person is responsible, he or she doesn't have to do the work on his or her own.

In the sense of our general approach, base lines define tasks. They bundle domain-related and goal-oriented activities relating to a document or system component. The respective tasks should be small enough that they can be completed by a minimum number of people within a manageable amount of time. This also results in an apparent contradiction: We said that base lines should only be oriented to quality characteristics and not to time. At the same time, we demand that time and work estimates should be linked through base lines. When base lines are distributed in a team, the team members responsible for a base line gradually develop a very good feeling about how long they will need to work on it.

Discussing base lines

Base lines should be discussed jointly in a team. The size of the team and the complexity of the project determine whether the base lines are also worked out by that team. In any case, certain means of representation are available. We have had good experience with wall pictures where base lines and personnel resources are displayed in tabular form. Depending on how they are arranged, these wall pictures show the logical connections between base lines and facilitate planning and discussion of the project scheduling (see Table 12.1 and Figure 12.16).

We can report from experience that the planning procedures described here have also been effective for projects with very critical application and technology domains. The cyclic approach and orientation to executable prototypes and system versions for project stages are obviously the key factors that make this procedure successful. The cyclic approach also helps projects dealing with imponderabilities. The concrete and presentable intermediate results motivate the team and demonstrate to "outsiders" that a project is progressing.

Base lines should be viewed as an alternative to the story cards of XP. They are perhaps somewhat more "conventional" than story cards and therefore easier to use in areas where the principles of eXtreme Programming cannot be applied.

Base lines are also used in the Unified Process. Here they are interpreted in a much more conventional way, because they are "frozen" at a certain point, and an explicit procedure is needed to change them again.

The Bank example

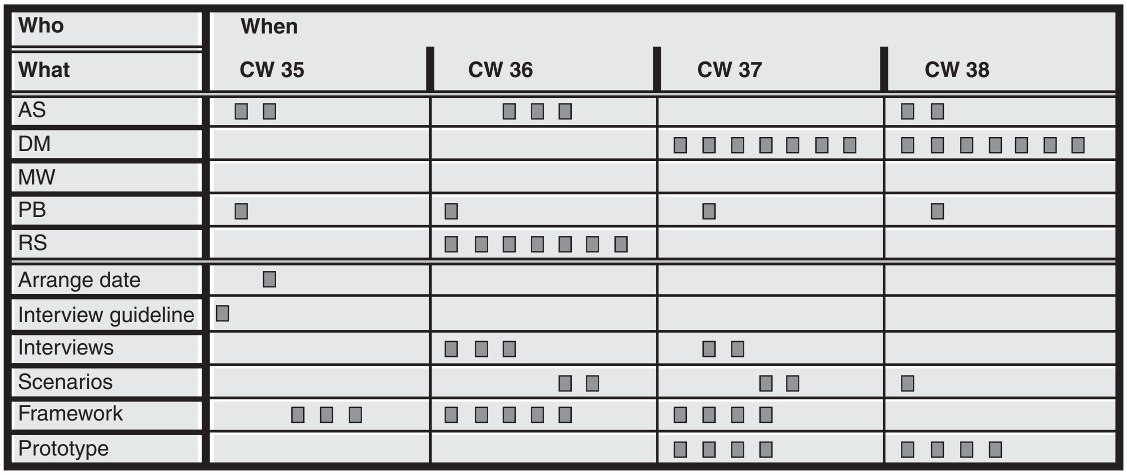
Example

Let's take another look at our previous example of a bank's workplace system for corporate customer advisors. The base lines in Table 12.1 are listed as a kind of purpose table (see Section 13.8).

The list always includes the names of the people responsible (using abbreviations), the base line, the purpose, the person-day estimates, and the type of checking used for the base line. The form of presentation is not what is important here; it should only indicate what has to be taken into account when the base line is worked out. We

Who	Does What with Whom/What	Why	p.d.	How Checked
PB	Arrange date with pilot bank	Preparing interviews	1	e-mail to team
RS	Elaborate interview guideline	Preparing interviews	2	Presentation at meeting
PB	Interviews with account managers	Basis for scenarios	10	Minutes in project db
DM	build tool framework	Preparing tool implementation	16	breadboard prototype of tool operative
AK	design and implement tool for call slip	Test of framework preparation of user workshop	8	breadboard prototype of tool operative

TABLE 12.1 Example of base lines.



Who	When			
What	CW 35	CW 36	CW 37	CW 38
AS	▨ ▨	▨ ▨ ▨		▨ ▨
DM			▨ ▨ ▨ ▨ ▨ ▨ ▨	▨ ▨ ▨ ▨ ▨ ▨ ▨
MW				
PB	▨	▨	▨	▨
RS		▨ ▨ ▨ ▨ ▨ ▨ ▨		
Arrange date	▨			
Interview guideline	▨			
Interviews		▨ ▨ ▨	▨ ▨	
Scenarios		▨ ▨	▨ ▨	▨
Framework	▨ ▨ ▨	▨ ▨ ▨ ▨ ▨	▨ ▨ ▨ ▨	
Prototype			▨ ▨ ▨ ▨	▨ ▨ ▨ ▨

FIGURE 12.16 Scheduling with base lines.

can see that base lines can be planned very precisely, involving from one to a few person-days in this example. Of course, this type of finely granular planning is only recommended for projects with tight deadlines and a high degree of uncertainty. On the other hand, experience has shown that it is not a good idea to have large work packages, because it makes estimating them more difficult, and there is not enough feedback to control the planning.

To carry our example a step further, Figure 12.16 uses the base lines of Table 12.1 as the foundation for detailed scheduling. The calendar overview, in this case divided into weeks, shows the initials of each team member. The boxes are used to fill in "absences" from the project: days off for vacation, committee meetings, or other reasons. This part of the overview is usually generated early on in a planning meeting and often shows in a sobering way the personnel resources available.

The individual base lines are then entered in the bottom part of the calendar, which shows when these base lines have to be handled. For the sake of keeping things simple, we left out cross-references between the bottom and top parts in Figure 12.16.

12.8.4 The UP and T&M Project Planning

So far, our project planning has been motivated primarily from the standpoint of system partitioning into core system and extension levels. However, this planning instrument can also be integrated into the general Unified Process approach. This means that this technique is also appropriate for very large projects or systems that are partitioned in a totally different way.

UP Evolutionary Cycles

At its highest level, the development process using the Unified Process allows a random number of evolutionary cycles that an entire system will pass through during its lifetime.

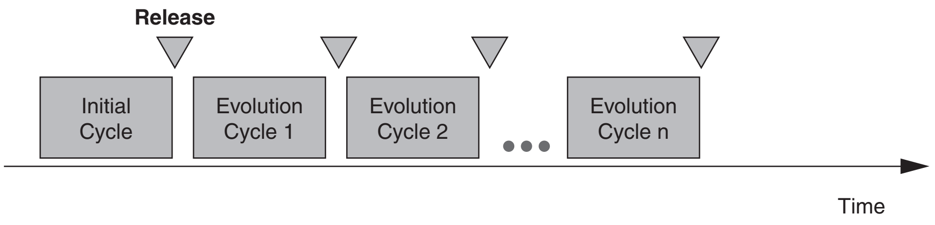


FIGURE 12.17
Evolutionary cycles.

An *evolutionary cycle* is the least accurate unit of scheduling and planning in a development process. Each cycle is organized as a project within a project family. A cycle ends when a system release is complete and deployed. Cycles are divided into stages.

A totally new system is developed in the first cycle, similarly to conventional development projects. The development of this system is continued in an evolutionary way over its entire life cycle. Our approach organizes this continued development in projects, that is, in the sense of UP, we talk about evolutionary cycles (see Figure 12.17).

Evolutionary cycles based on UP normally consist of four phases[1] that can be mapped to our concept of project stage.

A *stage* is a self-contained subproject in the development process; it has its own goals and produces an executable system version. Each stage comprises all necessary development activities and its structure follows its own project stage plan.

Looking across projects, each stage according to T&M has concrete goals and particular topics: conceptualization, design, construction, and transition. We see them as subordinate topics rather than steps like those in a waterfall model. Thus the topic "design" is produced through activities that analyze, model, and construct things within a stage. The following section discusses these topics.

Discussion

Interpreting the UP

This section interprets parts of the Unified Process from the view of the T&M approach. We give priority to a cyclic and iterative approach in projects. Other authors see this differently. For example, the literature often recommends defining the number of cycles and work steps precisely, which means that these authors encourage classic life cycle models.

Note that this section is not geared to "normal" development projects. Instead, we examine the long-term evolutionary process of developing a large system, or better, a family of applications in the course of several projects. An evolutionary cycle corresponds to the development of a system version, or major system part. Each individual project that contributes to the new version can be oriented towards general issues: conceptualization, design, construction, and transition (see Figure 12.18). The results of these subprojects are merged to create a complete system family at the end of the evolutionary cycle. The goal of creating a uniform system family can only succeed independently of organizational circumstances under the following conditions: if project

1. In UP lingo, these phases are called *inception*, *elaboration*, *construction*, and *transition*.

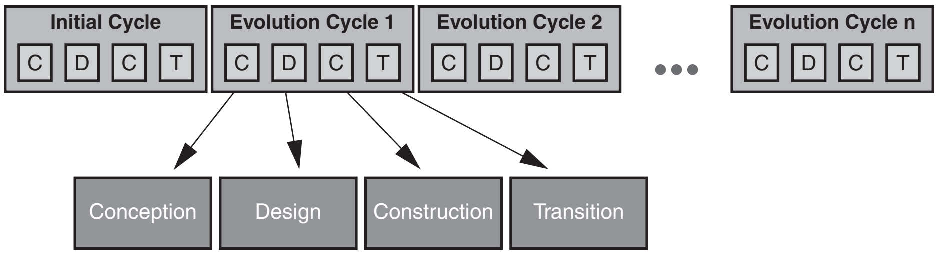


FIGURE 12.18

Stages in a T&M project.

members keep sight of the common basis of the subprojects and have a clear understanding of the fact that they all contribute to an existing whole.

THE UP PHASES

Interpreting UP phases as project stages

At the next level a T&M project is organized in stages. This is our interpretation of what is called *phases* in the UP. Each stage within an evolutionary cycle has its own goals. It is organized as a separate project. This means independent and consistent planning. It is important that each of these subprojects includes all activities of a software project, or the construction and evaluation work. This is in clear contrast to the steps in the classic waterfall model.

A concrete stage has project-specific goals that cannot be described in general terms. However, general issues or themes are abstracted to these goals. We explain these general characteristics of the individual stages below. The document types and tasks relevant for project stages will be described in more detail in Chapter 13.

The *conceptual stage* creates the concrete product idea for a new project. It can refer to a new system or a functional extension of an existing system. The objective of a conceptual stage is to examine the product idea for domain-related consistency and basic technical feasibility.

The conceptual stage describes relevant business use cases (actual state), identifies initial visions for the future system, and builds the first prototypes. These will initially be presentation prototypes. If possible, however, they should be extended to include functional and technical aspects in the form of a T-prototype (see Section 13.6). A rough range of system features and the technical basis for the project should be finalized by the end of the conceptual stage. In addition, a description of the key (critical) functions should be available. Cooperation pictures should be used to clarify the planned cooperation model. The results of the conceptual stage set the foundation for more accurate planning in the design stage.

The *design stage* elaborates system features and resource requirements for a project more precisely. The objective of the design stage is to map the domain requirements to an architecture, observing structural similarity.

Planning during the design stage is mainly based on the range of features defined in the conceptual project stage. Further system visions and other documents of the target model are specified during this stage, until the desired range of features is covered, and the project is ready for evaluation. The system visions are converted into domain and technical design models and realized through prototypes. At the same time, domain and software issues are further clarified. This stage focuses on providing fully operational prototypes and on specifying a logical architecture for the target system. We then use the results of this stage to confirm what we said in the beginning of this section, that this kind of decomposition into project stages is useful for large development projects only, because its high cost would not be justified in small projects.

The *construction stage* primarily deals with the construction of technical models, with particular emphasis on the implementation model. The objective of this stage is to realize the architecture at the level of processes and implemented components.

The construction stage uses unit tests to check the prototype resulting from the earlier stages to assure its quality. This is the last stage where things like the assertions of the contracts can be checked for completeness.

By now, the developing system should have matured so that certain questions regarding performance or embedding into the context of the system can be answered. Constant feedback, above all on the functional prototypes and pilot systems (see Section 13.6), helps to make sure that the scope of system features corresponds to the actual needs and ideas of the group involved.

The *transition stage* transfers the product to the actual users, including deployment, training, user support, and maintenance. The objective of this stage is the organizational implementation of the system.

If possible, the organizational implementation of the system should include the use of pilot systems. Experience has shown that domain inconsistencies and implementation errors can often be identified in this stage. These errors usually result from the special system configuration on site and are particularly unpleasant. The end of this stage completes the entire cycle of the four stages described in this section.

PROJECT STAGE PLANS

A *project stage plan* contains the concrete planning for a cycle. It is directed towards the concrete subgoals of a project. In general, the project stage plan should identify the specifc topics relating to the four general stages of conceptualization, design, construction, and transition.

Along with the goals for the different stages, the project stage plan contains the profiles of the project members and information about the resources required over the course of the entire cycle. Figure 12.19 shows that resources are distributed in different ways. In addition, the project stage plan contains rough details about the duration of the iterations and the subtasks already associated with them. The project stage plan is created early in the conceptual stage and edited as often as necessary. It shouldn't be more than two to three pages long.

Profiling a project

Obviously, the length of the individual stages is not the same in each project. The starting point for a project stage plan is a typical project profile with, for example, the following characteristics:

- The project is of a moderate size and its cost is justified.
- The project is in the initial cycle.
- The project does not have a predefined architecture.
- The project has a limited number of risks and unknown factors.

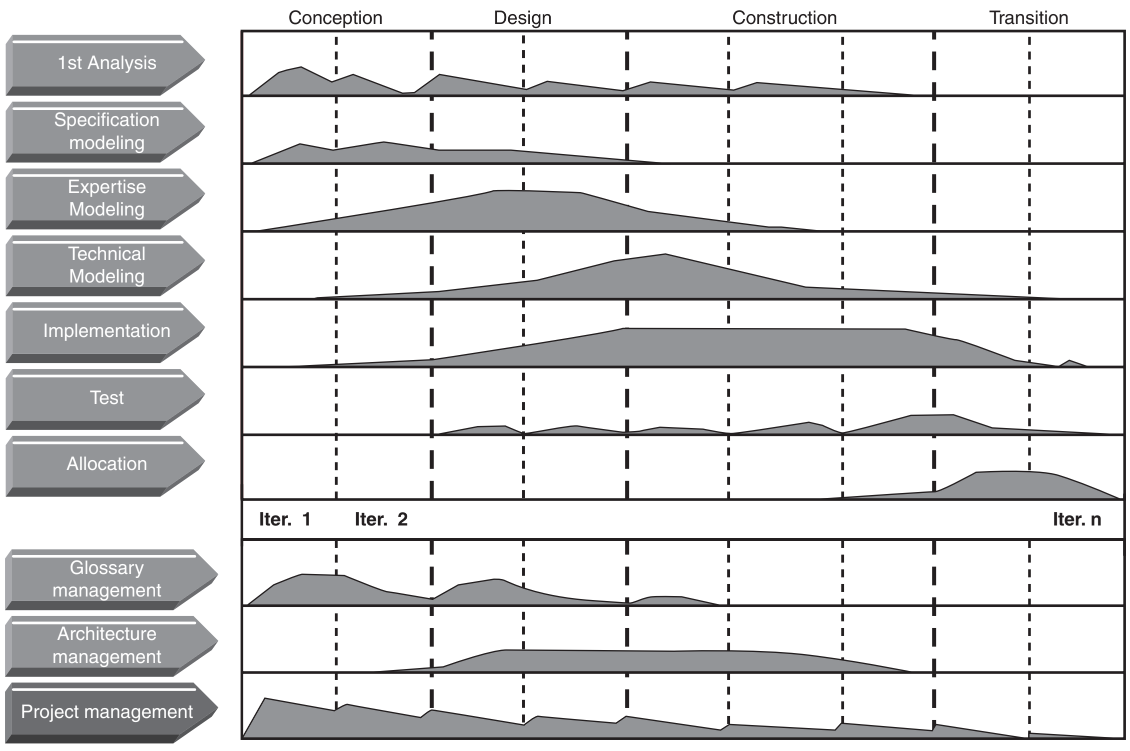


FIGURE 12.19 Distribution of resources in a project stage plan.

This basic project profile can be adapted to a specific project, based on the following rules:

Rules for adapting a project

- If the actual tasks are still not clarified at project start, that is, the requirements are not clear, then the conceptual stage may have to be extended.
- If an appropriate architecture cannot be found because some things are still unclear regarding system design, cooperative work is being supported, or a group of new employees is to be integrated, then the design stage has to be extended.
- If new technologies are to be used, a distributed system is performance-critical, a high level of parallel system access is expected, or a large number of technical problems exist, then the construction stage should be broken up to address each problem specifically.
- If the second generation of an existing product is involved (a new evolutionary cycle) and no far-reaching modifications are required, then the conceptual and design stages can be shortened.
- If a product has to be available on the market quickly, either because the development is seriously behind schedule or the customer wants to reshape the market, and the product has been announced for a specific shipping date, then the construction stage can be shortened and the transition stage extended.

- If the project involves a complicated installation, for example, replacing an old system with no downtime allowed, or if the acceptance procedures are particularly complex, then the transition stage has to be extended.

Discussion

Project stages

This section generalizes the concept of project stages and maps it to the general process model of the Unified Process. Section 12.8.2 defined project stages as being goal-oriented and producing an executable system vision or a prototype as the result. Stages are therefore very suitable as planning units in projects where the system is divided into a core system and extension levels. However, this type of partition becomes superfluous when a large system with the expectancy of a long span is reengineered (or refactored) on an evolutionary basis. In this case, the system already exists with a full range of features.

In this section we therefore orient the project stages towards more general topics. Each stage is planned as a separate project and implemented with all meaningful activities. Therefore each stage has a project-specific objective. The four topics, of conceptualization, design, construction, and transition should make clear that the results of the stages can always be directed toward different general issues. This does not mean that each stage only produces one concept, works out one design, constructs one program, or ships one product. This obvious deviation from the principles of the classic waterfall model is common to both the Unified Process and the T&M approach. We are pleased to see that an evolutionary and iterative approach is gradually catching on.

Iterations

Each of the stages discussed can be divided into several iterations. All necessary tasks are executed within each iteration in order to promote the maturing process of the development documents and the code.

An *iteration* is the smallest unit that can be planned in the development process. It is defined by a consistent set of base lines. Depending on the size of the entire system, an iteration can end in a prototype or a system version or lead to a sequence of builds or integrations based on the principles of eXtreme Programming.

An iteration can be interpreted as a complete miniproject, where all key development tasks are executed. The result of an iteration is an executable system version, usually a prototype, and a number of documents. An iteration therefore starts with a planning and requirements analysis and ends with an internal integration or an external release. We use the base lines of the T&M approach to identify the documents and prototypes generated in each iteration, their purposes, and the people in charge of their validation.

Discussion

The scope of an iteration

An iteration represents the finest granularity of incremental software development that can be planned. The scope of an iteration depends heavily on the project calibration. It only pays to embed true miniprojects with their own planning horizons within a stage if project families are arranged at the top end of the scale, that is, if they are to run for several years.

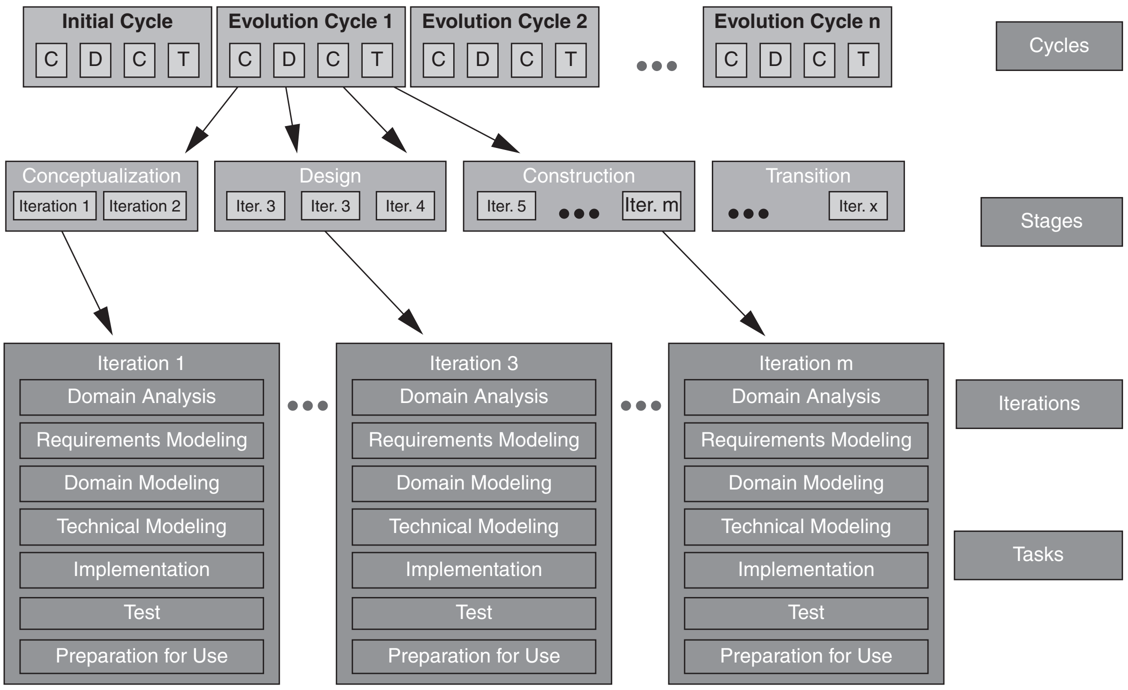


FIGURE 12.20 Example of using T&M iterations.

The sensible length of a full-blown and planned iteration within a stage is one to three months. We assume that a development process based on the principles of eXtreme Programming (see Sections 12.3.4 and 12.3.5) should be selected for an iteration duration of less than one month.

The iterations of medium-sized projects with an iteration duration of less than one month are reduced to the scale of what is referred to as short releases in eXtreme Programming and supported by planning games (see subsequent Section headed "Using XP for Detailed Iteration Planning"). Small projects of short duration can completely dispense with iterations within a stage since the entire stage can be planned and controlled at the planning games level.

Development activities are carried out with varying degrees of intensity, depending on the stage, number of iterations, and problems. Though there can be different focal points, all tasks pass at least one iteration (see Figure 12.20). As a consequence, all document types and models are basically at hand for processing. Each document or model is then handled with the intensity defined by logical and individual priorities.

Regarding content, each iteration is driven by selected visions or use cases selected from the range defined for the system. After each iteration, a set of base lines is checked to enable the detailed planning of the next iteration.

ITERATION PLANNING

The individual iterations are planned with the described base lines (see Section 12.8.3). Normally two iteration plans are important in a project in parallel:

- The current plan (for the iteration currently being run) for process tracking.
- The direct successor (for the next iteration) of the current plan that is approximately begun midway through the current iteration and has to be completed by the end of the current iteration.

USING XP FOR DETAILED ITERATION PLANNING

Story cards

We use the *story cards* from eXtreme Programming to make work orders easier to manage and to establish new requirements. Story cards are file cards used to document the requirements from the users' view, similar to use cases in UML. In XP, customers and users write the story cards themselves, which can create problems. We think that this method doesn't work well, unless both the customers and users are familiar with the requirements. Moreover, they are often not able to evaluate the technological options available. Therefore, developers using the T&M approach normally write the story cards based on their understanding gained from customer and user interviews. The story cards are then given to customers and users for comments.

Developers working in pairs (see "Pair Programming" in Section 12.3.4) often find that some parts of the system require refactoring. If this need for refactoring is extensive, then the developers can write their ideas on *engineering cards*. Engineering cards are like story cards, except that they refer to technical aspects. In addition, *task cards* are often used; they are similar to story cards, but go beyond a specific story. Tasks that are useful for several stories can therefore be extracted from them.

Engineering cards and task cards

Using story cards

Developers use these cards to plan individual iterations by sorting the cards based on usefulness and urgency in cooperation with the customer. Each card describes certain tasks and can be used to check a base line. The achievement of a base line is controlled on the basis of the cards assigned to this base line. For small to medium-sized projects, the story and engineering cards can simply be used as a substitute for explicitly formulated base lines. The cards are then directly allocated to the project stages.

Although story cards can be used for small projects, they are also a good means for coordinating several application projects with the development of a framework used. Story cards are excellent planning and control mechanisms for small projects, where the full set of system versions, core systems with extension levels, project stage planning, iteration plans, and base lines would be too extensive. Requirements are documented on the story cards and ranking by the customer according to usefulness and urgency. The developers take the story cards to identify the construction tasks to be handled, if necessary write task cards, and plan the individual tasks. These are then processed in a way that allows the tasks to be checked. The story card method follows a basic idea of the Unified Process to "define requirements when they become visible," while the "risk-first" principle applies to task planning. Finally, the set of defined requirements is used to check whether they have been met. The Unified Process offers use cases rather than story cards for this purpose.

Story cards are also used when several application projects work parallel to the further development of a framework. The framework developers use story cards to write down new ideas or document the requirements of the framework users (namely, the application developers). In regular architectural meetings, the product manager and framework users sort these story cards according to urgency, discuss individual problems and solution approaches, and distribute the work among the framework developers.

This ensures that the requirements of the application developers are documented and the requirements from the individual projects are compiled and examined in an overall context.

12.9 DISCUSSING T&M, UNIFIED PROCESS, AND XP

The evolutionary approach of T&M can be seen as an application-oriented interpretation of the Unified Process. At the same time, we integrate the techniques and presentation means of eXtreme Programming. The basic principles of UP coincide with the ideas of XP and T&M. A shift in emphasis occurs when it comes to the question of complexity and type of documentation and the size of the respective planning units.

12.9.1 Structure of the UP and T&M Development Processes

UP cycles

UP and T&M are similar in the size of planning units. A development project based on UP is divided into cycles, phases, and iterations. We know from Section 12.8.4 that the system versions and extension levels of a family of application systems can be developed in the cycles defined in UP. While UP looks at external and internal releases including increments, the T&M approach shapes core systems, special-purpose systems, and extension levels from an application-oriented view.

UP phase

In UP the phases are the next planning and control level. They can be seen as the abstract units of what we call stages. In UP, phases are subdivided according to the abstract topics, which are inception, elaboration, construction, and transition. The stages in T&M are always oriented towards project-specific goals. When we look at them in abstract terms, we see general topics. With a light shift of emphasis, compared to UP, we call our stages concept, design, construction, and transition stages.

UP iterations

In UP, systems are developed cyclically in iterations (see Figure 12.21). We have shown that the iterations of UP can easily be divided and planned through the use of the T&M base lines.

UP workflows

We see from Figures 12.21 and 12.22 that the UP and T&M iterations are similar, except that the T&M approach emphasizes the activities. As suggested by the workflows in UP, we first assume that all types of activities generally have to be done in parallel to complete these tasks, and then repeated in each iteration, while the priorities and actual sequence vary from one project to another. To better understand this basic idea, both figures list the activities in columns instead of a horizontal time sequence.

The main differences

The main differences appear in the number and type of activities. A brief comparison clearly shows this:

- *UP: Requirements, analysis, design, implementation, test.* These activities are based on the traditional steps of the waterfall model. They are primarily seen from the view of the developer. The UP authors suggest that these activities serve as examples and can be divided up differently.
- *T&M: Domain analysis, requirement modeling, domain modeling, technical modeling, implementation, testing, preparation for use.* T&M focuses on both application-oriented and technical activities. At the same time, preparation for

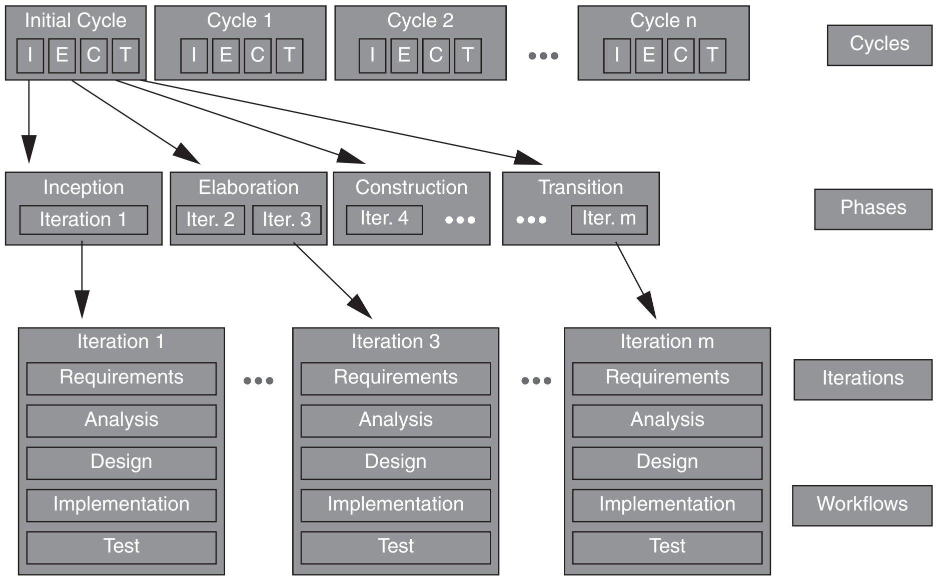


FIGURE 12.21

Example of UP iterations.

use is considered an explicit activity. A great deal of importance is attached to actual domain modeling.

Discussion

T&M is application-oriented

At the conceptual level, the T&M process model can be easily mapped to the general UP development process. The concretization in the T&M approach is shaped by application orientation. Domain analysis, requirement modeling, domain modeling, and preparation for use are activities that explicitly include the users and run in author-critic cycles. A characteristic feature is the strong focus on domain analysis and modeling. Although this is also found as a business model in UP, it is not as important there and not as integrated in the further modeling process.

The T&M approach works on the principle of structural similarity. This means that the terms and concepts of an application domain first have to be understood on the background of day-to-day tasks in that domain. Sound understanding of the actual work situation remains important for developers, even if tasks and processes change with the introduction of a new application system. This is the best way to ensure domain knowledge throughout a project to assess potential modifications and implement them into the system design.

Models in the Development Process

The models in UP

The different models in UP are allocated to workflows according to the breakdown of the development process, as shown in Figure 12.22.

We can see the following important points from Figure 12.22:

- Although the business use case model and the business object model are both specified in the UP book, they are not given the importance that other models are awarded.

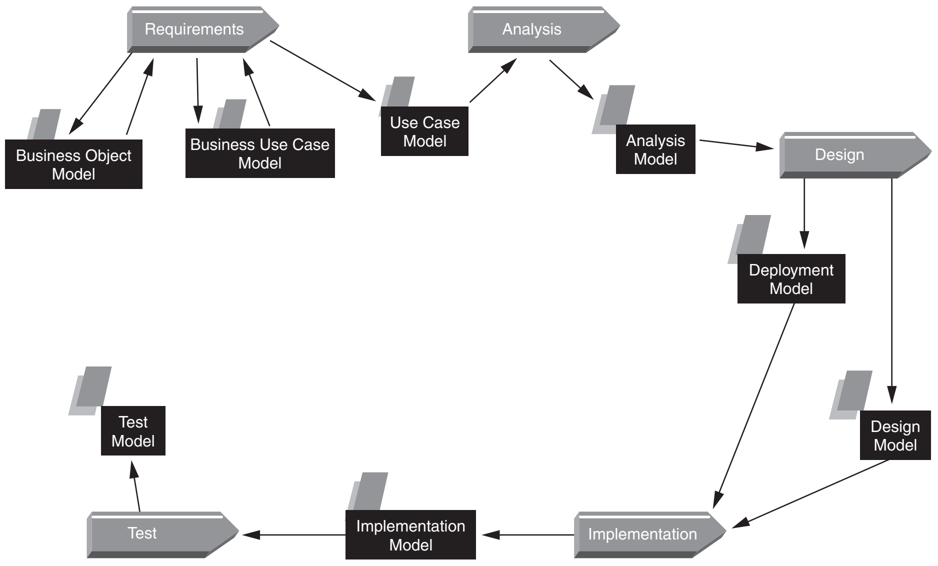


FIGURE 12.22
Up models and workflows.

- All models are relatively closely coupled to individual activities. This is particularly obvious in the test model.

The models in the T&M approach

Figure 12.23 shows that the T&M approach produces a somewhat different picture. In the model of the application domain, the concept model appears explicitly with the description of the actual situation. In addition, the model of the application system is split into application-oriented and technical models. The test model is linked to almost all activities, in compliance with the quality assurance concepts discussed in Section 12.3.

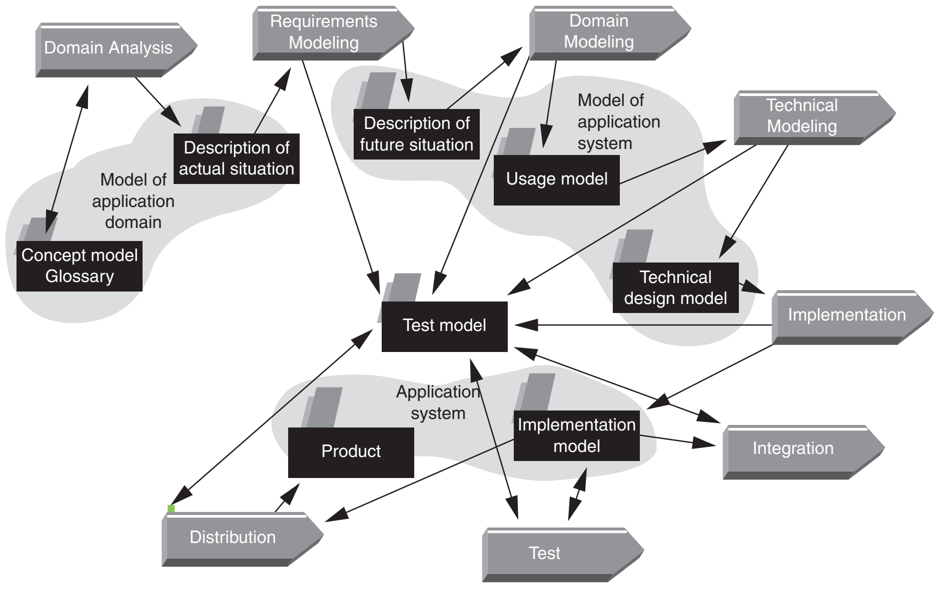


FIGURE 12.23
T&M models and workflows.

Discussion

The fact that the domain models of the current situation are given so little emphasis in UP can be interpreted to mean that these models are either not important enough or are elaborated by people who are outside the actual development project. We consider both interpretations to be dangerous: As we have stated, we find it important that developers acquire a detailed understanding of an application domain. This requires explicit models. However, these models also have to be elaborated in conjunction with the users. This is the only way to encourage the learning process in the author-critic cycles.

We have already explained the importance of application-oriented models. Another thing that distinguishes the T&M approach is the notion of constructive quality assurance. This is along the same line as what the authors of UP say in their books. However, in our view it also includes the intertwining of programming and testing. Section 12.4.2 describes different test types, which are closely linked to the design and construction activities.

XP and T&M

We have said that the techniques and representation means of XP are compatible with the T&M approach. Both approaches are characterized by strong user orientation, short development cycles, and quick releases. XP uses story, engineering, and task cards, managing simpler document types than T&M. However, if you look at T&M and XP as repertoires of methods rather than process models, then the cards and techniques (e.g., planning games) of XP represent a valuable enhancement to the T&M repertoire.

The differences between XP and T&M are found on the level of development objects. T&M distinguishes between versions and, consequently, between core system, special-purpose systems, and extension levels. Moreover, it allows for the construction of prototypes. XP is always oriented towards executable releases. At the most, it distinguishes between internal and external release, which is not something that can be seen to be of qualitative nature.

Summary

Without claiming completeness, Table 12.2 lists the three approaches discussed here, comparing the key planning units and development objects. One could argue that this comparison refers to different orders of magnitude. This is true, since UP is the most extensive and also most heavyweight approach, which means that UP can also be used to plan and implement very large projects or project families involving complex documentation and development requirements.

XP concentrates on smaller or lightweight projects. The development objects are also smaller, which means that larger projects first have to be partitioned. As an example of this scope, the task cards sometimes relate to very small tasks that can be completed in a few hours.

T&M (without XP integration) positions itself between the other two approaches. It is suitable for medium-sized projects and recommends the partitioning of larger projects. On the other hand, the scale of planning of base lines is usually more extensive than that of task cards in XP.

Unified Process	Extreme Programming	Tools & Materials
Cycle Cycles make up the life cycle of a system. A cycle results in a new (external) release of a system.	**Project** A system is developed in projects of less than 20 people.	**Evolution Cycle** A software system is developed in evolution cycles. Each cycle is organized as a project. An evolution cycle results in a new version of a system.
Phase Each cycle consists of four phases: inception, elaboration, construction, transition.	**Release Cycle** Within a planning game, a set of stories are written and selected by the customer that describe what the next release of a system should do. A story should be estimated by the programmers between one and a few days of team programming effort. A release cycle should take a max. of 6 months.	**Project Stage** A software project is subdivided into project stages. These are relevant external events of a project. A project stage has an explicit goal and produces a system version or a prototype. It lasts between 6 weeks and 3 months.
Iteration Each phase is subdivided into iterations. An iteration leads to an increment.	**Iteration** Within the scope of a release planning the developers use stories to write task cards in order to subdivide a release cycle into iterations. Each iteration takes a max. of 4 weeks.	**Base line** Base lines are used for detailed planning of a project stage and for constructive quality assurance. They define tasks and checkable results. They can be completed within a few person days.
Workflow A collaboration between workers (developers) who are using and developing artefacts. Core workflows are abstract descriptions. UP uses requirements, analysis, design, implementation, test.	**Task** Stories are realized by tasks. A task is what implements a part of one or more stories. The scope of a task is between several hours and two days.	**Task** A task has a defined goal and comprises a set of related activities involving documents or system parts. A task is the smallest unit of planning and should be finished within a few hours.
(External) Release A (product) release is a relatively complete set of models and documents (including a build) delivered to an external user.	**Release** A version of a software system that makes sense to the customer by containing the most valuable business requirements.	**Version** A system is shipped as versions. It is usually partitioned into a kernel system with extension levels plus special-purpose systems.
Increment A small and manageable part of a system. Each iteration results in a build which adds an increment to a system.	(Internal Release) Every couple of hours the new code is integrated with the latest release and all tests are run.	
Build An executable version of a system, usually for a specific part.	**Build** An executable version of a system.	

TABLE 12.2 Comparing UP, XP, and T&M.

12.10 REFERENCES

N. E. Andersen, F. Kensing, J. Lundin, L. Mathiassen, A. Munk-Madsen, M. Rasbech, P. Sørgaard: *Professional Systems Development*. New York, London: Prentice-Hall, 1990.

We have taken the general dimensions of a software project from this book.

K. Beck: *Extreme Programming Explained*. Reading, Mass.: Addison-Wesley, 2000.

The standard work by one of the outstanding promoters of extreme programming.

K. Beck: *Test Driven Development*. Reading, Mass.: Addison-Wesley, 2002.

This book explains the test first concept of eXtreme programming. For JUnit see http://www.junit.org.

K. Beck, M. Fowler: *Planning Extreme Programming*. Reading, Mass.: Addison-Wesley, 2000.

The book that is most frequently quoted for planning with XP.

R. V. Binder: *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Reading, Mass.: Addison-Wesley, 1999.

A seminal work on OO testing.

G. Booch: *Object Solutions: Managing the Object-Oriented Project*. Reading, Mass.: Addison-Wesley, 1995.

More on the topic of project management in OO projects.

B. Boehm: *Software Engineering*. IEEE Transactions on Computers, Vol. 25, 1976, pp. 1226–1241.

This paper made the traditional waterfall model popular.

B. Boehm: *The Spiral Model of Software Development and Enhancement*. Computer, Vol. 21, No. 5., Mai 1988, pp. 61–72.

The seminal paper on the spiral model.

R. Budde, K. Kautz, K. Kuhlenkamp, H. Züllighoven: *Prototyping*. Berlin, Heidelberg: Springer-Verlag, 1992.

One of our contributions to prototyping discussing traditional development strategies.

A. Cockburn: *Agile Software Development*. Reading, Mass.: Addison-Wesley, 2001.

A relevant contribution to agile methods.

T. DeMarco, T. Lister: *Peopleware: Productive Projects and Teams*. New York, N.Y.: Dorset House Publ, 1987.

A standard reference for the supporting management style.

C. Floyd: A *Systematic Look at Prototyping*. In: Approaches to Prototyping. R. Budde, K. Kuhlenkamp, L. Mathiassen, H. Züllighoven (Hrsg.). Berlin, Heidelberg: Springer-Verlag, 1984. pp. 1–18.

A seminal paper which has coined the terminology and concepts of prototyping

M. Fowler: *Refactoring–Improving the Design of Existing* Code. Reading, Mass.: Addison-Wesley, 1999.

The seminal book on refactoring in eXtreme Programming.

A. Goldberg, K. S. Rubin: *Succeeding with Objects: Decision Frameworks for Project Management*. Reading, Mass.: Addison-Wesley, 1995.

More on the topic of project management in oo projects.

D. Harel: *Statecharts: A visual formalism for computer systems*. Science of Computer Programming, 8/3, 1987, pp. 231–274.

The seminal paper on Statecharts.

J. Highsmith: *Agile Software Development Ecosystems*. Reading, Mass.: Addison-Wesley, 2002.

A relevant contribution to agile methods.

I. Jacobson: *Object-Oriented Software Engineering–A Use Case Driven Approach*. Reading, Mass.: Addison-Wesley, 1992.

The book in which Jacobson originally introduced his use case concept.

I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*. Reading, Mass.: Addison-Wesley, 1999.

Currently the standard work on procedures in projects in the context of UML.

M. M. Lehman: *Programs, Life Cycles, and Laws of Software Evolution*. Proc. of IEEE, Vol 68, No. 9, Sept. 1980, pp. 1060–1076.

Lehman explains in this paper this different types of software with respect to formalization.

T. Mackinnon, S. Freeman, P.: *Endo-Testing: Unit-Testing with Mock Objects*. In: G. Succi, M. Marchesi (eds.): Extreme Programming Examined, Reading, Masss: Addison-Wesley, 2001.

This paper describes the concept of mock objects for class testing.

D. L. Parnas, P. A. Clements: *A Rational Design-Process. How and Why to Fake It*. In: M. Nivat, H. Ehrig, C. Floyd, J., Thatcher (eds.): Formal Methods and Software Development, Proceedings of TAPSOFT'85, Berlin, Heidelberg: Springer, März, 1985.

An important critical discussion of traditional development strategies.

G. Pomberger, G. Blaschek: *Object-Orientation and Prototyping in Software Engineering*. New York, London: Prentice-Hall, 1996.

A relevant contribution to prototyping with a detailed discussion of development strategies.

Robert V. Binder: *Testing Object-Oriented Systems: Models, Patterns, and Tools*. The Addison-Wesley Object Technology Series, 1999.

G. Succi, M. Marchesi: *Extreme Programming Examined*. Reading, Mass.: Addison-Wesley, 2001.

Based on contributions submitted at the first XP conference (Italy, June 2000): This book offers an overview of conceptional and practical discussions on the topic of eXtreme Programming.

13

T&M Document Types

This chapter describes the document types used in the T&M approach. Document types, especially the application-oriented ones, are an essential element of our approach. They are the material basis for making author-critic cycles really work. Although little is new about the document types we propose, their interplay and their consequent use for actively involving all relevant parties concerned in application development is a trademark of the T&M approach.

The document types presented here have evolved from our project experience as being relevant for dealing with central issues in the development process. The fundamental ideas relating to documentation are described in Section 5.3.7. We first focus on the application-oriented development documents, that is, documents particularly suitable for cooperative work with users, including:

- Scenarios (Section 13.1)
- Concept model (Section 13.3)
- Glossaries (Section 13.4)
- System visions (Section 13.5)
- Prototypes (Section 13.6)
- Cooperation pictures (Section 13.7)
- Purpose tables (Section 13.8)

We describe how these document types are structured and how they can best be used in an evolutionary development process. As an important technique for the interaction with users and domain experts we describe qualitative interviews in Section 13.2. Where possible or meaningful, we provide a reference to the UML diagrams and models. We also describe our related experience in projects. The last section of this chapter (13.9) discusses technical UML diagrams and models.

13.1 SCENARIOS

A *scenario* is a prosaic text written in the users' language that describes the actual situation. The purpose of this description is to identify how and why tasks are performed in an application domain.

Scenarios always deal with a situation as it exists *before* the application system under development is introduced. Existing applications are included in a scenario, but not assumptions about how the new application system could change the actual situation.

We differentiate between scenarios according to the extent of their details; such as, *overview scenarios*, *task scenarios*, and *activity scenarios* will be described later.

THE PURPOSE

Developers must understand the tasks and problems that users face at different workplaces in order to produce an appropriate application system that users can comprehend. Developers must therefore analyze and model day-to-day work situations with their tasks and underlying concepts. The result is what we call a *domain model* (see Section 6.4). We describe the task and process-oriented aspects of this model in a document type called a *scenario*.

Scenarios are a part of the domain model

Scenarios are not only useful to describe tasks, work situations, and processes so they can be understood equally by both developers and users. They also embody the sense and purpose of different workflows, actions, and activities. They serve as an excellent means of communication between the participating groups, and help developers to familiarize themselves with an application domain from the users' view. These benefits make scenarios useful documents in the analysis and development process.

Scenarios can be written on the basis of different sets of goals and with different levels of detail. Naturally, scenarios for task analysis and domain design are primarily elaborated at the domain level and ignore the technical implementation.

Scenarios are generally written by members of the development team and evaluated by the users in author-critic cycles (see Section 5.3.4). This produces two effects:

Who are the authors and who the critics?

1. It forces developers to tell a meaningful story that is comprehensible to users in the language of the users. This assumes that the developers have understood the work situation being described.
2. The scenarios give users an indication of how well developers have understood their tasks. There is also a good chance that the scenarios will make users aware of tasks they forgot to mention in the interviews (see Section 13.2), because they are so commonplace. Thus scenarios also help reveal "blind spots" in the users' own understanding of their work as well as in how developers comprehend what they do.

THE STRUCTURE

Scenarios are written in prosaic form in the application domain language. To the extent that this can be done easily with the available technical means, *terms* explained in a glossary are *highlighted*. Each scenario also has a descriptive *title*. It is often useful to include references to *related development documents*, such as other scenarios, in each scenario.

Since each project normally writes several scenarios, it makes sense to generate a table of contents or, even better, an *overview scenario*.

THE CONTENT

Scenarios describe the aspects of the dynamics in an application domain, or the everyday work situations, where people use tools and manipulate objects as part of their work. This emphasis relates not only to individual actions (in the sense of workflows), but also to the objects used in the process. Written in the form of a short prosaic text in the language of the application domain, a scenario describes:

What does a scenario describe?

- everyday work situations;
- the flow of actions and activities;
- their meaning and purpose; and
- the objects and means used.

Task-orientation

The description is primarily oriented to the *tasks* that a person handles. Descriptions of machine processes are included in scenarios for embedded application systems and their components.

Tasks are performed differently from case to case. It is, therefore, important that the description includes the goal and content of the activities carried out in the tasks being handled.

EXAMPLE: BUYING A NEW DEVICE

The EMS example

Returning to our EMS example, let's assume that the device manager needs to buy a new device. A scenario might read like this:

> To procure a piece of equipment, the device manager obtains offers from different hardware suppliers to ensure that procurements are made on a cost-effective basis. Offers from three hardware suppliers are needed for a device costing $5,000 or less.
>
> The device manager obtains the first two bids by phone. He provides the person representing the hardware supplier with details of what is required in the device and has the offer sent by fax. For the third offer, the device manager uses the World Wide Web. He visits the website of a hardware supplier, prints the appropriate pages, and marks the device of interest on the printout. When the device manager has all three offers in written form, he puts a mark next to the best of the three offers. He writes a justification comment for the procurement of the device and places it in a clear plastic folder, together with the three offers. This plastic folder is then filed in the device manager's procurement file in a cabinet in his office. The procurement file contains all procurements in chronological order. The device manager will use the procurement file to determine, for example, when to sort out a device (see Activity Scenario: Sorting out a Device 13.12). When the offers are filed, the device manager uses an order form to place the order for the device. First he makes two copies of the order form. He keeps one copy in the plastic folder and sends the second copy to the person who will be receiving the device. This second copy is distributed so the other person can follow up on the active process.
>
> When the device is delivered, the device manager places the delivery note into the plastic holder with the offers, procurement justification comment, and order form. The manager uses the delivery note to generate a description of the device. Finally, he adds a unique identification for the device to the room plan.

COMMENT

This scenario identifies the acting persons and what they do to accomplish a specific task. It shows the objects and means used and the way they are handled. Finally, the scenario describes the purpose of the various activities, and it references another scenario where a part of the task at hand is detailed (see Section 13.1.2).

Basic Elements

Scenarios help developers to familiarize themselves with a domain work context, allowing them to better understand work situations, ask questions about specific work situations, and clarify concepts. However, scenarios should not be written exclusively by and for individual developers, since this would greatly reduce the desired communication and learning process for the entire development team.

Members of the development team usually conduct interviews with future users (see Section 13.2) to prepare a basis for scenarios and application-related glossary entries (see Section 13.4).

Obtaining domain knowledge

In addition to interviews, developers can use other techniques to obtain the necessary knowledge and experience for scenarios. The following techniques have proven useful in several of our real-world projects:

- *User observation*: This gives developers an opportunity to "keep an eye" on users. They observe users as they carry out their daily work over a short period and may help them when possible. Though this method takes more time than interviews, it gives developers a more intensive exposure to the application domain.
- *Active participation*: The more passive user observation method can be replaced by active participation, which is like a short internship. Here, individual developers are entrusted with selected tasks from the application domain for an average length of time (several weeks). This often requires some training, after which they work actively with users in their normal work activities. This even more time-consuming familiarization with the application domain makes sense if a domain is technically very complex and if a high level of comprehension of concepts and tasks is necessary for modeling and a constructive cooperation with users.
- *Ethnographic video-supported studies*: When the users in a domain are difficult to identify or are not readily accessible, such as people working in an airport or department store, ethnographic studies are sometimes the only option available to build a modeling basis. Videos are mainly used to document and then analyze certain tasks or processes. They also identify recurring or conspicuous behavior patterns. The use of video technology requires skilled specialist knowledge in the developer team because there is a chance that relevant information may not be captured.

13.1.1 Using Scenarios in the Development Process

The scenario is the primary document type in the communication between developers and users at the project's start. It supports the analysis and the modeling of an existing task area.

Scenarios are also important for the entire development process because they help to relate technical design decisions to relevant issues of the application domain. They also support this task if parts of the application system are to be developed later. It has therefore proven very useful to consult scenarios even with improvements that appear to be purely technical, because many technical design decisions are only clear in the light of a domain context. M. M. Lehman established that every five to eight lines of program code contain a nondocumented assumption about the domain context of a program.

Because a scenario (agreed upon by users) describes an important situation in the application domain, it is filed and not constantly updated. If the situation that a scenario describes changes, then a new version of the scenario should be written. A new scenario, discussed and agreed upon with the users, should also be written if the level of knowledge about a application domain has changed.

NUMBER AND SCOPE OF SCENARIOS

Whether an application situation has been described adequately cannot be derived from the number and extent of scenarios. Consequently, we cannot provide any rules of thumb about how many task scenarios should be written or how detailed activity scenarios should be. We do, however, warn against the misperception that scenarios have to be written *before* all other documents. This perpetuates the illusion that we can assess everything we need to know about an application domain at the start of a project. It is important to remember that we are often unaware of aspects of an application domain we should be looking at more closely until we are working on a project.

Scenarios are not confined to the start of a project

Therefore, a rough overview of the relevant tasks in the application domain should initially be worked out and then described in an overview scenario. Visualization in a business use case diagram (see Section 13.1.3) is also helpful. The initial set of task scenarios are written on this basis. More detailed scenarios are not written or additional task areas identified until there is a good idea about the tasks that can be supported by an application system and how extensive this support should be. Therefore, scenarios should accompany the entire development process.

Identifying and describing tasks in the application domain is a creative process that can only roughly be supported by guidelines. Again, we want to emphasize that scenarios (and other document types) represent a current understanding of a project that develops dynamically. With the different subtypes of scenarios we are presenting a way to record relevant situations in a document type. In this respect, we find all document types to be a “materialized” understanding of situations in the development process.

HOW SCENARIOS RELATE TO OTHER DOCUMENT TYPES

Expectations towards future work and weaknesses in current situations are frequently mentioned in interviews. They must be identified and carefully examined to find the document type in which they are established. Requests and ideas regarding future work methods belong to the document *identified requirements*. Recognized weaknesses in work processes or objects used in these processes belong in *scenarios*.

Identified requirements

The *identified requirements* document is a document type that more appropriately belongs with the project documents, comparable to a project order or offer. The reason is that the perceptions, system characteristics, and requirements described in this document type are not yet part of the agreed upon characteristics of the application system.

Glossary

The terms used in scenarios are written in a *glossary*. This ensures that there is a uniformity in the terms used in the different scenarios. A glossary also makes clear that terms are used differently in different scenarios.

Scenarios are the basis on which *system visions* are created. Since scenarios are oriented to user tasks, they can be used for the “rehearsal” of future tasks through the use of newly designed tools and materials.

Screenplays and prototypes

Scenarios help to produce screenplays which contain a description of work situations that prepare the evaluation of *prototypes*.

The conceptual model, the class design, and the presentation of prototypes in work groups create additional stimuli to refine a domain model. It often turns out that the existing view of an application is still not consistent. Working on system visions and technical designs will disclose uncertainties about the application domain, which should be consolidated. It is important that revised scenarios are agreed upon with the users and that old versions are kept.

System tests

Scenarios are becoming increasingly important for writing *system tests*. The reason is that scenarios contain a description of the application situations to be supported by the new system. Last but not least, scenarios can be used in the application domain as job descriptions for new staff members.

The frequent cross-references to other document types highlight the central importance of scenarios in the development process.

13.1.2 Subtypes of Scenarios

When describing complex work situations and large areas of responsibility, it is often useful (but not mandatory) to divide scenarios according to the granularity or detail of the contexts they describe. We present a classification that has evolved through our experience on projects. We have tried to generalize specific requirements so that they can be applied to new projects. The following subtypes show possible differentiations that make sense when a domain-specific assessment of a current project cannot be described at a uniform level of abstraction:

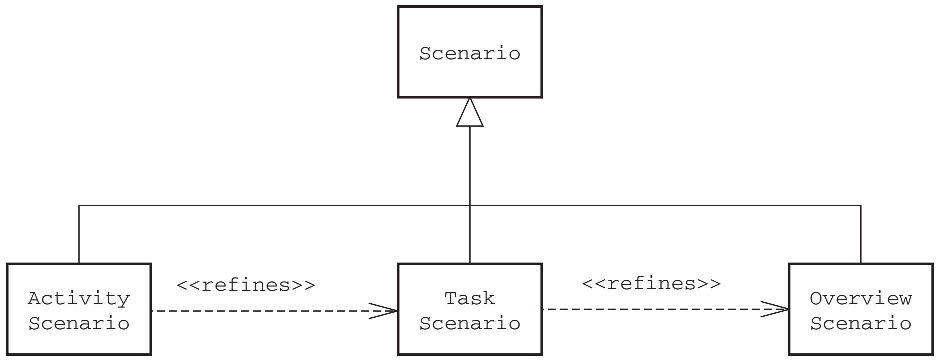
- *Overview scenarios* describe the overall picture of a workplace or an application domain and summarize the important tasks.
- *Task scenarios* are established at the level of individual tasks and related activities.
- *Activity scenarios* consider individual actions, activities, and processes.

Figure 13.1 shows the connection between these scenario subtypes.

OVERVIEW SCENARIOS

An *overview scenario* provides a survey of the entire work situation under discussion. It should give a picture of which people (in the sense of actors or roles) are working together to accomplish which tasks.

FIGURE 13.1
Using a scenario with subtypes.



The aspects considered include:

- Type of participating workplaces
- Participating actors (or workers in UP terms)
- Complexity (and frequency) of tasks
- Sense and purpose of the tasks

Focus on tasks

The emphasis is on the *tasks* listed. The objective of an overview scenario is to work out clearly which tasks are involved. The details in overview scenarios relate at most to the current quantity structure of tasks. The concentration is on *what* and *why* tasks are to be completed by different people in an application domain. What makes overview scenarios special is that they provide the reader with an impression of the tasks being handled by the people involved. It is usually very helpful to enhance an overview scenario with a business use case diagram (see Section 13.1.3).

In overview scenarios the named tasks are allocated to actors (or workers). References to detailed scenarios are also useful. In line with the terminology of the Unified Process, we assert the following:

An *actor* is a collection of tasks and responsibilities assigned to a prototypical workplace and a role. This role can then be "filled" by one or more people. This assignment can change over time, it can be on a long-term basis, and it can be informal or well-established within the corporate hierarchy. The important point is that a role is named and specified. Actors are often linked to a certain position or responsibility in a company.

The Unified Process

The Unified Process distinguishes between roles in the development process and roles in the application domain. In an application domain, users who interact with a use case are referred to as *actors* in the sense that we use the term here. But UP also calls machines that send events to a system actors. In the UP development process roles are assigned to so-called workers. Thus, in many projects the developer role is filled by people who are employed as "workers in a team," which is a worker type.

TASK SCENARIOS

The scenario subtype *task scenario* describes what constitutes an individual task and how this task is completed. This description has the character of a script or narration of a small scene.

Alternatives to different decision points of the action are recorded but kept to a minimum. A decision needs to be made weighing whether several alternatives to similar actions for completing a task should be described ("the customer provides a name and address or a name and account number"), or whether two basically different approaches to a task should be written to two different scenarios ("a new customer sets up an account and a standing order" and "a regular customer sets up another standing order"). The guideline that can be followed is that simple case selections can still be understood when read, whereas involved multiple-case selections or case selections that also relate to previous selections tend to be incomprehensible.

The detail of a task scenario is such that individual actions recurring in different tasks may be named, but they are not described in detail are and instead included in separate activity scenarios. Specific actions important to understand how a task is handled should only be described in special cases. It should be noted that this does not result in an instruction on the specific handling of a certain task.

ACTIVITY SCENARIOS

***Activity scenarios* describe the steps involved in completing a task in detail.**

These are often activities, such as filling out certain forms and operating a specific inspection that reoccur in the context of different tasks. A typical activity scenario could describe certain details to be taken into account when a form is filled out or a certain control procedure or calculation has to be executed several times in different places. In this sense, activities are concrete physical movements, mental processes, or mechanical actions that are oriented to work objects or means. They are the most detailed form of dividing work in an application domain. From the description of activities, we can determine exactly how something is done and what means are used.

The EMS example

THE EMS EXAMPLE: ACTIVITY SCENARIO—SORTING OUT A DEVICE

As in the previous chapters, we use the EMS example to better understand the above discussion.

To withdraw a device from service, the device manager has to update different files:

- The procurement file contains all procurements listed in chronological order. The device manager finds the clear plastic folder for the device to be sorted out, and identifies the device and purchase date from the specification sheet. The manager removes the plastic folder from the procurement file and places it in the disposed devices file.
- The disposed devices file contains all transparent folders for removed devices sorted by removal date. The specification sheet for the respective device with its history is kept on top in the folder.
- The device manager deletes the device from the room plan.

The device itself is collected by a waste disposal firm.

COMMENT

This activity scenario describes the sorting out of a device accurately enough so that an outsider can understand the actions involved. Once they have been described, these details need no longer be considered in the task scenarios concerned.

13.1.3 Scenarios and UML

Business use cases

Scenarios in the sense of the T&M approach correspond to business use cases described in the Unified Process. Business use cases are a form of use cases describing a company's current business. It is important to know the workers and business entities involved in the production of a useful result (called a *work unit*) for the customer (the *end user*). Unfortunately, business use cases have been described only briefly in the Unified Process. However, they play an important role as they make the context of a system comprehensible. The analogy between the task-related way of looking at things in the T&M approach and the customer-oriented view of the Unified Process is obvious. Both approaches describe actors or workers and the objects they use to produce a work result. In fact, business use cases are similar to task scenarios.

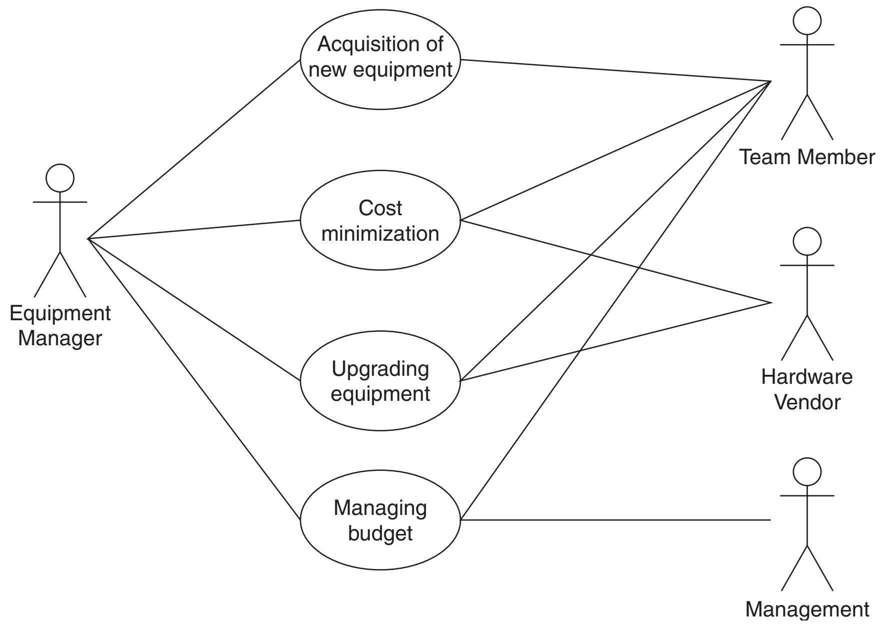


FIGURE 13.2

A business use case diagram in the EMS example.

BUSINESS USE CASE DIAGRAMS

Motivated by the Unified Process and UML, we have used business use case diagrams to add a clear graphic representation of the identified actors and tasks in an application domain to our textual overview scenarios. We use the standard UML notation for use case diagrams. This simple form has proven useful for listing all subtasks named in an overview scenario and the related task scenarios in the business use case diagrams.

Figure 13.2 shows how we can use a business use case diagram in our EMS example.

13.2 INTERVIEWS

Qualitative interviews are an important method used by developers to familiarize themselves with an application domain. They provide a sound basis for document types, such as scenarios and glossaries, and for the domain design. This section presents elements from the interview methods that we have used in real-world projects.

THE PURPOSE

Qualitative interviews

Interviews are part of the learning process in application analysis. *Qualitative interviews* are used to analyze an application domain. They are conducted by an interview team with relevant representatives from the user community. Interviews are conducted to provide a domain representation and interpretation of work situations and contents, rather than representing a quantitative collection of facts and data.

THE STRUCTURE

An interview consists of the following optional parts that will be described in more detail in subsequent sections:

- Presentation of the interview team and the goals of the interview.
- Role playing in a typical work situation.
- Open conversation with those being interviewed.
- Targeted questions asked by the interview team.

General interview principles

The following general principles apply to interviews:

- Interview partners should be guaranteed confidentiality and anonymity.
- The interviewer uses the language of the person being interviewed, that is, the interview is conducted in the language of the application domain.
- The interview should take place at the workplace to ensure that the interview partner is in a familiar atmosphere.

Characteristics of interviews

Interviews have the following general characteristics:

- They are verbal, personal, and nonstandardized. Standardized questions and fixed questioning sequences should be avoided.
- They are open in terms of the questions asked; questions are formulated in such a way that they can be answered with whole sentences and not only with a "yes" or a "no." The course the questions take is also open. The interview partner is the one who structures the interview in terms of content and decides how it is run.
- They are neutral and friendly in style, and should not be conducted in the sense of an interrogation. The interviewer should be sympathetic to the interview partner (and not to the responses).
- They take place as an individual interview to avoid outside influence.

13.2.1 The Interview Process

PREPARING INTERVIEWS

A main interview guideline is established when the content of the interview is prepared.

The *interview guideline* covers all areas that the interviewers find important and helps to direct the questioning so that unclear points are clarified. The guideline is not a questionnaire with standardized questions. It outlines all topics that the interview team wants to address.

During the preparation of a main interview guideline, the interview team often becomes aware of a lack of certain background knowledge. The team must then decide whether these gaps in knowledge can be closed in the interview or whether the team should familiarize itself with the domain content before the interview.

Roles of the interviewers

As part of the preparation for an interview, the interviewers also have to mutually agree on their roles:

- *Interviewer*: Conducts the interview and deals intensively with the interview partners. The interviewer concentrates exclusively on the discussion and does not take notes. It is important that the interviewer "actively" listens so that he or she always follows and motivates the interview partner to provide assessments and statements about his or her tasks. This role demands a great deal of concentration, and, in our experience, the interviewer frequently changes roles with the moderator.

- *Moderator*: Presents the interview team, keeps the interview guideline in focus, and carefully directs the course of the interview. Experienced moderators also manage to take rough notes at the same time. However, caution is needed here. A person often cannot concentrate both on taking notes and moderating interviews at the same time.
- *Note-taker*: Takes notes and at most asks questions regarding clarification; can use a tape or video recorder for support. It is important that an electronic recording is never a substitute for active note-taking, because getting the idea of what is said in the notes is what matters.
- *Observers*: This is a good role during the training phase when new members are integrated into the development team. The observer follows the course of the interview and afterwards comments on how well the interviewing technique was mastered and which improvements would be useful.

Questions to be clarified

There are other questions that typically need to be clarified in preparation for an interview:

- When and how are the roles in an interview changed?
- What is expected at the workplace?
- Who in an organization has to be asked for permission and informed of an interview beforehand?
- What kind of knowledge is available? Is it sufficient, or must additional knowledge be acquired?
- Which preliminary discussions are necessary?
- Is it necessary to conduct a test interview for the interview guideline?
- Which material needs to be prepared? Background material about individual projects, business cards, tape recorder?

Introducing the Interview Team and Approach

For the actual interview, it is important that the interview partner is informed about the meaning, purpose, and objective of the interview. This is the only way to ensure an open discussion atmosphere. In addition, the interview partner and his or her work environment should be informed about the approach to be taken in the project. Employees in the operating departments of companies have often had some bad experiences with software projects. It must be made clear to those involved that their active participation is an important prerequisite for the success of an application-oriented project. They need to understand that they are able to play a role in the creation of the future system and that this role already starts with the interviews.

It has therefore proven useful to make a short round of presentations before the actual interview takes place. Preliminary discussions conducted separately with all participants provide a better opportunity for important information to be conveyed and existing questions and objections to be discussed at the same time. The actual interview will then take place in a more favorable atmosphere.

An introduction takes place before the start of the interview, even if a preliminary round of discussions has already taken place. At a minimum, the interview team introduces itself with its roles, a short explanation is given again of the context, and business cards are distributed. The interview team should not forget to agree on the time frame of the interview again.

Role-Playing

Role-playing is recommended if the interview partner frequently deals with customers in her work. In this case, the interviewer assumes the role of the customer with the interview partner playing herself. A small everyday situation is selected and recreated as realistically as possible. This works particularly well if the interviewer is prepared to become truly involved in the situation being acted out and plays the role of the customer. Software developers sometimes have a problem with this situation and try to use indirect speech only ("what would you answer if I had asked the following . . . ").

Benefits of role-playing

Role-playing is an excellent addition to an actual interview, especially since users tend to adopt their everyday behavior very quickly. This results in things and actions being clarified that may otherwise not be noticed in a straight discussion. It also provides an opportunity to select situations that occur infrequently but are nevertheless important to achieve an overall understanding of the tasks involved.

Another advantage is that a step-by-step review of the role-playing can be conducted afterwards (in a sense in "slow motion"), and the interview partner has a chance to make comments.

Open Conversation

Open conversation is one of the most important characteristics of each interview. Here, interview partners have an opportunity to clarify their assessments and interpretations.

For an open conversation, it is important that the interview partner feels a realistic obligation to talk. Interview partners should be motivated to discuss their work situations narratively. When interview partners get the hang of talking, they start mentioning many things automatically. One could say that they are describing a movie that is being run before their very eyes.

Promoting the narrative flow

Interviewers mainly promote this narrative flow through short interjections and concentrated listening. They can ask questions, but during this phase, they should never try to control the issues of the conversation or change its direction. At this stage, interview partners should explain *their view* of things. They will be prevented from talking in their own language and presenting their own assessments about their tasks if the interview is too tightly controlled.

The interviewer basically directs the conversation towards the current work situation. On their own initiative, interview partners will express judgements and assessments about weaknesses in their work and desirable changes. This should be carefully noted and documented as part of the *identified requirements*. The moderator and interviewer should, however, make a concerted effort to avoid any discussion about the detailed design of future systems at an early point in the project (i.e., when no prototypes have been evaluated yet). Experience has shown that, although users often have an idea of what is wrong with their work or where their is room for changes at the beginning of a project, they seldom have enough technical experience and knowledge to make recommendations on software technologies for a new system.

Targeted Questions

After an open conversation, it is usually necessary to return to some of the statements that were made. The note-taker reviews his notes and addresses points that he did not completely understand or could not write down quickly enough. The interviewer will ask additional questions if she finds that some of the issues have not been addressed in the discussion, including the clarification of terms for the relevant domain concepts and objects. This is also a good time to ask the interview partner for copies of materials that were mentioned in the discussion.

Interview Assessment

Each interview is assessed by the interview team as soon as possible on the basis of the interviewer's memory and notes. Initially, this means that individual statements are recollected. We recommend writing the minutes of the interview at the beginning of a project or if the team is less experienced. This is not a record of how the interview progressed, but rather a summary of results, grouping statements by topics. Such a summary of results usually shows the areas where further questions—or perhaps even another interview—are required. Moreover, the minutes are very useful for user feedback (see next section).

As soon as the results of an interview have been recollected and agreed upon with the interview partner, scenarios and related glossary entries are written.

User Feedback on Interviews

Interview preparation and assessment cover the "author part" of an author-critic cycle. A "critical" evaluation of the documents produced is a good complementary action. The documents are presented to the interviewed users and discussed thoroughly with them. According to the interview concept, it is suggested that the entire interview team be present for this "circle of critics." The documents are then revised as a result of this analysis and evaluation process. This completes the author-critic cycle.

Guideline for analysis

The goal of the interviews and documents should be two main aspects of the analysis process:

- create *diversity*, and
- introduce *synthesis*.

Diversity is produced when interviews are conducted with different users and the corresponding documents are always evaluated by these same interview partners. This should enable developers to recognize the spectrum and bandwidth of the work forms and processes involved. The developers should make sure that this diversity is encouraged in the interviews and not suppressed by rash cross-references to other interviews.

Synthesis is introduced on the basis of the recognized diversity. The documents produced are not discussed individually with the interview partners; instead, they can be evaluated in a joint work session. The discussion partners are then specifically confronted with the different positions presented. The aim of such a discussion is to identify points that have been agreed and disagreed upon, so that an integrated application system can be created from the recognized diversity.

We have often experienced in such plenary meetings that many of the positions that initially appeared to be controversal could in fact be combined into a joint compromise, and that only a few real differences existed between the users. These differences can be adopted in alternative system solutions and presented to the users for their decision.

Both results are desirable: A compromise between different positions makes it easier to develop a slim system, and early identification of relevant differences in requirements clarifies where more flexibility is needed in the system design to cope with variations.

13.3 THE CONCEPT MODEL

A *concept model* is a taxonomy of terms based on the objects of the application domain. The central concepts of the application domain are related to one another and described in their characteristic interactions in the concept model.

Concept models are represented as a network of services (e.g., in the form of CRC cards as introduced by Beck and Cunningham) or in the sense of a UP domain model (in the form of class diagrams) from the concepts and relationships defined there.

A concept model is always based on the terms and objects identified in the application domain. It represents the concepts that are central to the design of the application system.

THE PURPOSE

While we are developing a common understanding of the "big picture" of the business processes and tasks in our analysis of an application domain, we have to concentrate our efforts very quickly. For we have to consider all important tasks, subprocesses, and objects in view of further system development early on while we are analyzing the application domain. This places us in a dilemma. We have to be familiar with the future system, if we want to decide which parts of the current situation in the application domain are important for the development of the system. At the same time, in order to design the future system, we have to be familiar with the application domain. This is where a concept model comes in handy.

The dilemma of analysis and design

A good concept model is the foundation for the domain architecture. It shows the concepts that we further develop as classes of the software system, and it also guides the ongoing domain analysis and modeling process. Our job is to group scenarios and glossary entries around the central elements of the concept model. This means that we have to concentrate on the topics, tasks, and processes linked to the concepts we selected from the model.

THE STRUCTURE

A concept model is usually developed in two forms: as a set of CRC cards or as the class diagrams of a domain model along the lines of UP.

CRC cards are particularly suitable for the actual modeling process. The relevant concepts are selected from the scenarios and glossary entries (see Section 13.4) and noted with their service relationships. It is important that the concepts and interactions used are completely application-oriented; they shouldn't be seen as programming language elements (e.g., attributes or method names). The CRC cards that constitute the core of the card network are those used for the concept model.

Using class diagrams for domain models

Class diagrams are used in UP to represent domain models. We use class diagrams for our concept models, with the following modifications:

- Concepts are represented as class rectangles with the compartment's name and, in some cases, responsibilities.
- Superconcepts and subconcepts are linked through the inheritance relationship.
- The general form of the use relationship means that two concepts are linked in the form of service provider/client. The client relies on the service provider to obtain a service. We use the dependency relationship (arrow with broken line) for this.
- The *contains* relationship is a special form of the use relationship. It associates a container or collection with the elements kept in it. We represent this through the association relationship (arrow with solid line).

THE CONTENT

We only place terms and notions from the application domain in the concept model when we are fairly certain that they will play a major role for the future system. A concept becomes important if it is central to the usage model of an application, and if it occurs in the software architecture.

We first identify the relevant objects of the application domain and how they are handled. The modeled objects of the application domain are then "coined" for the concept. To abstract common features existing across several concepts, we use *generalization* to build a more general concept, or a superconcept. The structure of superconcepts and subconcepts developed through generalization and specialization forms a concept hierarchy. This is often referred to as an "is-a" relationship in the literature.

Generalization

The mechanism used to build objects from other objects and then collect, manage, and order these objects is called a *composition* or *containment* relationship. The composition relationship between objects is retained between concepts at the modeling level, because it often refers to application domain containers. We have already said that collecting and ordering is one of the fundamental characteristics of human work.

Composition/containment

A *dependency relationship* means that an object can only provide a service with the help of another object. For example, a loan contract can be processed only provided that there is some form of security or collateral (see Figure 13.3).

Dependency

It is often uncertain whether we should use a containment or dependency relationship. In such cases, we use the containment relationship when the object can explicitly accept and release a collection of elements.

Note that these relationship forms have to be expanded, because they do not allow for an elegant modeling of different views and work contexts in an application domain. We use the role concept described in Section 9.4.1 to solve this problem. However, the relationship types discussed here are normally sufficient to model an application domain.

THE EMS EXAMPLE

The EMS example

Figure 13.4 shows a very "early" and simple concept model for our EMS example. Assume that we already identified the relationships between the concepts. This concept model provides the basis for the discussion of numerous decisions. For example, the room plan should only use business cards and data sheets, but not devices or

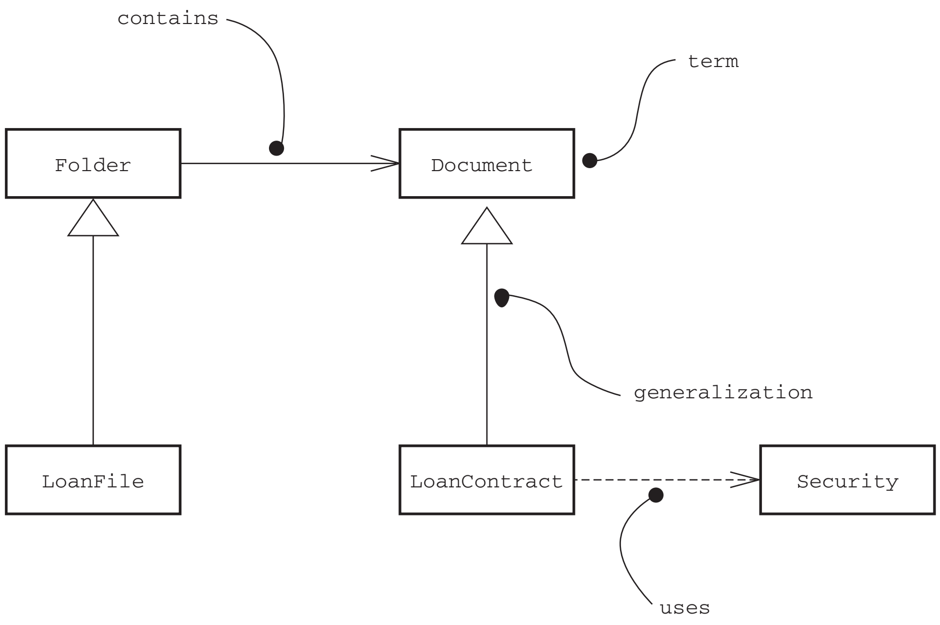


FIGURE 13.3

Generalization, dependency and composition.

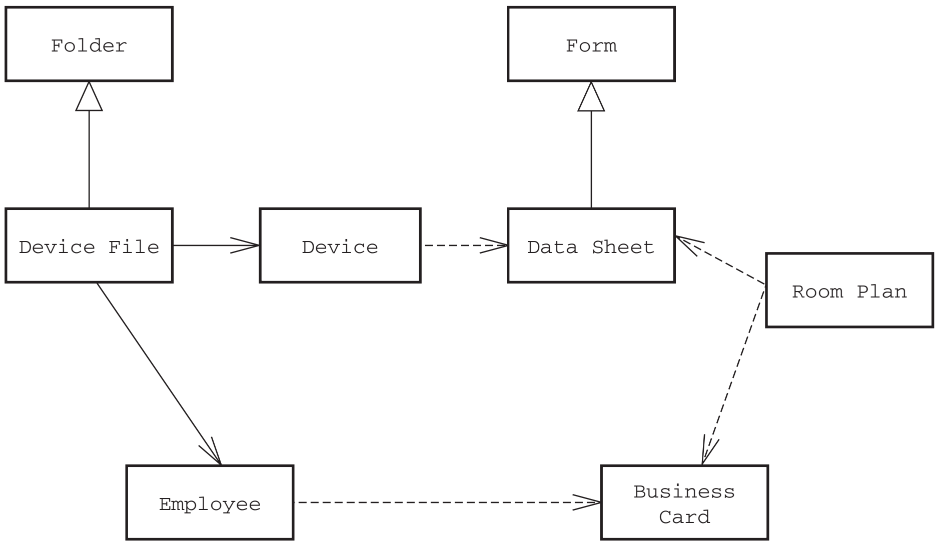


FIGURE 13.4

Using a concept model in the EMS example.

employees. On the other hand, the device file contains employees and devices. The data sheet is designed as a form.

Basic Elements

Interviews, scenarios, and glossary entries obviously form the basis of a concept model, showing the concepts in their context. However, not all concepts modeled in scenarios and glossaries are adopted in the concept model. This takes us back to the same dilemma. Analysis and current state modeling should not be separated from the design of a planned system. Although no use cases or system visions have yet been written, the initial perception of the system under development is an important factor for the concept model. In summary, the concept model should not include concepts that do not have to be supported in the application system.

13.3.1 Using a Concept Model in the Development Process

The concept model is particularly important at the beginning of the development process as it links analysis and design. When we talk about the concept model, we are still balancing the efforts for analysis and modeling of the application domain against the efforts for building the software.

The concept model continues to develop in the course of the project. It should always reflect the domain essence of the logical system architecture. The domain concepts and their relationships are not only important in the application domain; they also have to be represented in software, using classes. Thus regular examination of the concept model can prevent an application system from being developed "away" from the domain concepts.

Finally, a suitable description of the concept model often helps to clarify the basic system design concepts to non-IT experts.

Relation to Other Document Types

As we already mentioned, the concept model is based on interviews, scenarios, and glossaries. CRC cards are another important tool. The concept model directly affects class design and system visions or use cases, since it defines the domain core of these documents.

As described in the previous section, there should always be feedback between the class model and the concept model to ensure that the two do not develop in different directions.

13.3.2 Concept Models and UML

In its description, the concept model corresponds to the domain model of the Unified Process. Jacobson accurately speaks of a *domain object model*. He points out the risk of constructing such a domain object model solely on the basis of the developers' knowledge. Instead, he recommends elaborating a business model systematically.

Our model of the application domain with its different document types and aspects is definitely an interpretation of a business model. We place great importance on a systematic approach towards business modeling. The concept model is an important component of the application system model. In the sense of UML, we can therefore argue that our concept model can be thought of as a class diagram. Compared to the Unified Process, it corresponds to the domain object model.

13.4 GLOSSARIES

A *glossary* is a directory of terms relevant in an application domain. These terms relate to the objects used in the application domain, and to the associated handling characteristics.

A term listed in a glossary represents a *glossary entry*, so that the glossary forms a dictionary of terms used in the application system.

The Purpose

A glossary is the central location for domain and technical terms relevant in an application domain. It is therefore used to build a common project language, facilitating the training of developers in the application domain. In addition, they document where the developer team has agreed on a common understanding of a term.

The Structure

The primary question that arises when glossary entries are created is, "What is the purpose and meaning of the concept or object?" This does not relate to the structure or inner construction, but to the use planned for an object. A form file is a good example. A brief characterization would be the following: "A form file provides the forms needed in a consulting or sales situation as copies sorted by the different banking categories." Information about the content based on the following scheme would not be sufficient: "A form file contains the forms 'Contract form for saving with additional payment' . . . ."

Although there are many suggestions in the literature for the (conceptual) structure of glossary entries, we feel that overly rigid instructions for structure act as a hindrance in working with the glossary. It is important that a glossary entry starts with an explanation of the meaning and purpose of the described object. The following example for the structure of a glossary entry should only serve as a suggestion:

Structure of a glossary entry

- Name
- Meaning, or a brief description of content (what it is, how it works, what it is good for)
- Structure (how it is constructed, what parts it consists of)
- Example
- Context (what connection the term has with other terms and concepts)
- View (who created the description from which "role" and in which project)
- References to other development documents

The naming and structuring of a glossary entry depends on the respective context. Although an informative entry can contain structural characteristics, it does not have to comprise all components.

The Content

A glossary emphasizes the static aspects of the application domain we are to model, but not the dynamic aspects that are described, for example, in scenarios and use cases. Glossaries primarily explain objects and concepts rather than events or processes. Domain and technical terms used in the concept model are defined in the glossary.

The content and description type of a glossary are oriented primarily to the language of the application domain. All uncertain terms used in the application domain should be documented. There is no formal way of specifying which terms are so trivial or self-evident that they do not belong in a glossary. When different operative departments use the same terms but with a different meaning, or when several projects build a joint glossary, it is necessary to clarify the different sources or views of a glossary entry.

Term	Explanation
Offer	An offer shows the price of a device of a piece of equipment with VAT included.
Workplace	Every team member has a workplace of his or her own. It is located in a room identified by a room number. A workplace comprises a workplace computer.
Workplace computer	Synonym for device.
…	

TABLE 13.1
Using a glossary in the EMS example.

This is the only way to ensure that glossary entries can eventually conform closely to a changing domain language, and that design decisions can retrospectively be documented in domain terms.

The EMS Example

The EMS example

Table 13.1 shows a small part of a glossary in the EMS example.

Basic Elements

Glossary entries are usually created in conjunction with scenarios, cooperation pictures, and use cases. It is also important to link a glossary closely to the on-site interviews. Scenarios and the accompanying glossary entries are written from the interviews and the interviewer's recall of what was discussed. This is the only way to ensure that the *application domain language* and not the development team's version, is documented in the glossary.

We explicitly warn against trying to save time by taking domain terms from reference books without first coordinating with the users. It has been shown that even apparently universal definitions of terms (such as "account" or "service") have a "local" interpretation and use that are important for the subsequent design process. Once they have been entered in a glossary, "fuzzy" terms taken from the literature acquire their own momentum that is hard to stop and can lead to serious design errors.

13.4.1 Using a Glossary in the Development Process

Glossaries give future users a chance to think, together with developers, about the significance of the terms they use both in their normal and technical language. Glossary entries are often helpful to clarify the developer's understanding of things, serving as a catalyst for editing other documents.

Glossaries refer directly to the terms used in scenarios, use cases, and system visions. A glossary should reflect the current state and develop a "project language." Ideally, project glossaries and histories can be extracted from an integrated glossary. It therefore makes sense to link design documents with glossary entries.

Common or project-specific glossaries

In some cases, it may be necessary to clarify whether to create an integrated glossary or several project-specific glossaries. From the technical view, a single large glossary is more difficult to maintain than several small ones. From the application domain's view, however, a large glossary means that each project can access the consolidated results of previous projects for their own work. On the other hand, there will be little motivation from the view of an individual project in maintaining the consistency of a large glossary without the necessary technical support.

The optimal solution for glossaries in the development process depends on the availability of appropriate technical support, similarly to a good project library: each project can "check out its excerpt" from a glossary and work independently on this excerpt for an extended period of time. In fixed intervals, the project glossary is reintegrated ("checked in") into the complete glossary. This integration does not mean that the project-specific view is lost. It only means that the glossary has to be checked to see where standardization is sensible and necessary and where project-specific differences should be maintained for a term.

For example, the term "customer information" can mean something totally different for self-services support than for financial consulting. However, it would be useless if both projects had an incompatible interpretation of the term "account number."

Tool support

Great progress has been made through the use of data dictionaries, compared to manual glossary management. However, there is still room for improvement in sorting, searching, and managing cross-references in extensive glossaries. A large glossary is unlikely to be used unless it is extremely user-friendly. As a consequence, we see a second negative effect: When a glossary is only partially updated, it raises the question of which descriptions of terms are still up-to-date and which ones have already become outdated in practice.

We have had positive experiences in projects that use the hypertext capabilities of Web browsers for document management. Separate project websites are also very suitable; these are used as community spaces that provide general information and allow users to exchange documents.

One of the questions frequently asked in this respect is how many glossary entries should be created and what kind of effort to produce them is justified. The same thing we said about scenarios applies here as well. The terms of a domain language can never be defined exhaustively. Also, an attempt to create a glossary at the beginning of a project doesn't make much sense. The reason is that, as time goes by, the terms of the "original" domain language are gradually supplemented by terms evolving from the new application system, which will eventually be part of the application domain. These new terms help to determine where and to what degree changes have occurred in the domain language, that is, where the domain language has been "reconstructed."

This shows that a glossary has to undergo constant development during the development process.

Relation to Other Document Types

Altogether, we can say that the use of a glossary is not limited to the development and evolution of an application system; it also serves as the basis for user manuals.

The glossary is the basis for the concept model. Each term adopted in the concept model should be carefully defined in the glossary.

Coordination problems can occur between the glossary and the designs when, for example, form descriptions or computation instructions are so detailed that they cannot easily be added to the glossary. These descriptions should then be incorporated directly into a design document.

Together with the concept model and the scenarios, glossaries form the basis for system visions, use cases, and the terms of a cooperation picture (see Section 13.7), and, consequently, also for prototype preparation.

13.4.2 Glossaries, UML, and UP

UML does not use glossaries, and they play a subordinate role in UP. However, they are considered very important for the development process. In UP, a glossary is an artifact and corresponds to our definition. It is used with the requirement specification and supports use cases. It also plays an important role in domain modeling. This means that UP defines and uses glossaries in the same way as in our approach.

13.5 SYSTEM VISIONS

A *system vision* describes in short prosaic text form how tools, automatons, and materials can be used to handle domain tasks.

A system vision takes both the domain and the software views into account. When focusing on workflows, system visions and use case descriptions are equivalent.

A system vision provides developers with a sound basis to develop a common understanding of the design of a future application system. This means that system visions document the current understanding that the project team has of the system under development.

THE PURPOSE

In the T&M approach, system visions are important to link analysis and design. They form the logical connection between the domain models and the actual system and its prototypes.

System visions promote the design process in a team. Using system visions raises questions about the objects being designed, their usage model, and the tasks to be supported. In the case of large teams or teams that do not have an adequate common background of experience, answers to these questions should be documented. This is one of the main purposes of system visions.

Variants of system vision

System visions should serve different purposes due to the different kinds of issues involved. Consequently, they also vary in detail and focus.

- They provide an overview of the entire system being developed.
- They present a dynamic sequence of events from a domain or software engineering view. This corresponds to the use case descriptions in UML.
- Accordingly, they model the design metaphors of each system element.
- They represent specifications for system construction, thus serving as targets for the developers. The story cards and engineering cards used in eXtreme Programming can be interpreted as a special form of system visions.

The main idea behind system visions is to link the concepts of the application domain as seamlessly as possible with the concepts of software construction. They respond to questions that arise and document design decisions even before answers can be tested on prototypes or executable system versions.

THE STRUCTURE

A system vision

- has a domain-motivated title;
- is either domain or software-oriented; and
- helps find answers to specific questions.

This formal structure is kept very general and is always substantiated through the problems being worked on.

THE EMS EXAMPLE

The EMS example

In our EMS example, a short system vision might look like this:

Updating Devices

The device manager starts the device organizer at his workplace. In order to compile a list of devices that have to be updated, he does the following:

He clicks the "Find" button of the device organizer. He selects the option "Find all devices updated before . . . months" and enters "6." Then he selects the local equipment folder. After clicking "Compile list," the list of devices to be updated is compiled. The tool displays the number of devices found. After clicking the button "Edit list," the forms editor is started.

13.5.1 Using System Visions in the Development Process

System visions help developers to express their ideas and document them for further discussion. In our experience, only certain visions are meaningful for cooperative work with users. They have to provide an overview of the general purpose and features of the system without technical details. Then they can provide a useful decision-making basis for the management of the development and application organization, for example, by giving an idea of the future task distribution. These visions can be supplemented by use case diagrams.

More technical visions are normally not suitable for discussions between developers and users, because users lack the software engineering background needed to develop a consistent picture of things, such as the behavior of a tool or the execution of an automaton. Prototypes are much better for this.

For large teams or complex systems, system visions are used by the project team as sketches and meeting notes. The note-taker records his or her view of the discussion results in a system vision and is then the author. In the next project session the other members of the development team are the critics, whose job it is to combine the author's description with their own views. This procedure develops the next version of a document.

DISCUSSION

The kinds of visions that we should write, which questions are relevant in a project, and in which sequence questions should be handled are issues that have to be dealt with on a project-specific basis. Not all projects will require visions to cover the open questions. Small manageable applications are often developed only using the resources of eXtreme Programming, that is the story and engineering cards (see Section 12.8.4).

This applies mainly to well-trained teams that work jointly in short feedback cycles on one application system.

A system vision is a tool that helps developers to write a concept of a joint idea, or a vision of a future system. In most projects, the documents themselves are mainly memory aids or notes from the discussion process. This situation changes when a development process has to meet increased demands for quality assurance. This is the case when an external audit, an independent review team, or a formal certification process has to be incorporated. In these cases, system visions become domain specification documents that can be linked to program code and system tests.

Basic Elements

System visions are often written along with the scenarios. This allows us to establish a direct reference between the current and future situation, and gives developers a certain assurance that they have not overlooked anything important. Of course, a concept model and cooperation pictures are important for system visions.

It is important, however, that the domain model is only used as a starting point for system visions. In many cases a direct mapping of processes or even tasks is neither possible nor desirable. Cooperation pictures of future business processes help to decide which part of the future work should be supported by the system under development.

Number and Scope of Scenarios

There is a certain risk inherent in the use of nonexecutable documents for developers. Documentation work should never be approached as an end in itself and with a demand for perfection and completeness.

Decisions for using system visions

The work involved for a vision must always be viewed in relation to gaining knowledge. The team has to make a decision concerning:

- whether system visions are even sensible or whether, for example, it would not be better to build prototypes on the basis of scenarios and concept and class hierarchies with CRC cards; and
- whether a discussed system vision should be revised or whether the team has developed a common understanding of the issues in the meantime.

Project management should make sure that this decision is not made too hastily. Many teams, particularly when the members are inexperienced, would greatly benefit from working with system visions, but often tend to start programming, while neglecting the documentation part. This is clearly not our approach.

Relation to Other Document Types

Visions must be consistent among themselves and with other documents. The developers must check whether the tasks described in the *scenarios* and requiring support by the future system have also been modeled sufficiently in visions. The terminology used in the visions and in the *glossary* also has to be consistent, and lastly, the tasks being supported must be modeled in their processes as well as at the level of the individual components.

We therefore already tend to specify in system visions how a system is to be designed through tools, materials, automatons, and service providers. Here, we also use the domain terms that are then arranged in hierarchies of the concept model and are realized as a class model. We have stipulated this connection with our requirement for structural similarity (see Section 5.2.5).

13.5.2 System Visions and XP

Story cards

Story cards are the key application-oriented document type in eXtreme Programming. They are used in the planning game where actors in the roles of programmers and customers agree on the requirements of a future system. The story cards are written by the customer in "original" XP and then elaborated by the programmers after priorities have been jointly set and a time estimate projected. Here story cards are pure domain descriptions of the characteristics and interaction forms of the system.

We also use story cards for the planning game, but in a different way than in "pure" XP, and our planning game also differs in some aspects. Firstly, we divide the customer role into "customer" and "user." While the customer signs the contract and arranges for payment, the user will actually use the future system. In most of our projects, neither the users nor the customers write story cards, because they normally do not have the time, appropriate skills, or process knowledge necessary. But this depends on type of project and on the customer and users. Our developers usually write story cards based on interviews with users and observations of their actual work situations. These story cards are reviewed by the users and the customer. The users have to assess whether the implementation of the story cards will support them. They therefore review the developers' understanding of the application domain. The customer decides which story cards to implement in the next development iteration and which priority to assign to them. To avoid serious conflicts of interest between the users and the customer, both parties are involved in the planning game. Users can then articulate their interests and discuss the priorities of the story cards with the customer.

Our experiences have shown that the users and the customer will normally compromise on their mutual interests. However, the decision about what will be implemented next will not be made finally by developers but by the customer, regardless of a planning game's outcome.

The limitations of using story cards

A complex project will have a mass of story cards, which makes it difficult for users, customers, and developers to obtain an overall picture from the story cards. For such projects, we use the additional document types described in this chapter.

In addition to application-oriented story cards, we use engineering cards for refactoring in XP. These cards are written by developers for developers and define a development task. These cards also include a time estimate and are ordered by priority. The programming pair that works on a story or an engineering card ticks off a card and writes down what work was actually involved.

13.5.3 System Visions and UML

Use cases

It is obvious that system visions are closely related to use cases. The basic idea of our version of interaction visions is directly comparable to normal use cases.

A closer look shows that use cases, as Jacobson originally proposed them, can be seen as an important addition to our system visions. Use cases are brief descriptions of how an action initiated by an actor is processed by the system, which produces a reaction. This means that each use case describes one potential execution throughout the system. The set of all use cases describes all reactions possible in the system. This set is normally never written in its entirety, but the basic idea is very helpful. It provides a domain and software description of the system behavior. The technical version of use cases is usually represented in a sequence or interaction diagram. This establishes a connection to the technical construction. Moreover, system tests can be based directly on use cases.

Human and technical actors

Use case diagrams represent an important addition to system visions. System visions can be easily drawn as overviews in these diagrams.

At this point another comment about the actor concept would be useful: According to UML/UP, people as well as technical aggregates may be regarded as actors. This is certainly useful when it comes to embedded application systems. We recommend, however, that human actors be represented differently than mechanical ones. Developers must always be clear about whether a person or a machine will be expected at the interface of the system they are developing.

13.6 PROTOTYPES

We will start with a definition:

A *prototype* is an executable model of selected aspects of an application system. Prototypes serve as a discussion and decision-making basis for developers and users. They offer the most important foundation for evaluating future application systems.

However, prototypes are also used to experiment and gain knowledge and experience. This means that they help us clarify specification and construction problems.

THE PURPOSE

Prototypes can be built to support various activities in a software development process, including:

- Project initiation
- Analysis of the application domain
- Design and construction of the application system

Following Christiane Floyd, we distinguish between different *prototyping kinds*, depending on the relationship between a prototype and its purpose:

Prototyping kinds

***Explorative prototyping* is used when problems are not clear. The requirements of users and the management of an application system are clarified. The developers familiarize themselves with the application domain and the tasks handled by the users.**

***Experimental prototyping* is used when the technical implementation of a development goal needs clarification. The users experimentally clarify their ideas of the application system. The developers obtain a good idea about the feasibility and suitability of the system.**

***Evolutionary prototyping* is an ongoing process to adapt an application system rapidly to changing conditions. Software is developed in a continuous, evolutionary process rather than in self-contained projects.**

THE STRUCTURE

We can say little that is specific about the structure of a prototype, besides the guidelines given for general T&M system construction. But we can distinguish between the following prototype categories and the associated questions regarding goals and purposes, and how they are evaluated for development:

Prototype categories

A *presentation prototype* supports formulating a first impression of a future system. Its goal is to show the customer what the application system will basically look like. It should provide developers and users with an initial idea of how the system will be handled and what use context it covers. Presentation prototypes are normally evaluated by the corporate management.

A *functional prototype* helps to clarify problems and answer design questions. It shows parts of the user interface, combined with a section of the functionality. Functional prototypes usually already incorporate the architecture of the applications system and, therefore, also support the construction of the system. These prototypes are evaluated by all groups.

***Breadboards* are pure technical prototypes aimed at clarifying the software engineering issues that arise during the construction of an application system. They use a section of the implementation model for this purpose. This prototype category is also found in traditional development projects. Breadboards are evaluated by developers only.**

A *pilot system* is used and evaluated during the transition stage in the application domain. This is a technically mature prototype. The initial pilot system often corresponds to the core system of the future application. It is enhanced in an evolutionary way through extension levels (see Section 12.7.2). Pilot systems offer comfortable and secure handling and a minimum of user manuals. Users have a major influence on the evaluation of pilot systems.

THE CONTENT

We have organized the issues and contents for the construction of a prototype, based on the prototype category we want to build:

Issues of the prototype categories

Presentation prototypes:
- Which guiding metaphors and design metaphors should the prototype clarify?
- Which parts of the application domain can potentially be supported by the future system?
- Which basic manipulation and presentation should the system offer?

Functional prototypes:
- Which problems or design issues need to be clarified with the users?
- Which tasks and work sequences should be supported and how?
- Can selected tasks and sequences be dealt with easily and in line with business rules and regulations with the prototype?

Breadboards:

- Which construction alternatives are available for a technical problem?
- How do the various implementation possibilities work with the architecture of the application system?
- How does the application system behave in an embedded system basis?

Pilot systems:

- How well does the pilot system prove itself in the handling of everyday work activities?
- Can the extension level planned still be tackled on the basis created?
- How does the application system behave in real-time operation?

The T-Prototype

The T-prototype as a mixed form

In practice, mixed forms of prototypes are frequently developed along with the pure prototype categories listed above. The so-called *T-prototype* has proven effective in many projects. It consists of a combination of a presentation prototype and breadboard. Like a presentation prototype, it is structured broadly so that it clarifies the fundamental handling and presentation of an application system at the interface. As a breadboard, it is structured deeply to demonstrate the technical feasibility. This produces an overall impression of the application system that is then enhanced by individual operative components.

T-prototype construction is particularly good if the prototype is evaluated by different target and interest groups. Often both domain and technical criteria are emphasized, especially when both users and the development management group evaluate the success of a prototype.

13.6.1 Using Prototypes in the Development Process

Prototypes provide the prime basis that enables developers and users to talk substantially about a system. An evaluation of domain-related and technical design decisions is possible on the basis of the different prototypes. The most important prototypes are those that encompass a developed user interface as well as the domain model of the application system and an implementation model. Again, this stresses the importance of our demand for structural similarity (see Section 5.2.5).

Benefits of prototyping

- The use of prototypes shows users that the analysis process with application-oriented document types (scenario, glossary, cooperation pictures, and purpose tables) produces relevant operational results that support their work.
- For the development team it is important that the system visions do not have the character of illusions, that is, they have to show that it is basically possible to construct the solution they are seeking.
- Development management receives clarification that the innovative development process will lead to workable results (pilot systems) at an acceptable cost in the near future, and that these pilot systems can be integrated into the existing application domain and the technical basis.
- Lastly, user management clearly sees that the development process is leading to a solution that can be justified in organizational and operational terms, and that this solution is more efficient than the current one in terms of dealing with the respective tasks.

In addition to prototypes, breadboards in a narrower sense—as prototypes used by the developer team to evaluate design alternatives—are always a valuable enhancement to the development process. They are useful because they help to clarify construction problems for developers and also often serve as a catalyst for new and far-reaching ideas for system design.

The use and evaluation of prototypes in the development process is important for the success of prototyping as part of the development strategy (see Section 12.3.2).

Workshop Visits

In addition to the classic form of prototype evaluation within project reviews and user work groups, we have had great success using workshop visits as a methodical element in different projects.

***Workshop visits* are informal meetings at the developers' workplaces. In their own work environment, the developers present the prototypes of a future application system to its users.**

This kind of prototype demonstration has several strategic advantages over normal meetings, workshops, or reviews. First, the atmosphere is informal, and communication can take on a more personal character. This atmosphere can help clarify many problems in advance, before official project deadlines or reviews. Another advantage of workshop visits is that they contribute to the ongoing cooperative work between developers and users. Partial results can be presented at a workshop visit even before a prototype is completed. No one would expect to see a finished product. Nevertheless, the impression is given that something is happening, one can see where the development is going, and users feel they are actually involved.

In general, feedback cycles allowing users to evaluate prototypes should be within short intervals, because these discussions form the basis for a common project culture in which mutual domain competence is accepted.

Prototype Screenplays

We have also been successful in using prototype screenplays with workshop visits and other forms of prototype evaluation.

A *prototype screenplay* is a short script that represents a brief description of everyday work situations in the application domain. The handling of tasks is described in a way that approximates everyday reality, without explicit reference to the new application system. The scenes from a screenplay are then played out, with a prototype.

The main value of prototype screenplays is that they facilitate a targeted and practical evaluation of prototypes. The emphasis is on relevant tasks, without relying on the concrete usage model.

Prototype screenplays help to avoid the risk that users or developers might make unnecessary or nonsensical things the subject of their evaluation by interactively playing with prototypes. For example, instead of conducting fruitless discussions about the detail of a screen layout, participants can concentrate on the domain-specific core of a prototype and the usability of modeled tools and materials. Incidentally, we should mention that domain-motivated screenplays are very useful in evaluating standard software.

Number and Scope of Prototypes

The kinds of prototypes just discussed are normally built over the entire period of a development project. This means that we use the entire spectrum of prototype categories, depending on the problem involved, especially in large projects. It also means that prototyping is not reduced to one development phase, such as identifying requirements.

The value of prototyping to high-quality software is undisputed. This is also confirmed by the Unified Process demand for an iterative and prototype-oriented approach.

Finally, prototypes are the most effective means for presenting the dynamics of a design and turning it into something that can be discussed by all participating groups. They play a key role when project progress is identified in stages (see Section 12.8.2), and as pilot systems form the flexible transition between a core system and its extension levels. It is therefore not sensible to limit the number of prototypes in general.

Relation to Other Document Types

Prototypes, apart from relating to all document types described in this chapter, are particularly connected to system visions. One reason is that system visions occur at the decisive interface between the analysis of a current situation and the design of a future system. On the other hand, we have pointed out that system visions only have restricted suitability for discussions between all participating groups. We therefore have to coordinate system visions properly with prototypes that can be evaluated on both a domain and a technical basis.

Prototype evaluations typically intensify the author-critic cycles. Scenarios and glossaries are often revised after a prototyping session. On the technical side, functional prototypes and breadboards can influence questions about the embedding of new system parts.

13.6.2 Prototypes and UP

Prototypes do not play an independent role in UML, which is obvious. The Unified Process is a different matter. Prototypes are very important in an iterative approach. UP uses two prototype categories:

UP prototype categories

- *Exploratory prototypes* are supposed to show potential solutions, but are not developed into complete applications (disposable prototypes). Examples include prototypes that implement the user interface or an interesting new algorithm.
- *Evolutionary prototypes* are supposed to be developed in successive stages, such as architectural prototypes.

We conclude that prototyping is firmly integrated in the Unified Process. However, the concepts are not as differentiated as they are in T&M. Moreover, it appears that authors do not yet have as much experience with the approach. Overall, however, we can see that there is a shared fundamental understanding of prototyping.

13.7 COOPERATION PICTURES

A *cooperation picture* is a visual representation of a cooperative work situation. Self-explanatory pictograms and arrows are used in cooperation pictures. Cooperation pictures show how people (as actors) work together on the basis of a division of labor and what they exchange with each other in their work.

Cooperation pictures can be used, among other things, for actual state analysis and for the design of future IT-supported work sequences.

THE PURPOSE

Scenarios, glossaries, and system visions are very useful in describing different tasks and activities from the view of individual workplaces. What is lacking, however, is an overall view of the cooperative situation, especially with complex comprehensive tasks. It is not easy simply to create this overall view, because it is unlikely that any of the application experts will have dealt with this aspect of day-to-day work before the software project started. This problem arises when complex cooperative tasks have to be supported.

Creating the overall view

Application development is often accompanied by organizational changes to introduce new workflows or business processes. For software developers, it is important to understand these new processes and represent the relationship between these processes and IT support in an intuitive way for the users.

There is another, more general problem: The set of scenarios for complex cooperative tasks, such as loan management in a bank or patient admittance in a hospital, normally involves a considerable amount of text, so that these documents are not suitable for overviews or joint discussions with participants.

To solve this problem, cooperation pictures can be used additionally to clarify how and in what way participants should cooperate in a complex environment. Apart from providing feedback on analytical results, cooperation pictures can be used in a joint analysis of cooperative tasks. And finally, cooperation pictures are also suitable in assessing the effects of organizational changes.

THE STRUCTURE

Cooperation pictures are labeled icons representing roles, actors, or workplaces connected by arrows. The arrows between the actors or workplaces are labeled with pictograms and indicate that materials or information is passed on or exchanged. The pictograms on the arrows describe the type of material or information exchanged. Cooperation pictures show especially well how these things are exchanged. Arrows can be numbered so that they present a typical or sample sequence of task completion.

Pictograms symbolize objects or information that is being exchanged between workplaces or actors. If the symbol is a telephone or a little running man, then the medium used to send information or a work object is also obvious.

There are no more rules and regulations for composing cooperation pictures.

EXAMPLE

The cooperation picture in Figure 13.5 shows an example from a used cars company.

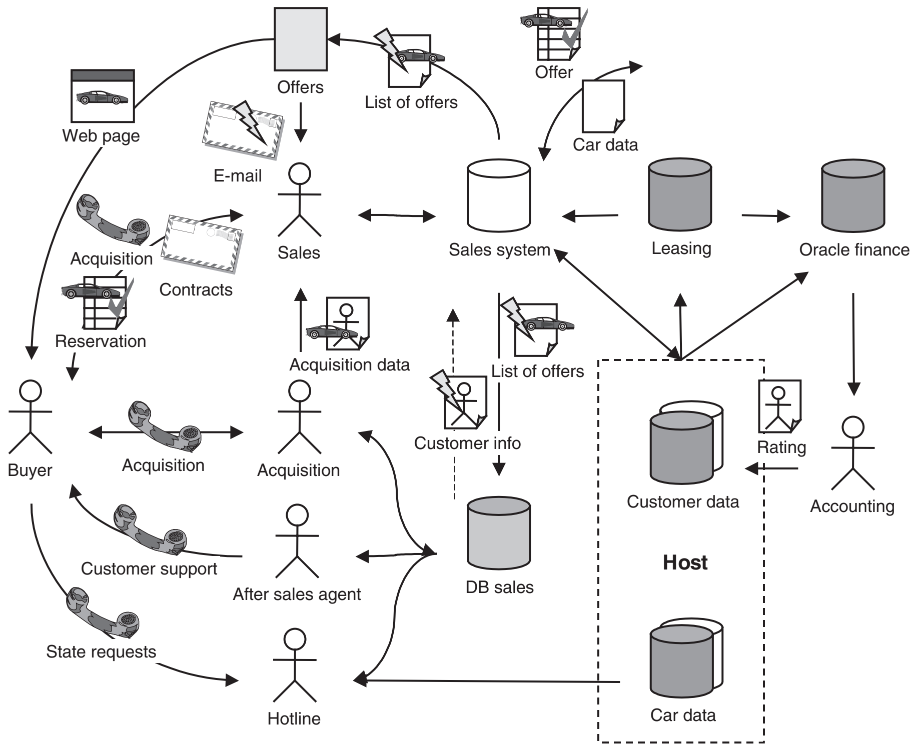


FIGURE 13.5 An example of cooperation pictures.

THE CONTENT

Issues for cooperation pictures

Cooperation pictures show how a cooperative task is or should be carried out between participants in a work situation. Thus they illustrate the current situation or possible future ones, and provide information on the following:

- Which actors, roles, or workplaces are involved in a cooperative task.
- Which object or information is exchanged between them, and which medium is used for exchange and coordination.
- Which roles are covered by the employees or customers of an organization.
- Which typical or special sequences occur.

SUBTYPES OF COOPERATION PICTURES

The following subtypes of cooperation pictures have proven useful in many of our real-world projects:

- *Overviews*: Here the focus is on representing a complex comprehensive task or whole set of tasks. The sequence and individual work steps (see Figure 13.6) are of little interest.
- *Task pictures*: A task picture shows how a comprehensive task is carried out by different people. The individual steps involved between the participants are presented and ordered into sequences (see Figure 13.7).

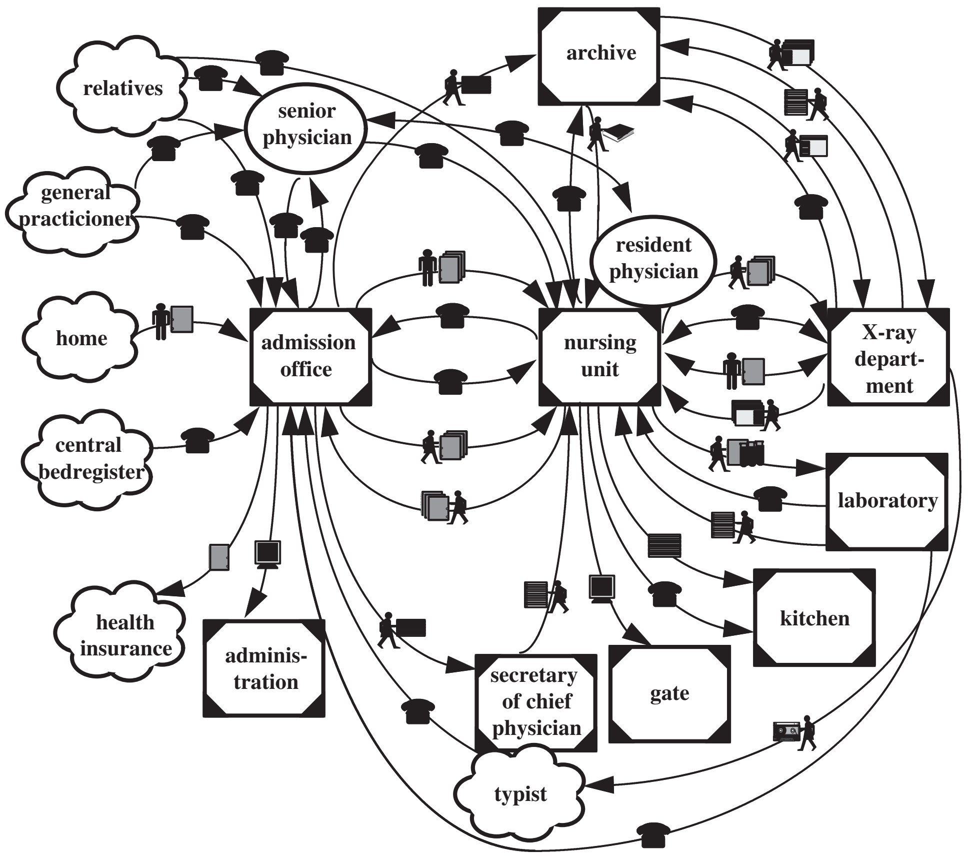


FIGURE 13.6

Using overview cooperation pictures.

- *Planning pictures*: This is a picture of a cooperative task with the different possibilities for supporting it with an application system. Here the potential system support is presented in the different versions of a cooperation picture. The starting point is usually an overview or a task picture of the current situation (e.g., the different extension levels of a hospital information system with their effect on patient admittance). Symbols of system components can be added as materials or cooperation media. Transparencies that are marked or placed one on top of the other, or appropriate presentation software, are suitable for this purpose.

13.7.1 Cooperation Pictures in the Development Process

The way in which a cooperation picture is produced generally depends on its respective purpose.

Cooperation pictures are often created as part of the domain model, as shown in the example that follows. The preliminary work for developers consists in conducting a series of interviews and writing scenarios. The objective is to develop a preliminary understanding of the general tasks and obtain appropriate feedback from users on

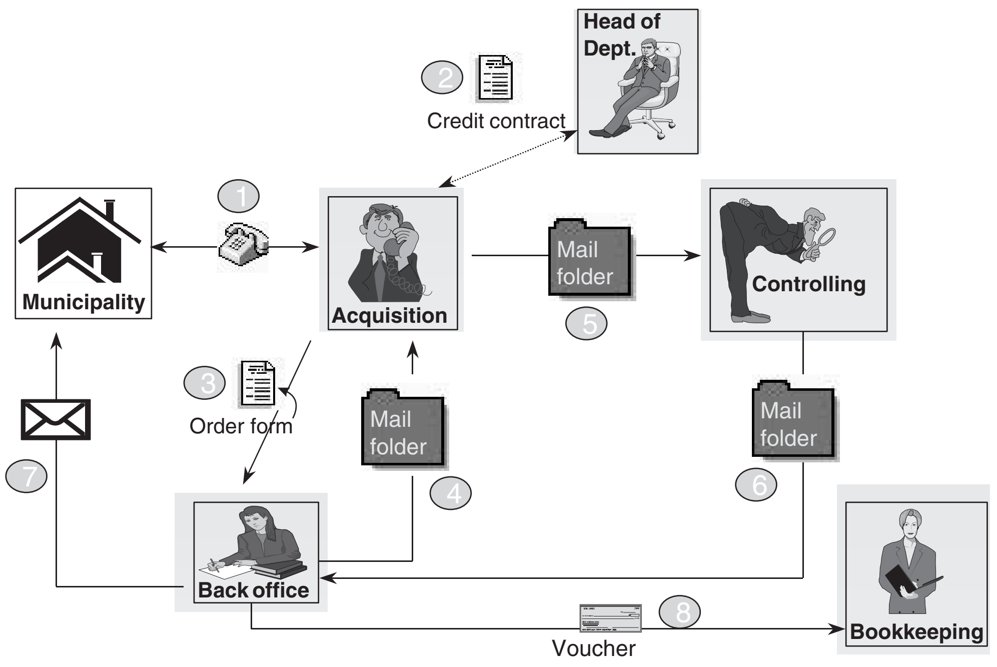


FIGURE 13.7 Using task pictures.

individual work contexts. The setting is formed by a workshop in which small groups work two to four hours, depending on the scope of the task. The participants include:

Participants of a workshop for designing cooperation pictures

- Interview partners
- Employees who are familiar with the respective tasks from their everyday work
- Other members of the user organization who have an interest in or a relationship to these tasks
- Developers

In the workshop, we have usually produced cooperation pictures as *wall pictures*. This involves preparing labeled and unlabeled icons and a large number of different pictograms. To create these wall pictures, we begin with a specification of "start symbols." The cooperation picture is developed from the discussion with the work group. A developer who acts as the moderator or her helpers draws the arrows by hand directly on the paper using a felt-tip pen and affixes the corresponding pictograms and icons to a pin wall. The other developers maintain a passive role and make notes. It is useful if the developers produce a cooperation picture in a preparatory phase. This puts them in a better position to prepare the necessary pictograms and gives them an initial impression of what the cooperation picture covers. Recently, we have started to produce cooperation pictures with an extended version of a graphics tool (Adonis by BoC). This has helped a great deal to speed up the design process.

Cooperation pictures can also be worked out by developers. In this case, they use information from scenarios and other analysis documents. We mainly take this approach when we want to present the potential effects of IT systems on previously modeled work situations. Here, the cooperation pictures help to clarify the consequences of design decisions for developers and users in advance. Ideally, cooperation pictures of the actual situation are already available. Developers can then demonstrate how work processes, responsibilities, and work objects can change in the future. The result can be presented to users in a slide show, using presentation software, with overlays to collect the relevant feedback.

In summary, cooperation pictures accompany the entire development process. Of course, a model of the current situation plays a major role, especially at the beginning of a project. Since developers and users gradually develop an understanding of comprehensive tasks, frequent revisions will be made later using cooperation pictures. Cooperation pictures show the processes taking place as the system develops.

Example

The first cooperation picture (see Figure 13.6) that we ever prepared was in a work session for one of our first hospital projects. All the participants (nursing staff, doctors, administrative personnel, project members) were in for a surprise. It turned out that the regular morning admittance routine for a patient involved up to seventeen phone calls and numerous actions by nursing staff. This realization immediately resulted in a discussion about which procedures were necessary and sensible, and what aspect of the work involved could be handled through computer support. The cooperation picture produced as a wall picture made it obvious—to many participants for the first time—that most of their work does not involve caring for patients and that a considerable part is dedicated to cooperation and documentation work.

Discussion

Cooperation pictures help users and developers in meetings and workshops to work out a common understanding of the character and sequence of cooperative activities. Cooperation pictures are suitable for *users* in the following way:

Benefits for users from using cooperation pictures

- They *promote an understanding* of each person's own work situations. Compared to other means of software representation, cooperation pictures do not have a formal or largely abstract syntax. Instead, they can be understood almost immediately. Users find themselves and their work reflected in these pictures.
- After only a few minutes of familiarization and explanation, users can actively participate in the analysis and planning of tasks *from the outset*. Thus, in a relatively short period of time, a wall picture can be worked out to substantiate how a comprehensive task can be completed by different people, who does what, and why something is done in a particular way.
- Cooperation pictures *illustrate* to users *the complexity* of their work. They are a vehicle for discussion that easily puts users in a position of thinking about their own organization and analyzing it.
- Cooperation pictures promote a *mutual understanding*. Discussions involving work with cooperation pictures clarify to many users (often for the first time) which tasks and subtasks are handled by other organizational areas and how they are executed. This can result in developing a better understanding among different departments.

- Cooperation pictures are effective in helping application management and staff to recognize organizational weaknesses in cooperative work. They serve as the basis for possible changes in the process organization as well as in the structural organization.
- Users can use cooperation pictures to discuss potential changes in workflows in view of a new application system with developers. Since developers can demonstrate the places where a new software system will intervene in the work process, users can quickly assess the effect this will have on their work.

Benefits for developers from using cooperation pictures

Cooperation pictures are also suitable for *enabling developers* to familiarize themselves with complex application domains that have an extensive division of labor.

- Cooperation pictures are a technique used for *feedback with user groups*. They enable developers to check whether their view of work contexts based on individual scenarios corresponds to the users' view. Unclear points can be solved quickly.
- *Unknown connections* can be worked out, so that developers are able to recognize "a piece of uncharted territory" in their analysis. They receive tips on which actors to interview.
- Cooperation pictures *substantiate* the interrelations between workplaces. They show how and where information and communication flows, and how objects are forwarded. They provide the background needed to *identify* cooperation forms and categorize objects, particularly documents, according to their importance for cooperation and coordination.
- Substantiating cooperation connections also shows the "density" of information flow. This provides the opportunity for an *evaluation* of the cooperation forms that a system should support.

RELATION TO OTHER DOCUMENT TYPES

In some cases, cooperation pictures in their pure form may not be understood by third parties, because they usually develop as a result of direct dialogs between developers and users. Moreover, most of them are not detailed enough to exist as independent models.

Developers need scenarios before they can create a cooperation picture and follow what is said in discussions with users. The reason is that scenarios clarify the tasks at individual workplaces. The glossary supplies the necessary explanation of terms. Therefore, a combination of cooperation pictures, related scenarios, and a glossary usually provides a solid basis for discussion even with people not directly involved in the software project.

A cooperation picture with purpose tables (see Section 13.8) that detail and motivate certain segments can be supplied for ongoing work.

Since cooperation pictures clarify which tasks an employee can complete, which tasks are completed by others, and which ones can only be carried out jointly, this has an impact on the character of workplaces and work environments (see Section 7.6). It affects all other design documents, the prototypes, and ultimately the application system.

Cooperation pictures provide the basis for prototype screenplays and prototyping sessions (see Section 13.6.1) in which future task handling is "acted out" according to typical tasks from scenarios and domain-oriented use cases.

13.7.2 Cooperation Pictures and UML

UML uses a diagram type that models workflows and cooperative work; these diagrams are called *activity diagrams*. Activity diagrams are actually flowcharts that show a system's flow of control from activity to activity. Activity diagrams do not give a truly object-oriented view of work processes, because—at best—work objects can be represented in the names of the activities or by using *object flows* to extend a diagram.

We found that, even with these extensions, activity diagrams are a presentation tool for developers and are not really suitable for cooperative work with users. This means that UML lacks an application-oriented diagram type similar to cooperation pictures, which is why we also use them successfully in UML-based projects.

13.8 PURPOSE TABLES

A *purpose table* lists selected procedures or work processes, specifying who does what with whom and with what. The "what" is explicitly aimed at the purpose of an activity or the use of an object.

A method from the *object behavior analysis* (OBA) proposed by Rubin and Goldberg serves as the conceptual basis of purpose tables. This method describes tasks according to the pattern "who – does what – with what – which service must an object provide."

The Purpose

Purpose tables describe procedures or work processes that involve several people. They essentially record why or for what purpose objects are worked with and why they are passed on. Purpose tables therefore mainly complement cooperation pictures for selected tasks. They are used in the detailed modeling and feedback of these tasks and activities. Purpose tables are also useful in clarifying whether these tasks should be maintained or modified, based on the domain, and if so, how.

Purpose tables show the details of the dynamics of individual cooperation relationships. They mainly address domain-related questions without dealing with technical issues.

The Structure

Purpose tables are named according to the task they describe. The simplest purpose tables consist of two columns. The first column lists the activities (of the task described) while the second column shows why an individual activity is carried out and what its consequences can be (see Table 13.2).

The rows in purpose tables may be numbered. This is useful when the table is used to refer to the steps in a cooperation picture, such as in Figure 13.7. Then the numbering relates to the numbers on the arrows in the cooperation picture. If used on their own, a numbering scheme is useful for sequencing and cross-referencing.

The Content

We represent selected cooperative tasks in purpose tables to highlight the significance of the objects used in a work situation. In the T&M approach, the objects of an application domain form the starting point for domain modeling.

Individual Activities of an Order Entry	Purpose/Implications
Physician writes the order on the physicians order form.	It is documented who ordered the test at what (forensic, quality assurance). To kick on the implementation of the test.
Physician puts the order entry sheet in the nurse's mail basket.	Nurse is alerted that she has to act. She knows what is planned with her patient.
Nurse enters patient's name, other relevant data and the type of test on the order entry sheet.	Nurse prepares the order entry sheet in order to relieve the physician of such burdens.
Nurse enters the test with pencil on the patient's flowsheet.	It is documented for every member of the care team and physicians when the examination was ordered and to which further examinations he is scheduled.
Nurse puts the order entry sheet in the physician's mail basket.	Physician knows that he has to validate the order.
Physician sees the order entry sheet in his basket, enters the relevant clinical information, signs it and puts it in the nurse's mail basket.	The physician that carries out the test knows what to do and that the ordering physician is responsible for the test.
Nurse carries the order entry sheet to the X-ray department.	The X-ray department can schedule the test and the performing physician can check the order.
Radiology technician chooses a date for the test and conveys it by phone to the unit.	The tests are coordinated within the X-ray department. The nurses know when to take the patient to the X-ray Department.
Nurse enters the date of the test in the units calendar.	Whole nursing stuff knows about the date.

TABLE 13.2 Example of a purpose table.

We have said that there is no one-to-one representation of the objects in the model of the application system. Consequently, there is a risk of overlooking the purpose and various implications of an object when modeling cooperative tasks. In the example shown in Table 13.2, it is important that the order entry sheet be located at a particular spot depending on what is being worked on. This is how doctors and nurses coordinate their work.

If we decide not to model an object in a system or to do so only with certain selected characteristics, then we can use purpose tables to estimate the related effects on the organization of cooperative work and discuss this with application experts.

13.8.1 Using Purpose Tables in the Development Process

We have pointed out that, in the case of comprehensive tasks involving a division of labor, there is no "single user" to whom developers can direct their questions, because the users themselves are only able to make a limited assessment of the detailed relationships of their individual activities (see Section 13.7). As a consequence, it is difficult to check new processes and cooperation forms that may be introduced with a new application system as to whether they are meaningful in everyday practice and support the work of the users. Purpose tables are helpful in forming a good basis for discussion.

Purpose tables are created by developers based on scenarios to supplement cooperation pictures. They are usually prepared before the actual evaluation process with the users. When complex comprehensive tasks are analyzed, purpose tables are prepared together with users and along with cooperation pictures in workshops.

Purpose tables are a useful means for supporting communication and learning processes, because they promote intensive domain-specific discussion with the application experts. They make users aware of details and implicit relationships in the application domain.

Purpose tables usually cover only sections of an application model. We normally select the critical sections for use in modeling the future system. Therefore, the choices written in purpose tables are always design decisions.

Purpose tables ultimately have an effect on the organizational development. They help to define the grounds for easy implementation of organizational changes. In particular, they show why something is done. Thus, if a task disappears or is handled by another person or through software support, we have to ensure that the purpose associated with that task is still achieved.

Relation to Other Document Types

In addition to their obvious relationship with cooperation pictures and system visions, purpose tables are also associated with design documents. For tasks based on a division of labor, the objects of an application are often important for cooperation. Purpose tables characterize an important aspect of domain objects that we have to consider in the class model.

Since purpose tables form the basis for analyzing the effects of modeling domain objects in the future system, they also influence prototype development. Finally, purpose tables are not a substitute for scenarios. They "consolidate" knowledge about the purpose of objects and task connections.

13.8.2 Purpose Tables and UML

In our view, UML does not have a diagram type similar to purpose tables. All UML diagram types are useful in presenting the *what* and *how* in the development process, but they do not show *why*.

13.9 TECHNICAL DOCUMENT TYPES IN UML

All document types, and particularly diagram types that developers use for the technical and construction aspects of software development, are considered technical document types.

Technical documents are not application-oriented

We are very pragmatic in our selection of such technical document types. The reason is that important criteria of application-oriented documentation do not apply here (see Section 5.3.9). Technical documents are designated for software developers and should be comprehensible and easy to use for this target group. Furthermore, technical document types should serve their purpose and describe the relevant aspects of large object-oriented application systems.

During the last few years, we have observed a stronger concentration on document types that more or less meet these criteria effectively and have become one of the basic tools in object-oriented software development. These document types have been described and used in many publications in recent years. The quasi-standardization of UML has combined the key document types and notations for technical design. We outline this choice below and report on our experience with UML. We assume that the reader is familiar with the graphic notations of the different diagrams or will look them up in the relevant literature.

Our view of technical documents

Our general opinion is as follows:

- Technical document types as such are frequently overrated. A multitude of diagrams cannot amend a poor design.
- Technical document types cannot be evaluated by domain experts, if only technical documents are used so there is the inherent chance of missing the domain-related goal of a project.
- Technical documents should be able to capture the elements and relations of the technical (programming) model used.
- Experienced developers should combine the diagrams and technical document types that best meet their requirements.

UML currently defines the following nine diagram types:

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Statechart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram

13.9.1 Class Diagrams

In this book, we use class diagrams to represent the statics of object-oriented programs. In the use relationship between classes, we graphically distinguish object references, aggregations, and creation. We use the general undirected associations of UML as little as we use association classes.

Until now we have never provided a complete graphic specification of a system, so our class diagrams always show only the visible interfaces or operations. Consequently, we have not yet used the visibility modifiers of UML.

13.9.2 Object Diagrams

We use object diagrams to present object networks in object-oriented programs at run-time. We find it confusing to mix classes and objects in one diagram, something that UML allows.

13.9.3 Use Case Diagrams

Use case diagrams provide an overview of use cases and participating actors. In our view, they are useful for combining and discussing business use cases, use cases, and scenarios. In addition, they can be used in an actual/target comparison: existing tasks with their actors can be compared with the planned situation. Figure 13.2 shows a simple use case diagram.

13.9.4 Interaction Diagrams

To show the dynamics of object-oriented systems in this book, we have used the object-interaction diagrams originally introduced by Jacobson. With some minor extensions, they correspond to those recommended in the UML standard, which calls it *variant sequence diagram*. We find collaboration diagrams confusing and do not use them.

13.9.5 Statechart Diagrams

Statecharts originate from the world of reactive technical embedded systems and were developed by Daniel Harel. They are suitable to model the behavior of systems that can be described by the principle of state machines or finite automatons.

We have primarily used statecharts to model critical classes or, more precisely, instances of these classes. The theory and formal notation of statecharts are relatively complex. Unfortunately, this is not well covered in the UML literature. For projects that have to model technical systems, we therefore recommend the relevant original literature or the work of Bruce Douglass.

In addition to strictly formalized statechart diagrams, we often use domain-motivated statecharts for state modeling of operations on objects (see Section 12.4.2).

13.9.6 Activity Diagrams

In our discussion of cooperation pictures (see Section 13.7), we also looked at activity diagrams. We stressed their limited suitability for cooperation with users. However, we still have not used this diagram type for technical modeling and construction. So far in our real-world projects, we have noticed that activity diagrams induce developers to take an imperative and procedural view of a design. Developers from traditional environments are initially able to deal well with this form of flowcharts. However, there is a risk that processes become more important than objects. These developers consequently write object-oriented software in the form of a control object (main program) with calls of stateless procedural objects (subprograms).

13.9.7 Component Diagrams

Component diagrams can be used like class or object diagrams. Since they are good at representing interfaces, they are another means of expressing the static relationships of a system.

13.9.8 Deployment Diagrams

We have not yet used deployment diagrams and, to our knowledge, they have not been used in any other project. On the other hand, such diagrams are certainly useful for the actual deployment of a software-hardware system.

13.9.9 Application-Oriented and Technical Documents

The role of documentation has changed over the last ten years. Traditionally, software engineering stressed the importance of formal technical documents in the development process. While this type of documents still plays a predominant rule in the design of technical embedded system, the importance of application-oriented documents for application system development has become clear. The current discussion on agile methods like eXtreme Programming again has shifted the focus: we should minimize documentation to that extend which is needed for communication between developers and customers. The resources saved should better be invested in additional iterations of the software. But we are sure that the discussion on documentation has not reached its end point.

13.10 REFERENCES

B. P. Douglass: *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*. Reading, Mass.: Addison-Wesley, 1999.

A seminal work on UML for real-time systems.

G. Booch, J. Rumbaugh, I. Jacobson: *The Unified Modeling Language*. Reading, Mass.: Addison-Wesley, 1999.

The current standard work on UML.

K. Beck, W. Cunningham: *A Laboratory for Teaching Object-Oriented Thinking*. ACM SIGPLAN Notices, Conference proceedings on object-oriented programming systems, languages and applications, Volume 24, Issue 10, 1989, 1–6.

The seminal paper that introduced CRC cards.

C. Floyd: *A Systematic Look at Prototyping*. In: Approaches to Prototyping. R. Budde, K. Kuhlenkamp, L. Mathiassen, H. Züllighoven (Hrsg.). Berlin, Heidelberg: Springer-Verlag, 1984, pp. 1–18.

A seminal paper which has coined the terminology and concepts of prototyping.

D. Harel: *Statecharts: A visual formalism for computer systems*. Science of Computer Programming, 8/3, 1987, pp. 231–274.

The seminal paper on Statecharts.

I. Jacobson: *Object-Oriented Software Engineering–A Use Case Driven Approach*. Reading, Mass.: Addison-Wesley, 1992.

The book in which Jacobson originally introduced his use case concept.

I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*. Reading, Mass.: Addison-Wesley, 1999.

Currently the standard work on procedures in projects in the context of UML.

A. Krabbel, I. Wetzel, S. Ratuski: *Participation of Heterogeneous User Groups: Providing an Integrated Hospital Information System*. In: Proceedings PDC-Conference, Boston, November 13–15, 1996, pp. 241–249.

A paper introducing the principles of cooperation pictures.

M. M. Lehman: *Uncertainty in Computer Applications is Certain*. In: Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering, IEEE, Tel Aviv, May 1990.

Lehman shows in this paper that we cannot understand software on the basis of its code only.

K. S. Rubin, A. Goldberg: *Object Behaviour Analysis*. In: CACM, Vol. 35, No. 9, September 1992, pp. 48–62.

The paper introduces the object behaviour analysis which is a source of our purpose tables.

INDEX

abstract classes, 34–35, 420
  defined, 34
  example, 35
  operations declared in, 34
  patterns with, 35
  specializing, 35
  See also classes
abstract data types
  classes as, 39
  contract model and, 39–40
abstract factory, 330
abstract operations, 34
Abstract Windowing Toolkit (AWT), 229
ACID properties, 371
activity diagrams, 498
activity scenarios, 466
actors
  defined, 465
  human, 483
  technical, 483
adapter pattern, for class collaboration, 198
addressability, 24
adjusting tool pattern, 181–183
  architecture, 260
  background, 182
  illustrated, 181
  problem, 181–182
  rationale, 183
  relationship, 182
  solution, 182
  See also conceptual patterns
adjusting tools
  design, 182
  in multiprocessing space, 260–261
  with probes, 183
  technical automatons and, 259
  using, 182–183
  See also tools
altering states, 67
announcing, 262
application domains, 118, 119–120, 427
  actual state, 112
  business domain layer and, 311
  context, 127–128
  defined, 119
  domain model of, 283
  domain-specific language, 119
  importance, 119
  modeling, 120–122
  size of, 427
  software development and, 127
  types of, 119
  work environments in, 156
  See also domains
application frameworks, 91–92
  characteristics, 92
  complexity, 92
  defined, 91
  See also frameworks
application orientation, 2, 102–103, 394
  background, 103
  characteristics, 102
  defined, 102, 105
  focus, 4
  selection, 427
  tasks and, 102–103
application software
  characteristics, 102
  defined, 2, 102
  designing, 57–59
  guiding metaphors, 59–63
  implementation, 102
  interactive, 2, 5, 104
  large-scale, 2
  usage quality, 4
application system model, 122–125
  defined, 122, 123
  design model, 123–124
  examples, 124–125
  implementation model, 124
  structural similarity, 123
  submodels, 123
  usage model, 123
  See also software development model
application systems, 125–127
  characteristics, 126
  connecting, to databases, 358
  defined, 126
  documentation, 402
  elements, 126
  embedded, 178
  influences, 127
application-oriented document types, 7, 109–110
  defined, 109
  technical documents and, 499
  See also document types
application-oriented software development, 2, 101–115
  arguments for, 101
  author-critic cycle, 107–108
  characteristics, 105
  defined, 105
  as document-driven process, 106
  documents, 109–110

application-oriented software development (*continued*)
  motivation for, 101
  objectivity and independence, 106
  prerequisites, 106
  process, 105–115
application-oriented transparency, 344–345
applications
  distributed, 54
  handling/presentation and, 308
  "look and feel," 118
applied technique, 128–129
  defined, 128
  example, 128
  system base, 128–129
  technology, 129
  See also software development contexts
architectural style, 282
architecture group, 411–412
  defined, 412
  qualification profile, 412
  responsibilities, 412
  See also IT organization development
archive management, 359
aspect classes, 195–196
  standard implementations and, 195
  subsystems and, 198
  tool use of, 196
  use benefits, 195
aspect pattern, 190–201
  background, 192–194
  defined, 187
  example, 192
  illustrated, 190
  intent, 190
  problem, 190
  rationale, 194
  relationship, 191
  solution, 191
  trade-offs, 194
aspects
  behaviors, specifying, 194
  characteristics, 191
  construction obstacles, 194
  generalizing, 197
  implementing (inheritance/interfaces), 195–198
  implementing (object adapters), 198–200
  inheritance, 195–198
  interfaces, 193, 195–198
  materials and, 194, 197
  multiple inheritance and, 195
  realizing (development tools), 200–201
  single inheritance and, 194
  static typing, 196–198
  use alternatives, 201
asynchronous communication, 262
asynchronous coupling, 261
author-critic cycle, 107–108, 413
  defined, 8, 107
  example, 108
  schematic, 107
  task analysis, 107
automaton pattern, 167–171
  defined, 138
  example, 168
  illustrated, 167
  intent, 167
  problem, 167
  relationship, 168
  solution, 168
  trade-offs, 170
  See also conceptual patterns
automatons, 70–71
  adjusting, 261–262
  asynchronous communication, 262
  big, 71
  building, 168
  characteristics, 70–71
  database, 381–382
  defined, 6, 70
  as design metaphor, 70–71
  for domains, 252
  for functional workplaces, 168
  implementing, 168
  interactive user interfaces, 170, 182
  large, 170
  in multiprocessing space, 260–261
  persistence, 389
  for routines, activities, 170
  for routines, criteria, 170
  run location, 70
  small, 71, 168
  software, 71
  in T&M design, 71
  as technical processes, 169
  tools vs., 169
  use of, 169
  uses, 170
automatons in embedded systems pattern, 258–264
  defined, 189
  example, 261–263
  illustrated, 258
  intent, 258
  problem, 258–259
  rationale, 264
  relationship, 259
  trade-offs, 263–264
  See also design patterns

back-office workplace type, 75–76
  characteristics, 75
  defined, 75
  handling, 76
  See also workplace types
bank example
  application system model, 124
  business domain modeling, 291–293
  business domains, 289–290
  combination of layers, 304–305
  combination of layers with bridge pattern, 305
  customer orientation, 409
  direct manipulation guiding metaphor, 61
  domain hierarchy, 328
  domain model, 121
  domain relationships, 293–294
  domain values, 49
  ideal business domains, 291
  layered architecture, 313–314
  minimal core system, 435
  object life cycle, 29
  object-oriented layer architectures, 300
  open-closed principle, 301–302
  product domain, 288
  project goals, 430–431
  use context, 286

work environment, 155
workplace types, 286–287
base lines, 441–443
checking, 441
defined, 441
defining tasks, 441
discussing, 442
entering, 443
example, 442–443
scheduling with, 443
in Unified Process, 442
See also project planning
base operations, 35
big automatons, 71
black-box frameworks, 92–94
connecting, 94
defined, 92
using, 92
white-box frameworks comparison, 93
See also frameworks; white-box frameworks
black-box tests, 422
BLOBs, 385
body/handle pattern, 242
breadboards
content, 485
defined, 484
See also prototypes
business domain layer, 311–312
application domain and, 311
defined, 311
implementing, 311
reusing parts of, 312
business domains, 289–293
company organization and, 291
concepts/objects, 290
constructors in, 335
defined, 290
design criteria, 290
elements, 290
example, 289–290
ideal, 291
minimal, 290
modeling, 291–293
overlap, 289
restructuring, 293
See also domains
"business modeling," 121
business transactions, 175
business use case diagrams, 467
C++
types and, 45
using, 219
calibration, 428–430
dimensions, 428, 429
important aspects of, 429–430
See also projects
Certified Aircraft Engine Software project, 11–12
chain of responsibility, 217–219
event pattern vs., 218
example, 218
functionalities, 217
pattern, 82
rationale, 219
trade-offs, 219
class diagrams, 473, 497
class evolution, 382
class libraries, 90–91
as basic elements, 297
characteristics, 90
defined, 90
example, 91
open-closed principle and, 90–91
class mapping, 389–392
class-based languages, 19
classes, 25–27
abstract, 34–35, 420
as abstract data types, 39
aspect, 195–196, 198
collaboration, 198
defined, 18, 25, 26
in design, 43
in domain model, 25–26
domain value, 238–240
form, 253
illustrated representation, 26
inheritance relationship, 18
modules and, 36
organization, 422
in software model, 26
superclass, 238
testing, 421–422
type differences, 43–44
types and, 43–45
user-defined, 49
client-server architecture, 374–378
persistence-capable clients, 374–375
persistence-capable servers, 375
SyLab, 387, 387–388
trade-offs, 375
cohesion, 37
`Collection` interface, 277
combination tools
constructing, 209–210
defined, 206
event pattern for, 214
feedback problem, 213
implementing, 209
See also tools
command objects, multiple registration, 235
common user access (CUA) guidelines, 114–115
company organization, 284
complex tools, 216–217
component diagrams, 498
components, 96–99, 297
background, 80
for building combination tools, 209–210
for building tools, 208–209
characteristics, 97
current products, 98
defined, 79, 97
frameworks and, 98–99
implementation, 97
loosely coupled, 36
programming, 97
runtime, 97
software, 96
technical, representing, 178
technologies, using, 410
composability, 37
composition relationship, 473, 474
Computer Aided Software Engineering (CASE) tools, 8, 127
computer-supported cooperative work (CSCW), 58, 59, 341–345
defined, 341, 342
as important design issue, 341–342
See also cooperative work
concept models, 472–475
content, 473
defined, 472
in development process, 475
document types and, 475
elements, 474–475

concept models (*continued*)
example, 473–474
generalization, 473, 474
purpose, 472
structure, 472–473
UML and, 475
See also document types
conceptual patterns, 3, 83–84, 87, 135–183
adjusting tool, 181–183
allocating, 84
automaton, 138, 167–171
container, 138, 159–163
defined, 83
in development process, 136
domain service provider, 138–139, 171–176
form, 138, 164–167
guided tour, 137–139
as guideline, 84
hierarchy, 137
illustrated, 87
interrelation of tools and materials, 84, 137–138, 139–142
list, 136
material design, 138, 142–145
probe, 179–181
problems addressed by, 136
T&M, 136
technical automaton, 139, 176–179
tool design, 138, 146–151
work environment, 138, 152–159
See also patterns
conceptual stage (UP), 445
conference planning, 440–441
configuration scripts, 337
connectors, framework, 94
consistency checks, 157, 383
general principle, 158
types of, 157–158
See also work environment
construction guidelines, 87
construction stage (UP), 446
`Container` interface, 277
container pattern, 159–163
background, 162–163
defined, 138
example, 160–161
illustrated, 159
intent, 159
problem, 159
rationale, 163
relationship, 160
solution, 160
trade-offs, 161–162
See also conceptual patterns
containers, 71–73
changes to, 250–252
characteristics, 72
contents, 161
data structures and, 163
defined, 71
as design metaphor, 72
domain, 162, 163, 244–252
explicitly modeling, 163
loading, 247
management function implementation, 162
material transport, 162
as materials, 162
modeling, 245–246
moving, 160
order principles, 161–162
ordering, 72
polymorphic, 161
processes and, 72
representations, 71
services, 160
specialized, 162
in T&M design, 72–73
tables of contents, 248–249
technical, 163, 246–247
transport medium implementation, 162
contains relationships, 144
context tools
defined, 206
designing, 210
See also tools
continuity, 37
continuous feedback, 399
contract model, 38–40
abstract data types and, 39–40
defined, 38
introduction, 38
service provider/client in, 39
T&M design and, 40
contracts
defined, 38
operation calls and, 38
pre-condition specification, 38–39
violations, 40
cooperation
explicit, 343, 346–356
implementing, 158–159
implicit, 343, 345–346
material, 343
routinized, 353
rules, 432
support with mailing system, 349
cooperation medium
defined, 343
mailboxes, 346–348
mailing system, 349–350
routing slips, 354–355
cooperation model
defined, 342
mailbox systems, 349
mailboxes, 347
for registry, 361–363
routing slips, 354–355
cooperation model (registry), 361–363
access to copies only, 362
background, 361
example, 363
exclusive access, 361–362
illustrated, 363, 366
problem, 361
solution, 361
trade-offs, 363
usage model with, 365
cooperation pictures, 488–494
benefits, 492–493
content, 489
defined, 8, 488
designing, 491
in development process, 490–493
document types, 493
example, 488–489, 492
illustrated, 489
issues, 489
mutual understanding and, 492
overviews, 489, 490
planning pictures, 490
purpose, 488
structure, 488
subtypes of, 489–490
task pictures, 489, 491
UML and, 494

unknown connections and, 493
See also document types
cooperative development process, 402, 405–406
cooperative work, 341–356
analysis, 7
characteristics, 342
computer-supported, 58, 59, 341–345
coordination and, 342–343
defined, 6
domain concepts, 358
original and copy in, 361
Cooperative Workplace System project, 13
coordination
cooperative work and, 342–343
defined, 343
phase, 408
core concepts, 320, 322, 327
core objects, 317
defined, 318
dynamic extension, 322
extending properties of, 322
illustrated, 319
interaction, 320
interfaces, 321
use relationship, 322
core system
coordination, 432
defined, 433
example, 433–434
extension levels and, 434–436
minimal, 434–435
with special-purpose systems, 434
coupling
defined, 37
loose, 35–36
coupling tools and materials pattern, 85
CRC cards, 472
creating objects, 28
current work objects, 157
customer orientation, 1–2, 409
defined, 285
example, 409
implementation, 285
customers
group segmentation, 285
satisfaction, 2
uniform image, 173
cyclic structures, 296–297

data consistency, 371
data warehouses, 379
databases
automaton, using, 381–382
connecting application systems to, 358
legacy, 382–383
relational, 377–378
decision principles, 431
decomposability, 37
deleting objects, 28
dependency relationship, 473
deployment diagrams, 498
descriptive software development model, 118
design guidelines, 58–59, 114–115
design metaphors, 5, 63–64
automaton as, 6, 70–71
basis, 87
characteristics, 64
container as, 72
defined, 6, 63, 64
example, 64
materials as, 6, 68
patterns and, 66–67
requirements, 64
tools as, 6, 7
in uniform application system design, 287
work environment as, 6, 69–70
See also metaphors
design model, 123–124
defined, 23
representation, 123
design patterns, 3, 84–85, 185–279
aspect, 187, 190–201
automatons in embedded systems, 189, 258–264
characteristics, 84
construction parts, 86
coupling tools and materials, 85
defined, 83
descriptive part, 86
domain container, 244–252
domain services, 189, 264–268
domain values, 188, 236–243
environment, 189, 268–271
example, 85
feedback between interaction forms and IP, 188, 232–236
feedback between tool parts, 188, 212–220
form system pattern, 189, 252–258
frameworks and, 85
guided tour, 187–190
hierarchy, 186
illustrated, 87
introduction, 185–186
for JWAM framework, 190, 271–279
object comparison, 333
overview, 187–190
product trader, 327–338
programming guidelines and, 114–115
role, 315–327
separating FP and IP, 221–227
separating function and interaction, 187, 202–204
separating handling and presentation, 188, 227–232
T&M model architecture, 86, 314–338
tool composition, 187–188, 204–212
in uniform application systems design, 287
uses, 85
See also patterns
design stage (UP), 445–446
development domain, 428
development process, 8, 393–455
activities sequence, 400–401
application-oriented, 410
concept models in, 475
cooperation pictures in, 490–493
cooperative, 402, 405–406
defined, 395
division of labor/depersonalization, 401
evolutionary, 394
glossaries in, 477–478
IT organization development, 409–412
models in, 452–454
objectifying, 401–402
organizational/domain integration, 406–409

development process (*continued*)
project management, 425–428
project planning, 428–432
project structuring, 432–436
purpose tables in, 495–496
quality assurance, 413–419
scenarios in, 462–464
scheduling, 437–451
staff fluctuation in, 406
strategy, 400–412
system visions in, 480–482
T&M, 451–455
task planning, 437–451
UP, 451–455
development tools, for aspect realization, 199–200
direct manipulation guiding metaphor, 61
direct mapping, 38
direct user integration, 413
distributed applications, 54
distributed communication usage model, 263
Distributed Component Object Model (DCOM), 96
document types, 459–499
application-oriented, 110–113
classification, 111–113
concept model, 472–475
cooperation pictures, 488–494
defined, 110
glossaries, 475–479
identified requirements, 114–115, 463, 470
interviews, 467–472
prototypes, 483–487
purpose tables, 494–496
scenarios, 459–467
system visions, 479–483
T&M, 113
technical, 112, 496–499
documentation, 108–109
defined, 108
guidelines, 114–115
problems, 109
writing, 109
document-based modeling, 398–399
documents
application-oriented, 109–110, 499
defined, 109
in development process, 111
domain, 425
elaborating, 109
milestone, 109, 113
planning, 426
project, 113–115
project control, 114
ranking within communication process, 112
types, 110–113
in UML, 110
domain container pattern, 244–252
background, 245
example, 245
illustrated, 244
intent, 244
problem, 244–247
rationale, 246
relationship, 245
solution, 245
trade-offs, 245–246
See also design patterns
domain containers
building, 245, 246
design, 244
modeling, 245
using, 364
domain core
application, 283–284
business domain, 289–293
domain relationships, 293–295
product domain, 287–289
software architecture, 283–295
use context, 285–287
domain functionality, 143
implementing, 265
in material design, 143
domain model, 20, 120–122
application domain aspects, 121
building, 119–120
class diagrams for, 473
classes, 25–26
defined, 120
inheritance, 30
interrelations, 20
objects, 21
scenarios as part of, 460
structural similarity, 122–123
domain service provider pattern, 171–176
background, 172–173
defined, 138–139
example, 172
illustrated, 171
intent, 171
problem, 171
rationale, 176
relationship, 171
solution, 171–172
See also conceptual patterns
domain service providers
behind application frontends, 173
business transactions and, 175
collections of functions and, 175–176
defined, 172
implementation, 172
information management, 175
resources and, 174
as stateful, 174
domain services
clients, 267
functionality, 265
implementing, 265
JWAM framework, 279
knowledge, 265
operations, 267
requirement, 265
technical features, 267
domain services pattern, 264–268
defined, 189
example, 266–267
illustrated, 264
intent, 264
problem, 264–265
rationale, 268
relationship, 265
solution, 265–266
trade-offs, 267–268
See also design patterns
domain transparency, 344–345
domain values, 49–50
building, 50
classes, 238–240
composite, 237
data types, 237
defined, 45, 49
example, 49
factory, 276
finite, 237
for FP-IP interaction, 239–240

for IAFs, 140
immutable objects, 240–241
implementing, 49–50, 237
implementing, as streams, 243
infinite, 237, 238
JWAM framework, 274–277
motivation for, 49
mutable, 242–243
objects, 276
operations on, 50
for PFs, 240
summary, 50
in T&M design, 49
tables of contents
implementation as, 249–250
types, 49
types by configuration, 243
using, 239–240
See also values
domain values pattern, 236–243
background, 238
defined, 188
illustrated, 236
intent, 236
problem, 237
rationale, 238
relationship, 237
solution, 237
See also design patterns
domains
application, 118, 119–120, 127–128, 427
business, 289–293
change, handing within project team, 295
change impact on architecture, 295
development, 428
integration, 408–409
product, 287–289
relationships, 293–295
separation of, 305
technology, 427–428
terms, reconstruction of, 120
use contexts, 285–287
domain-specific identity, 24
dynamic binding, 34
dynamic identifiers, 22

E programs (Lehman classification), 403–405, 406
author-critic cycle and, 405
defined, 403
evaluation, 405
illustrated, 404
understanding of, 406
eclipse component model, 98, 99
Eiffel, 45
electronic commerce frontend workplace type, 76–77
characteristics, 76
defined, 76
See also workplace types
embedded application systems, 178, 259
Embedded Medical Lab System project, 12
encapsulation, 24, 419
business transactions, 175
data exchange and, 158
defined, 24
with IAFs, 230
implementing, 158
system platform, 231
use, 26
engineering cards, 450
Enterprise JavaBeans, 98, 99
enterprise resource planning (ERP), 98
environment pattern, 268–271
background, 269
defined, 189
example, 270
illustrated, 268
problem, 268
rationale, 271
relationship, 268
solution, 269
trade-offs, 271
See also design pattern
environments
building, 269
concepts, 269
defined, 269
features, 270
functions, 269
JWAM framework, 279
tools/materials in, 269
workspace and, 270
Equipment Management System (EMS) example
activity scenarios, 466
application domain, 120
application system model, 124–125
applied technique, 128
aspect interface, 195
aspect pattern, 192
author-critic cycle, 108
automatons, 71
business use case diagram, 467
chain of responsibility, 218, 219
combination tool building, 210
concept model, 473–474
conceptual patterns, 84
container changes, 250, 251–252
container pattern, 160–161
containers, 72–73
defined, 14
device description, 21
direct manipulation guiding metaphor, 61
domain container implementation, 246
domain container pattern, 245
domain model, 122
domain service provider pattern, 172
event objects, 215–216
feedback between IAFs and IP pattern, 234
flyweight pattern, 241
form pattern, 165–166
form system pattern, 255
glossary, 477
handling and presentation, 130
interrelation of tools and materials pattern, 140
material design pattern, 143–144
materials, 69
object life cycle, 29
patterns, 81
room plan, 15
scenarios, 461
separating FP and IP pattern, 222–223
separating handling and presentation pattern, 231
software tools, 68
system vision, 480
tool boundaries identification, 211
tool composition, 206–207

Equipment Management System (EMS) example (*continued*)
  tool design pattern, 147–148
  work environment, 70
  work environment pattern, 153–154
equipment manager, 14–15
ethnographic video-supported studies, 462
event objects, 215–217
  example, 215–216
  rationale, 217
  trade-offs, 216–217
  See also objects
event pattern, 214–215
  for combination tools, 214
  with event objects, 216
  example, 275
  rationale, 215
  trade-offs, 214–215
evolutionary cycles, 443–444
  defined, 444
  phases, 444
evolutionary process model, 398–399
  actual/target states adjustment, 399
  constant feedback, 399
  document-based modeling, 398–399
  idealized, 398–399
  illustrated, 398
  principles, 398
  See also process models
evolutionary prototypes, 484, 487
evolutionary system development, 108
  characteristics, 108
  defined, 7, 108
exclusive access, 361–362
  to materials, 361–362
  on original, 362
experimental prototyping, 483
expert activities, 66
expert workplace guiding metaphor, 64–66
  materials, 65
  supportive view, 65–66
  tools, 65
expert workplace type, 74
  characteristics, 74
  defined, 74
  See also workplace types
explicit cooperation, 346–356
  by exchanging materials, 346–350
  defined, 343
  model, 350–356
  transaction processing support, 350–356
  See also cooperation
explorative prototyping, 483
exploratory prototypes, 487
extension levels, 435–436
  core system with, 436
  defined, 435
  example, 436
  See also core system
eXtreme Programming, 4, 416, 449
  comparison, 455
  for detailed iteration planning, 450–451
  story cards, 450
  system visions and, 482
  T&M and, 454
  uses, 454

factory design pattern, 240
factory guiding metaphor, 62–63
factory method, 329
feedback between interaction forms and IP pattern, 232–236
  defined, 188
  example, 234
  illustrated, 232
  intent, 233
  problem, 233
  rationale, 236
  relationship, 233
  solution, 233–234
  trade-offs, 234–235
  See also design patterns
feedback between tool parts pattern, 212–220
  background, 213
  defined, 188
  illustrated, 212
  intent, 212
  problem, 212–213
  relationship, 213
  solution, 213
  See also design patterns
feedback mechanisms
  building, 213
  distinguishing, 213
  implementing, 233
feedback problem
  combination tool, 213
  defined, 212–213
  between FP and IP, 224
file logbook, 359
fixed-price projects, 408
Fleet Management System project, 11
flow organization, 284
flyweight pattern, 240–241
form pattern, 164–167
  defined, 138
  example, 165–166
  illustrated, 164
  intent, 164
  problem, 164
  rationale, 167
  relationship, 164
  solution, 164–165
  trade-offs, 166
  See also conceptual patterns
form system pattern, 252–258
  defined, 189
  illustrated, 252
  intent, 253
  problem, 253
  rationale, 258
  relationship, 253
  solution, 253–254
  trade-offs, 255–256
  See also design patterns
formal reviews, 415–416
  defined, 415
  rules, 415–416
  See also reviews
forms
  classes, 253
  consistency conditions, 165, 257
  developing, 246–247
  generic modeling, 165
  hierarchy, 254
  as independent conceptual patterns, 164
  JWAM framework, 278
  layout, 165, 257
  materials and, 166
  paper, 164

problems, 255–256
tools, 165, 257
use support, 253
using, 255
forms editor, 329, 330
case statement, 329
specialized, 330
frameworks, 2–3, 90–96
application, 91–92
background, 80
black-box, 92–94
characteristics, 91
components and, 98–99
connection, 296
connectors between, 94
defined, 79, 91
design patterns and, 85
directories for, 315
JWAM, 95–96
patterns and, 91
using, 410
white-box, 92–94
functional part (FP), 203, 204, 226–227
buffers, 260
combining with IP, 222
defined, 222, 226
design decisions, 226–227
rationale, 227
separating, 227
as starting point, 222
task separation, 226
tool division, 222
functional prototypes, 484
functional workplace type, 74–75
characteristics, 75
defined, 74
identification, 74
See also workplace types
functional workplaces, 370
automatons for, 168
defined, 370
See also workplaces
functionality
domain, 143, 265
interaction, 214
simple tools, 206
tools, 146–147
functions, 25

generalization, 473, 474
generic operations, 145
defined, 27
at internal interface, 28
in T&M, 27–28
uses, 27
generic persistence service, 369–373
architecture, 373
background, 369–370
data consistency, 371
high performance/throughput requirements, 372
material handling and, 371
problem, 369
software registry vs., 372
solution, 370
trade-offs, 371–373
See also persistence
glossaries, 475–479
common, 477
content, 476–477
defined, 7, 112, 463, 475
in development process, 477–478
document types and, 478–479
elements, 477
entries, 476
example, 477
project-specific, 477–478
purpose, 476
structure, 476
UML and, 479
UP and, 479
See also document types
goals (project), 430–431
example, 430–431
formulating, 430
project stages, 446
graphic user interfaces (GUIs), 103, 155, 221–222
designing, 228–229
widgets, 232
Graphic Workflow Editor project, 11
gray-box tests, 421
groupware, 342
guiding metaphors, 59–63
application software, 59–63
back-office, 5
defined, 5, 59
direct manipulation, 61
electronic commerce frontend workplace, 76
expert workplace, 5, 64–66
factory, 62–63
functional workplace, 5
functions, 59
"granular," 6
object worlds, 60–61
requirements, 64
in software development, 59, 60

handling and presentation, 129–130
defined, 129
example, 130
as independent context, 130
See also software development contexts
handling and presentation layer, 96, 310–311, 366
defined, 310–311
implementing, 311
HTML clients, 265, 266
human computer interaction (HCI), 58, 59

identified requirements, 114–115, 463, 470
identifiers, 375–376
defined, 21
dynamic, 22
object (OIDs), 376
static, 22
trade-offs, 376
identity
defined, 24
domain-specific, 24
software-specific, 24
imperative variable concept, 48
implementation
components, 97
model, 124
specifications and, 35
implicit cooperation, 345–346
characteristics, 345
cooperation material and, 346
defined, 343
transparency, 345
See also cooperation
incoming interface, 22
information hiding, 38, 420
defined, 24
principle, 420

inheritance
  aspects and, 194, 195
  for common properties
    abstraction, 31
  concept, 20
  domain model, 30
  for hierarchy of terms, 31
  for incremental modifications, 31
  layered architecture and, 301
  multiple, 30, 317
  object-oriented layer
    architectures, 301, 309
  relationship, 18, 30
  role hierarchies and, 323
  single, 30
  software model, 30
  to specialize customers for
    product domain, 317
  structure-oriented hierarchies, 32
  in T&M design, 31–32
  testing and, 420
  three-tier architecture and, 307
  using, 30–31, 421
insourcing, 407
instances
  subtype, 42
  type, 44
instruction-function-predicate
  pattern, 86
instructions, 25
integration tests, 421, 422
interaction diagrams, 498
interaction forms (IAFs)
  defined, 230
  domain values, 231, 240
  examples, 230
  JWAM framework, 278
  separating, 232
  set of, 231
interaction planning, 449–451
interactive application software, 2,
  5, 104
interactive application systems,
  357–392
  background, 357–358
  persistence services and, 358–373
interactive part (IP), 203, 204,
  225–226
  combining with FP, 222
  defined, 222, 225
  FP operation identification, 233
  GUI design of, 228–229
  main task, 226
  rationale, 226
  separating, 227
  task separation, 226
  tool division, 222
interfaces, 24
  aspect, 193
  component model, 193
  explicit, 38
  few, 38
  incoming, 22
  internal, 22, 26, 28
  outgoing, 22
  private, 27
  public, 22, 26
  restricting, 30
  small, 38
  structuring, 25
  syntax specification, 41
internal interface
  defined, 26
  generic operations at, 28
  See also interfaces
interrelation of tools and materials
  pattern, 82, 83, 84, 137–138,
  139–142
  background, 140
  defined, 137–138
  example, 140
  illustrated, 139
  intent, 139
  problem, 139
  rationale, 142
  relationship, 139
  solution, 140
  See also conceptual patterns
interviewers, 468–469
interviews, 467–472
  assessment, 471
  characteristics, 468
  general principles, 468
  guideline, 468
  interviewers, 468–469
  moderators, 469
  note-takers, 469
  observers, 469
  open conversation, 470
  process, 468–472
  purpose, 467
  qualitative, 467
  questions to be clarified, 469
  role-playing, 470
  structure, 468
  targeted questions, 471
  user feedback on, 471–472
  See also document types
inventory directory, 359
IT Department Reorganization
  project, 11
IT organization development,
  409–412
  architecture group, 411–412
  product planners, 411
  project teams, 411
iteration planning, 449–451
  plan types, 449–450
  XP for, 450–451
iterations, 448–449
  content and, 449
  defined, 448
  length, 449
  of medium-sized projects, 449
  as miniprojects, 448
  scope of, 448
  UP, 451, 452

Java
  types and, 44
  using, 219
JWAM framework, 95–96, 99
  component-like tools
    construction, 278
  defined, 95
  design patterns for, 190, 271–279
  domain services, 279
  domain values, 274–277
  domain-specific application layer,
    95
  forms, 278
  handling and presentation layer,
    96, 366
  IAFs, 278
  language extensions, 96
  materials construction, 271–272
  PFs, 278
  software registry architecture,
    365–369
  structure illustration, 95
  technology layer, 96
  tools construction, 272–274
  work environment, 279

See also frameworks
JWAM projects, 383–386
  experiences from, 386
  OODBs, 385
  RDBMS mapping, 384–385
JWAM registry, 174

language extensions, 309–310
  defined, 309
  elements, 310
languages
  class-based, 19
  functional, 238
  information hiding, 24
  object-based, 19
  object-oriented, 19–20, 238
  objects in, 48–49
  types and, 44–45
  values in, 48–49
large projects, 429
large-scale application software, 2
late binding, 23
layered architectures
  build rules, 312–313
  example, 313–314
  object-oriented, 299–303
  protocol-based, 297–299
layers
  combining, with bridge pattern, 305
  concept, 303–305
  direct combination of, 304–305
  directories for, 315
  interfaces/relationships and, 303
  model architecture, 303
  open-closed principle implementation with, 302–303
  transparency of, 304
legacy databases, 382–383
Lehman software classification, 402–405
  applying, 405
  background, 402–405
  defined, 402
  E programs, 403–405
  P programs, 403, 404
  S programs, 402–403
load-on-demand, 379–381
  example, 380–381
  problems, 380
  trade-offs, 380–381
  using, 379
loan applications example, 351–353
"look and feel," 118
loose coupling, 35–36
  achieving, 233
  defined, 36
  object creation and, 36

macrostructures, 285
  defined, 125
  identifying, 283
  modeling units and, 293
  structural similarity and, 125
mailboxes, 346–348
  common, 347
  construction approach, 348
  cooperation model, 347
  example, 348
  personalized, 348
  unrestricted, benefit of, 347
  See also cooperation medium
mailing systems, 349–350
  construction approach, 350
  cooperation model, 349
  dispatch folders and, 355
  example, 350
  to support cooperation, 349
  See also cooperation medium
management activities
  connection between, 424, 426
  interplay of, 426
management process, 425–428
  contexts of, 427–428
  continuous feedback, 399
  defined, 395
  development and, 398
  fundamental activities, 425–426
management styles, 425
Management System Redesign project, 13
material design pattern, 142–145
  defined, 138
  domain functionality focus, 143
  example, 143–144
  generic operations, 145
  illustrated, 142
  intent, 142
  problem, 142–143
  rationale, 145
  relationship, 143
  solution, 143
  trade-offs, 144–145
  See also conceptual patterns
materials
  adapting for use contexts, 199
  aspects and, 194
  binding tools to, 191
  collections, 174
  conceptual unity, 143
  construction, 271–272
  copies, 344
  cultural history, 66
  defined, 6, 68
  as design metaphor, 68
  in environments, 269
  exclusive access to, 361–362
  in expert workplace guiding metaphor, 65
  forms and, 166
  interaction and, 144
  loading, 247–248
  loading, from containers, 248
  modeling, 145
  notion of place and time, 344
  objects and, 141
  original, 344
  references between, 380
  relationships between, 143
  similarity of, 140
  software, 68–69
  state, 145
  in T&M design, 68–69
  tables of contents, implementing, 249
  tools and, 149–150, 193
  transport, 162
MedIS, 386
medium-sized projects
  defined, 429
  iterations of, 449
  See also projects
message brokers
  exchanging information through, 368
  implementing, 368
  runtime architecture using, 369
metaclasses, 52
metalevel architectures, 52
metaobject protocols (MOPs), 50–55
  categories, 51

metaobject protocols (MOPs), (*continued*)
  defined, 51
  introduction, 50–51
  motivation for, 51
  operations list support, 53
  representing application model, 52–53
  representing runtime system, 53–55
metaphors
  defined, 63
  design, 5, 6, 63–64
  generic, 63
  guiding, 5–6, 59–63
  patterns and, 66–67
microelements, 125
"middleware," 130
milestone documents, 113
minimal core system, 434–435
  defined, 435
  example, 435
  with extension levels, 436
  specification questions, 435
  See also core system
model architecture
  criteria, 308
  defined, 282
  defining, 295
  layers, 303
  prerequisites, 288
  technology implementation, 314
model-view-controller (MVC), 203
moderators, 469
modularization, 36–38, 285
  criteria, 37
  object-oriented, 37–38
  rules, 38
  use relationships and, 33
modules, classes and, 36
monolithic tools, 274
multiple inheritance
  defined, 30
  use of, 31
  See also inheritance
multiprocess space
  adjusting tools/automatons in, 260–261
  defined, 260

note-takers, 469
object adapters
  for aspect implementation, 198–200
  benefits, 199
  drawbacks, 199
  use recommendation, 200
object behavior analysis (OBA), 494
object diagrams, 497
object identifiers (OIDs), 376
  as common key, 377
  SyLab system, 388–389
object life cycle, 28–29
  creation, 28
  defined, 28
  deletion, 28
  T&M design and, 29
  transformation, 28–29
object metamodels, 17–55
  classes, 18, 25–27
  defined, 18
  descriptions, 18
  domain model, 20
  elements, 282
  introduction, 17
  purpose, 18–19
  software model and, 20
  values in, 48
object worlds guiding metaphor, 60–61
object-based languages, 19
object-oriented database (OODB) systems, 376–377
  load-on-demand, 379–381
  problems with, 385
  using, 385
object-oriented design, 37
object-oriented languages, 19–20
object-oriented layer architectures, 299–303
  defined, 299–300
  elements, 300
  example, 300
  inheritance and, 301, 309
  layers, 299–300
  open-closed principle and, 300
object-oriented models, 57
Object-Oriented Programming Environment project, 10
objects, 20–24
  characteristics of, 46–47
  command, 235
  composition/containment relationship, 473, 474
  core, 317, 318, 320
  creating, 28
  current work, 157
  defined, 21, 46
  deleting, 28, 219
  dependency relationship, 473
  with different technical identities, 316
  domain, mapping to model concepts, 121
  in domain model, 21
  domain value, 276
  identifiers, 21
  identity, 24–25, 375
  immutable domain value, 240–241
  interface, 22, 24
  materials and, 141
  mutable, 242–243
  organization, 25
  persistent, 54
  in programming languages, 48–49
  protocols, 23
  register, 337
  request, 391
  role, 317, 318, 320
  runtime creation, 21
  services, 22, 23
  in software model, 21
  state, 22
  structures, 379
  transforming, 28–29
  types, 21, 40–45
  values and, 45–50
observers, 469
OO testing, 419–424
  characteristics, 419–421
  complexity and dependencies, 420
  encapsulation, 419
  information hiding, 420
  inheritance, 420
  polymorphism, 421
  process, 421–424
open conversation, 470
open-close principle, 38
  class libraries and, 90
  defined, 37

example, 301–302
implementing, with layers, 302–303
layered architecture and, 300–302
operation calls, 54
operations
abstract, 34
activating, 21
base, 35
calling, 23
declaration, 22
defined, 23
of domain services, 267
generic, 145
inherited, 30
names, passing, 234
on domain values, 50
redirecting, 54
signature, 21
order principles, 157
organization structures
by business sectors, 284–285
company, 284
flow, 284
as macro elements, 284
object principle and, 284–285
organizational context, 394
organizational integration, 407
outgoing interface, 22
overview scenarios, 464–465
actors, 465
defined, 464
tasks, 465
See also scenarios
overviews, 489, 490

P programs (Lehman classification), 403, 404, 406
author-critic cycle and, 405
defined, 403
elements, identifying, 406
evaluation, 405
illustrated, 404
pair programming, 407, 416–418
advantages, 417–418
collective ownership, 418
defined, 416
example session, 417
potential, 417–418
roles, 416
for team training, 417
See also quality assurance
pattern collections, 89–90
pattern form, 88
background, 88
context, 88
defined, 81
problem, 88
solution, 88
in T&M design, 88
patterns, 80–90
with abstract classes, 35
adjusting tool, 181–183
aspect, 187, 190–201
automatons, 138, 167–171
automatons in embedded systems, 189, 258–264
background, 80
body/handle, 242
chain of responsibility, 82
characteristics, 81–83
conceptual, 3, 83–84, 135–183
container, 138, 159–163
context, 82
coupling tools and materials, 85
coverage, 82
design, 3, 84–85, 86, 185–279
domain container, 244–252
domain service provider, 138–139, 171–176
domain services, 189, 264–268
domain values, 188, 236–243
elements, 81
environment, 189, 268–271
as experience-based concept, 81
factory design, 240
feedback between interaction forms and IP, 188, 232–236
feedback between tool parts, 188, 212–220
flyweight, 240
form, 138, 164–167
form system, 189, 252–258
frameworks and, 91
handling guidelines, 87
as hot issue, 80
instruction-function-predicate, 86
interrelation of tools and materials, 82, 83, 84, 137–138, 139–142
material design, 138, 142–145
metaphors and, 66–67
probe, 179–181
programming, 85–86
purpose of, 81
separating FP and IP, 221–227
separating function and interaction, 187, 202–204
separating handling and presentation, 188, 227–232
singleton, 279
T&M roadmap, 89
taxonomy, 83
technical automaton, 139, 176–179
tool composition, 187–188, 204–212
tool design, 138, 146–151
understandable descriptions, 82
work environment, 138, 152–159
persistence
architectural concepts, 358
distributing, 374–375
generic service, 369–373
implementation design criteria, 373–383
services, 358–373
software registry, 358–369
technical, concepts, 369
workplace types and, 357
persistence automation
interface, 372
polymorphic queries implementation, 389
sample code, 370–371
persistence-capable clients, 374–375
persistence-capable servers, 375
persistent objects, 54
personalized mailboxes, 348
pictograms, 488
pilot systems
content, 485
defined, 484
See also prototypes
planning pictures, 490
pointer swizzling, 55
polymorphic containers, 161
polymorphic queries, 389
number of, 390
relational joint operations for, 391

polymorphism, 33–34, 201
  defined, 33
  dynamic binding and, 34
  for object binding, 304
  testing and, 421
  unrestricted, 33
  use of, 421
  See also inheritance
predicates, 25
presentation forms (PF)
  defined, 230
  domain values, 240
  JWAM framework, 278
  separating, 232
presentation prototypes, 484
private interface, 27
probe pattern, 179–181
  background, 180
  illustrated, 179
  problem, 179–180
  rationale, 181
  relationship, 180
  solution, 180
  See also conceptual patterns
probes
  adjusting tools with, 183
  attachment, 180
  defined, 180
  design, 180
  technical automatons and, 181, 259
  uses, 180–181
probing states, 67
procedures, 25
process models
  application orientation, 394–395
  background, 393–399
  context, 394
  evolutionary, 398–399
  "how-to" rules, 393
  spiral, 397
  waterfall, 395–396
process patterns, 353–354
  defined, 353
  for loan application cases, 354
  normal case pattern, 353
process-controlled view, 63
processes
  containers and, 72
  technical, representing, 178
  tools and, 148–150
Product Design System and Contract Management project, 12–13
product domains, 287–289
  constructors in, 335
  defined, 288
  example, 288
  inheritance for specializing customers for, 317
  layer, 312
  as logical architecture basis, 289
  modeling, 289
  problematic coupling, 292
  use context and, 288–289
  workplace types and, 294
product planners, 411
product trader pattern, 327–338
  background, 328
  benefits, 334
  creators, 336
  drawbacks, 334–336
  error-prone code, 334
  example, 328–330, 331
  implementation, 336–337
  intent, 327
  interactions, 333
  object comparison, 333
  participants, 332–333
  problem, 327–328
  rationale, 338
  sample code, 337–338
  solution, 330–331
  specification ambiguity, 335
  specifications, 336
  structure, 332
  structure complexity, 334
  trade-offs, 334–336
  See also design patterns
product traders
  allocation specifications management, 336
  context-specific, 334
  use situations, 338
products, class hierarchy, 334
program concatenation, 30
programming
  components, 97
  eXtreme, 4, 416, 449, 450–451, 482
  guidelines, 114–115
  pair, 407, 416–418
programming patterns, 85–86
  defined, 83, 85
  example, 85
  instruction-function-predicate, 86
  languages and, 85
  See also patterns
project contracts, 113–114
  defined, 113
  elements, 113–114
project control, 425–426
  documents, 114
  importance, 425–426
project documents, 113–114
  identified requirements, 114–115
  project contract, 113–114
  project control, 114
project management, 425–428
project planning
  backward, 439
  base lines for, 441–443
  calibration, 428–430
  decision principles, 431
  defined, 425
  establishment, 432
  goals, 430–431
  stages, 438–441
  team, 440
project profiles, 446–448
  adaptation rules, 447–448
  characteristics, 446
  defined, 446
project stage plans, 446–448
  contents, 446
  defined, 446
  project profiles and, 446–448
  resources distribution, 447
project stages, 448
  backward planning and, 439
  conference planning and, 440–441
  defined, 438, 444
  evaluating, 439
  example, 440
  finishing, 439
  goals, 446
  planning, 438–441
  project-specific objectives, 448
  size, 439
  T&M project, 445
project steering, 425

Zull-Ind.qxd 31/8/04 2:40 PM Page 515

project teams, 411
projects
  calibration, 428–430
  classification of, 429
  establishment, 432
  goals, 430–431
  large, 429
  medium-sized, 429, 449
  small, 429, 449
  strategic importance, 429
  structuring, by system decomposition, 432–436
  team experience, 429
protocol-based layer architectures, 297–299
  characteristics, 298–299
  defined, 298
  hierarchy, 298
  layers, 298
  three-tier, 297–298
protocols, 23
prototypes, 414, 483–487
  address problems, 414
  breadboards, 484
  categories, 484
  content, 484–485
  defined, 7, 483
  in development process, 485–487
  document types and, 487
  evaluating, 415, 486
  evolutionary, 487
  exploratory, 487
  functional, 484
  number of, 487
  pilot system, 484
  presentation, 484
  purpose, 483–484
  scenarios and, 464
  scope of, 487
  screenplays, 486–487
  structure, 484
  T-prototype, 485
  UP and, 487
  workshop visits, 486
  See also document types
prototyping, 414
  benefits, 485
  evolutionary, 484
  experimental, 483
  explorative, 483
  types of, 483–484
public interface, 22, 26
purpose tables, 494–496
  content, 494–495
  defined, 494
  in development process, 495–496
  document types and, 495
  example, 495
  purpose, 494
  structure, 494
  UML and, 496
  See also document types

qualitative interviews, 467
quality assurance, 413–424
  in construction, 419–424
  in development process, 413–419
  direct user integration, 413
  OO testing, 419–424
  pair programming, 416–418
  prototyping, 414
  refactoring, 418–419
  reviews, 414–416
queries
  operative data, 378
  polymorphic, 389, 390, 391

reaction mechanisms, tool component with, 219–220
Refactoring: Improving the Design of Existing Code, 419
refactoring, 418–419
  benefits, 418
  defined, 418
  historical use of, 419
  time pressures and, 418
  See also quality assurance
reference semantics, 33
referential transparency, 48
register objects, 337
registering, 261
registrar
  defined, 359
  interface, 367
  registration means, 359
  software registry use, 360
registration means, 359, 360
registry. *See* software registry
regression tests, 422
relational database management systems (RDBMS), 377
  BLOBs in, 385
  mapping, 384–385
  structures, 384
relational databases, inheritance in, 377–378
requests
  defined, 25
  objects, 391
reservation list, 359
Reverse Auctions Application Redesign project, 12
reviews, 414–416
  formal, 415–416
  informal, 414
  road shows, 415
  user work groups, 415
  See also quality assurance
road shows, 415
role concepts, 318, 321
role objects, 317
  defined, 318
  dynamically adding, 321
  illustrated, 319
  implementing, 318
  interaction, 320
  interfaces, 321
  managing, 321
  use relationship, 322
role pattern, 315–327
  background, 318
  benefits, 320–321
  coding, 321
  drawbacks, 321
  illustrated, 320
  implementation, 321–323
  intent, 315–316
  interactions, 320
  participants, 319
  rationale, 327
  recursive application of, 323
  role dependencies and, 327
  sample codes, 323–327
  trade-offs, 320–321
  See also design patterns
role relationships, 32
roles
  client request of, 320
  of core concept, 327
  hierarchies, 323
  implementing, 322
  roles of, 322
  roles of, object diagram, 324

roles (*continued*)
  strong dependencies between, 327
root concept, 322
routing slips, 354–355
  characteristics, 354–355
  construction approach, 355–356
  cooperation model, 354–355
  designing, 355–356
  modifying, 355
  using, 355
  See also cooperation medium
routinized cooperation, 353
runtime components, 97
runtime system, 53–54
Runtime Type Information (RTTI), 52

S programs (Lehman classification), 402–403
  author-critic cycle and, 405
  defined, 402
  elements, identifying, 406
  evaluation, 405
  illustrated, 403
scenarios, 459–467
  activity, 466
  content, 460–461
  defined, 7, 112, 459
  description, 461
  in development process, 462–464
  document types and, 463–464
  elements, 462
  example, 461
  number of, 463
  overview, 464–465
  as part of domain model, 460
  prototypes and, 464
  purpose, 460
  scope of, 463
  structure, 460
  subtypes of, 464–466
  system tests and, 464
  task, 465
  titles, 460
  UML and, 466–467
  writing, 460
  See also document types
scheduling book, 359
separating FP and IP pattern, 221–227
  background, 223–224
  defined, 221
  example, 222–223
  illustrated, 221
  intent, 221
  problem, 221–222
  rationale, 225
  relationship, 222
  solution, 222
  trade-offs, 224–225
  See also design patterns
separating function and interaction pattern, 202–204
  background, 203
  defined, 187
  illustrated, 202
  intent, 202
  problem, 202
  rationale, 204
  relationship, 202
  solution, 203
  trade-offs, 203–204
  See also design patterns
separating handling and presentation pattern, 227–232
  background, 228–229
  defined, 188
  example, 231
  illustrated, 227
  intent, 228
  problem, 228
  rationale, 232
  relationship, 228
  solution, 230–231
  trade-offs, 231–232
  See also design patterns
service architectures, 267
services
  call of, 23
  of containers, 160
  defined, 22
simple tools, 206
single inheritance
  defined, 30
  use of, 31
  See also inheritance
singleton pattern, 279
small automatons, 71
small projects
  defined, 429
  iterations and, 449
Smalltalk
  development environment, 407
  types and, 45
sniffers, 364
software architects, 412
software architecture
  combining elements rules, 296–297
  components, 296
  connectors, 296
  contexts influence, 130–132
  design, 284
  domain core, 283–295
  elements, 282, 296
software construction, 31
software development, 2
  application domain and, 127
  application-oriented, 2
  author-critic cycle, 107–108
  as communication/learning process, 106
  defined, 105
  dilemma of, 402
  documentation, 108–109
  document-driven, 106
  evolutionary system development, 108
  as modeling process, 117–132
  See also application-oriented software development
software development contexts, 127–130
  application domain, 127–128
  applied technique, 128–129
  handling and presentation, 129–130
  influences between, examples, 132
  models and, 131
  as semi-autonomous dimensions, 131
  software architecture influence, 130–132
  types of, 127
software development model
  application domain and, 119–120
  application system model and, 122–125
  contexts, 118
  descriptive, 118
  domain model and, 119–122

illustrated, 118
paradox, 122
simplified, 117–118
software factory. *See* factory guiding metaphor
software model
classes, 26
inheritance, 30
objects, 21
structures, 308
software projects
allocation of, 409
application-oriented, 410
characteristics, 394
dimensions, 395
fixed-price, 408
goals, 394
software registry, 358–369
architecture for, 365–369
background, 359, 360
based on cooperation model, 363
characteristics, 360
classes for, 367
combining with persistence service, 374
comparison model, 361–363
concept, 360
domain usage model, 358
elements, 359
generic persistence service vs., 372
with originals/copies, 361
problem, 358, 360
purposes, 360
solution, 360
usage model, 364–365
using, 382
software system. *See* application system
software-specific identity, 24
special-purpose systems
alternative, 433
core system with, 434
defined, 433
example, 433–434
specifications
implementation and, 35
product trader pattern, 336
spiral model, 397
state modeling, 423–424
defined, 423–424
explicit implementation, 424
illustrated, 424
statechart diagrams, 498
states
altering, 67
defined, 22
material, 145
probing, 67
representation, 52
tool, 149–150
work, 155–156
static identifiers, 22
static typing, 196–198
story cards, 450, 482
defined, 450
use limitations, 482
using, 450
streams, 243
structural mapping, 289, 377
classes to relations, 377
insert operations and, 390
structural similarity, 122–123
benefits, 104
defined, 104
macrostructures and, 125
structure, this book, 3–4
subclasses
defined, 34
operations implemented in, 420
use of, 35
white-box framework, 93
subfunctionalities, 209, 210, 214
context functionality and, 210
interaction, 214
subjects
closed, 318
defined, 318
identity, managing, 323
implementation, 322
modeling, 319
state integrity, 322
subtools, 205
building, 206
defined, 206
functionality reuse for, 208
subtypes, 42
superclasses
attributes, 389
defined, 34
domain values, 238
form, 278
supportive view, 65–66, 73
characteristics, 65–66
defined, 65
SyLab
background, 386–387
class mapping, 389–392
client-server architecture, 387
defined, 386
identifiers, 388–389
material structure, 389
structural mapping, 389
tasks covered by, 386
technical data modeling, 389
workplaces, 387
system base, 128–129
system base layer, 310
system decomposition, 433
system tests, 464
system visions, 479–483
for complex systems, 480
decisions for using, 481
defined, 7, 112, 479
in development process, 480–482
document types and, 481–482
elements, 481
example, 480
purpose, 479
story cards and, 482
structure, 480
technical, 480
as tools, 481
UML and, 482–483
variants, 479
XP and, 482
See also document types

T&M model architecture, 281–338
background, 307
build rules, 312–313
business domain layer, 311–312
concepts and elements, 295–314
defined, 282
design patterns, 314–338
document types, 459–499
domain core, 283–295
elements, 308–309
handling and presentation layer, 310–311
language extensions, 309–310
layer concept, 303–305

T&M model architecture, (*continued*)
object-oriented layer architectures, 299–303
problem/solution, 307
product domain layer, 312
product trader, 327–338
protocol-based layer architectures, 297–299
role pattern, 315–327
software architecture components, 296
system base layer, 310
technology implementation, 314
technology layer, 310
user context layer, 312
T&M process model
comparison, 455
at conceptual level, 452
structural similarity principle, 452
UP development process vs., 451–452
uses, 454
workflows and, 453
XP and, 454
T&M projects, 9–15
Certified Aircraft Engine Software, 11–12
Cooperative Workplace System, 13
Embedded Medical Lab System, 12
Fleet Management System, 11
Graphic Workflow Editor, 11
IT Department Reorganization, 11
Management System Redesign, 13
Object-Oriented Programming Environment, 10
planning, 443–451
Product Design System and Contract Management, 12–13
Reverse Auctions Application Redesign, 12
scope, 10–14
Workplace Frontend, 14
Workplace System for Retail Banking, 10
`TableOfContents` class interfaces, 277–278
tables of contents, 248–250
implementing, as domain values, 249–250
implementing, as materials, 249
See also containers
targeted questions, 471
task cards, 450
task pictures, 489, 491
task scenarios, 465
tasks
application orientation and, 102–103
defined, 103
overview scenarios, 465
tool support, 148–150
team planning, 440
team training, 417
technical automaton pattern, 176–179
background, 178–179
defined, 139
illustrated, 176
intent, 177
problem, 177
relationship, 177
See also conceptual patterns
technical automatons
adjusting tools and, 259
complex, 179
concept, 178
defined, 177
example, 179
parameters, modifying, 183
probes and, 181, 259
See also automatons
technical containers, 163
to domain container implementation, 246–247
illustrated, 247
technical context, 394
technical documents, 112, 496–499
activity diagrams, 498
application-oriented documents and, 499
class diagrams, 473, 497
component diagrams, 498
deployment diagrams, 498
interaction diagrams, 498
object diagrams, 497
opinion, 497
statechart diagrams, 498
use case diagrams, 467, 483, 498
technical integration, 409
technical transparency, 344, 344–345
technology
defined, 129
domain, 427–428
selection, 428
technology layer, 310
template methods, 34
testing
classes, 421–422
OO, 419–424
tests
black-box, 422
cases, 423
classes, 423
defined, 25
gray-box, 421
integration, 421, 422
regression, 422
unit, 421
three-tier architecture, 283, 297–299, 305–307
defined, 297
for distributed systems, 299
domain separation, 305
drawbacks, 306–307
example, 297–298
illustrated, 297
inheritance and, 307
specialized, 299
See also protocol-based layer architectures
time estimates
general rules, 437–438
problems, 438
rules of thumb, 437–438
tool composition pattern, 204–212
background, 206
defined, 187–188
example, 206–207
illustrated, 204
intent, 205
problem, 205
rationale, 208
relationship, 205
solution, 205
trade-offs, 207–208
See also design patterns
tool design pattern, 146–152
defined, 138
design characteristics and, 150–151

example, 147–148
illustrated, 146
intent, 146
problem, 146
rationale, 151
relationship, 146
solution, 146–147
trade-offs, 148–150
See also conceptual patterns
toolkits
defined, 228
encapsulating, 228
library, 229
linking tools to, 229
tools, 140
adjusting, 182–183
asynchronous communication, 262
automatons vs., 169
binding, to materials, 191
boundaries identification, 210–212
building, by composition, 205
building, JWAM framework, 272–274
building, with components, 208–209
CASE, 8, 127
combination, 206, 209–210
combination, building, 209–210
component model, 219
components, with reaction mechanisms, 219–220
conceptual division, 203
conceptual properties, 67
context, 206, 210
cultural history, 66
defined, 6, 67
design alternatives, 207
design characteristics, 150
design guidelines, 203
as design metaphors, 67
designing, 150
in environments, 269
in expert workplace guiding metaphor, 65
form, 165, 257
function granularity, 147
functional part (FP), 203, 204
functionality, 146–147
graphical representation, 150
handling, 67
handling and presentation, 202
handling modes, 150
implementing, 203
initializing, 151
interactive, responsibilities, 203
interactive part (IP), 203, 204
linking, 207
linking, to toolkit, 229
materials and, 149–150, 193
matured, 141
memory, 150
monolithic, 274
name, 150
processes and, 148–150
purposes, 67
as reactive systems, 223–224
schematic, 151
separate interaction and functionality, 273
simple, 206
size, 149–150
software, 67–68
state, 149
status indicator, 150
subtools, 205, 206
in T&M design, 67–68
task support, 148–150
understanding, 141
user interface, 223
Tools & Materials Approach (T&M), 1, 4–9
automatons in, 71
central architectural theme, 5
classes and types in, 45
conceptual patterns, 135–183
construction overview, 275
containers in, 72
contract model in, 40
design, 25
design patterns, 86, 185–279
document types, 111–113
domain values in, 49
expert workplace metaphor, 64–66
generic operations in, 27–28
inheritance in, 31–32
as method, 8–9
as methodical framework, 9
object life cycle and, 29
object metamodel, 17–55
overview, 4–8
pattern form, 88
pattern roadmap, 89
pattern taxonomy, 83
relationship to UML/UP, 9
software materials in, 68–69
software tools in, 67–68
work environment in, 70
See also T&M model architecture; T&M projects
T-prototype, 485
transactions
business, 175
defined, 381
processing support, 350–356
transforming objects, 28–29
transition stage (UP), 446
transparency
application-oriented, 344–345
in implicit cooperation, 345
technical, 344
types, 40–43
behavior and, 41
class differences, 43–44
classes and, 43–45
defined, 41
hierarchy, 42
instances behavior, 44
introduction, 40
programming languages and, 44–45
properties, 42
subtypes, 42
theoretical concept of, 41–42
See also objects

understandability, 37
Unified Modeling Language (UML), 3, 4
activity diagrams, 498
class diagrams, 473, 497
component diagrams, 498
concept models and, 475
cooperation pictures and, 494
deployment diagrams, 498
documents in, 110
glossaries and, 479
interaction diagrams, 498
object diagrams, 497
purpose tables and, 496
scenarios and, 466–467
statechart diagrams, 498
system visions and, 482–483
T&M relationship, 9
technical document types in, 496–499

Unified Modeling Language (UML), (*continued*)
  use case diagrams, 112, 467, 483, 498
Unified Process (UP), 3
  actors, 465
  base lines in, 442
  comparison, 455
  conceptual stage, 445
  construction stage, 446
  cycles, 451
  design stage, 445–446
  development process structure, 451–454
  evolutionary cycles, 443–444
  glossaries and, 479
  interpreting, 444–445
  iterations, 451, 452
  phases, 445–446, 451
  project planning, 443–451
  prototypes and, 487
  T&M relationship, 9
  transition stage, 446
  workflows, 451
unit tests, 421
usage model, 123
  defined, 58, 103, 123
  distributed communication, 263
  dynamics, 123
  implementation, 129
usage model (registry), 364–365
  background, 364
  cooperation model with, 365
  problem, 364
  solution, 364
usage quality, 104
use case diagrams, 112, 467, 483, 498
use cases, 482
use contexts, 285–287
  defined, 285
  as domain analysis starting point, 286
  example, 286
  identifying, 287
  layer, 312
  product domain and, 288–289
  workplace types and, 286–287, 293
use relationships, 32–33, 144
  defined, 32
  modularization and, 33
  in software model, 32
user feedback, 471–472
user interface toolkits. *See* toolkits
user observation, 462
user work groups, 415
user-defined classes, 49

values, 45–50
  characteristics of, 46–47
  defined, 46
  domain, 45, 49–50
  in object metamodel, 48
  objects vs., 46–47
  in programming languages, 48–49
  using, 47–48
variables
  defined, 48
  instance, storage manipulation, 54
Visual Basic Extension (VBX), 96

waterfall model, 395–396
  defined, 395
  illustrated, 396
  weaknesses, 396
  See also process models
white-box frameworks, 92–94
  black-box frameworks comparison, 93
  defined, 92
  subclasses, 93
  using, 92–93
  See also black-box frameworks; frameworks
widgets
  defined, 228
  GUI, 232
  linking, 228
  use of, 231
work contexts
  defined, 156
  realization, 285–286
work environment pattern, 152–159
  background, 154–155
  defined, 138
  example, 153–154
  illustrated, 152
  intent, 152
  problem, 152
  rationale, 159
  relationship, 152
  solution, 152–153
  trade-offs, 155–159
  See also conceptual patterns
work environments, 69–70
  in application domain, 156
  as conceptual unit, 154
  consistency checks, 157
  as cooperation location, 154
  defined, 6, 69
  as design metaphor, 69–70
  electronic desktop, 155, 156
  as location, 154
  modeling, 152
  objects, actions for, 156
  order principles, 157
  personalizing, 156
  representation problems, 154–155
  requirements, 153
  space implementation, 155
  spatial/logical dimensions, 153
  in T&M design, 70
  use context implementation, 156
  for work organization, 154
work relationships, 141
work state, maintaining, 155–156
work units, 466
workflows, 102
  container representation, 71
  management systems, 63
  T&M models and, 453
  UP, 451, 453
Workplace Frontend project, 14
Workplace System for Retail Banking project, 10
workplace types, 73–77
  back-office, 75–76
  characteristics, 74
  defined, 74
  electronic commerce frontend, 76–77
  example, 286–287
  expert, 74
  functional, 74–75
  identification of, 342
  overview, 73
  persistence and, 357
  product domains and, 294
  use context and, 286–287, 293
workplaces
  defined, 69
  functional, 168, 370
  SyLab, 387
workspace, environment and, 270