



Model-Driven Development with Executable UML

Dragan Milicev



Updates and Wrox technical support at www.wrox.com

Model-Driven Development with Executable UML

Preface	xxi
Part I: Introduction	
Chapter 1: Information Systems Modeling	3
Chapter 2: Traditional Approaches to IS Development	15
Chapter 3: The Object Paradigm	25
Part II: Overview of OOIS UML	
Chapter 4: Getting Started	41
Chapter 5: Basic Language Concepts	49
Chapter 6: Interaction and Querying	97
Part III: Concepts	
Chapter 7: General Concepts	141
Chapter 8: Classes and Data Types	181
Chapter 9: Attributes	247
Chapter 10: Associations	281
Chapter 11: Constraints	353
Chapter 12: Querying	391
Chapter 13: Operations and Methods	423
Chapter 14: State Machines	483
Chapter 15: Collaborations and Interactions	517
Chapter 16: Commands, Presentation, and Architecture	547
Part IV: Method	
Chapter 17: About the Method	593
Chapter 18: Conceptual Modeling	609
Chapter 19: Modeling Functional Requirements	631
Part V: Supplemental	
Chapter 20: Characteristics of Information Systems	663
Chapter 21: Process and Principles of Software Development	685
Chapter 22: The Relational Paradigm	695
Chapter 23: Structured Analysis	727
Chapter 24: Introduction to the Object Paradigm	737
References and Bibliography	749
Index	753

Model-Driven Development with Executable UML

Model-Driven Development with Executable UML

Dragan Milicev



Wiley Publishing, Inc.

Model-Driven Development with Executable UML

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2009 by Dragan Milicev

Published by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-48163-9

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Control Number: 2009927339

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Wrox Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

*To Miloš, Mina, Jovan, and Sneža
Милошу, Мини, Јовану и Снежи*

Credits

Executive Editor

Carol Long

Development Editor

Kevin Shafer

Technical Editor

Bran Selic

Production Editor

Kathleen Wisor

Copy Editor

Nancy Rapoport

Editorial Manager

Mary Beth Wakefield

Production Manager

Tim Tate

Vice President and Executive Group Publisher

Richard Swadley

Vice President and Executive Publisher

Barry Pruett

Associate Publisher

Jim Minatel

Project Coordinator, Cover

Lynsey Stanford

Cover Photo

Goran Nikolasevic,

www.fotostudionikolasevic.co.rs

Proofreader

Publication Services, Inc.

Indexer

Johnna VanHoose Dinse

About the Author

Dragan Milicev, PhD, is an associate professor at the Department of Computer Science at the University of Belgrade, School of Electrical Engineering. He is the founder and CTO of Serbian Object Laboratories d.o.o. (SOL, www.sol.rs), a software development company specializing in building software development tools using model-driven technology, as well as in building custom applications and systems. With 25 years of experience in building complex software systems, he has served as the chief software architect, project manager, or consultant in more than 20 academic and international industrial projects. Of note is the fact that he was Chief Software Architect and Project Manager for most of SOL's projects and all its products: SOLOist, a rapid application model-driven development framework for information systems; SOL UML Visual Debugger, one of the world's first UML visual debuggers, designed for the Poseidon for UML modeling tool; and SOL Java Visual Debugger, a plug-in for Eclipse that enables modeling of test object structures using UML object diagrams. He has published papers in some of the most prestigious scientific and professional journals and magazines, contributing to the theory and practice of model-driven development and UML. He is the author of three previous books on C++, object-oriented programming, and UML, published in Serbia. You may contact him at dmilicev@etf.rs.

Acknowledgments

I would like to express my great debt of gratitude to Bran Selic, one of the pioneers and most respected authorities on model-driven software engineering and key contributors to UML, whose careful and thorough review of the manuscript and invaluable comments helped improve this book's structure and technical accuracy.

Special thanks to my colleagues from SOL who have been involved in the development of SOLoist and its application to many industrial projects. Their contribution to the implementation of many ideas presented in this book, as well as their help in making the concepts indeed pragmatic and effective for industrial systems, is deeply appreciated.

I would like to thank my research assistants, as well as my graduate and undergraduate students from the University of Belgrade, School of Electrical Engineering, Department of Computer Science, who took part in the research and development of some ideas and concepts presented in the book. They also offered many valuable comments on the early drafts of the manuscript.

Many thanks to the companies where I have served as a consultant. I have enjoyed the opportunity to brainstorm with colleagues and discuss many ideas presented in this book, as well as the opportunity to apply the ideas in practice.

The work on the development and implementation of the approach presented in this book is partially supported by the Serbian Ministry of Science, under the national program for technology development, grant TR-13001.

Finally, I would like to express my boundless gratitude for forbearance, understanding, and support to my beloved daughter, Mina, sons Miloš and Jovan, and wife, Snežana, to whom I dedicate this book.

Contents

Preface

xxi

Part I: Introduction

Chapter 1: Information Systems Modeling	3
Definition of Information Systems	3
Models and Modeling Paradigms, Languages, and Tools	5
Modeling	5
Modeling Languages	6
Modeling Tools	9
Modeling Paradigms	9
Processes and Methods	12
Chapter 2: Traditional Approaches to IS Development	15
Characteristics of Traditional Modeling Paradigms	16
Usability Aspects	17
Development Aspects	19
Scope Discontinuities	20
Semantic Discontinuities	21
Development Phase Discontinuities	22
Implications of Discontinuities	22
User-Interface Development Problems	22
Chapter 3: The Object Paradigm	25
Object-Oriented Modeling	25
The Unified Modeling Language	27
Characteristics of UML	28
Profiling UML	30
Traditional OO Development Approach	31
Desired Characteristics of Object-Oriented Information Systems	33

Contents

Usability Aspects	34
Development Aspects	36
The Rest of This Book	37

Part II: Overview of OOIS UML

Chapter 4: Getting Started	41
Key Features of OOIS UML	41
The Organization of OOIS UML	44
Chapter 5: Basic Language Concepts	49
Classes and Attributes	49
Requirements	49
Concepts	50
Interactive Manifestations	56
FAQ	58
Associations	62
Requirements	62
Concepts	62
Interactive Manifestations	66
FAQ	68
Generalization/Specialization Relationships	69
Requirements	69
Concepts	69
Interactive Manifestations	74
FAQ	75
Operations	76
Requirements	76
Concepts	76
Interactive Manifestations	82
FAQ	83
Polymorphism	84
Requirements	84
Concepts	84
Interactive Manifestations	85
FAQ	86
Consistency Rules	86
Requirements	86
Concepts	87
Interactive Manifestations	91
FAQ	93

Chapter 6: Interaction and Querying	97
Customizing Presentation	97
Requirements	97
Concepts	98
Interactive Manifestations	104
FAQ	106
Customizing Behavior	108
Requirements	108
Concepts	109
Interactive Manifestations	124
FAQ	125
Querying	128
Requirements	128
Concepts	128
Interactive Manifestations	135
FAQ	137

Part III: Concepts

Chapter 7: General Concepts	141
The Dichotomies of OOS UML	141
Specification/Realizations and Classifier/Instances Dichotomies	141
Modeling and Execution	142
Compilation and Interpretation	143
Basic and Derived Concepts	144
Formal and Informal Concepts	145
Structure and Behavior	146
Core and Extended Parts	146
Model Elements and Diagrams	147
General Language Concepts	151
Elements and Comments	151
Packages	152
Namespaces and Visibility	155
Dependencies	167
Multiplicity Elements	171
Chapter 8: Classes and Data Types	181
Common Characteristics of Classes and Data Types	181
Notions of Class and Data Type	181
Classes and Data Types as Classifiers	183

Contents

Discriminating Characteristics of Classes and Data Types	185
Identity	185
Features	193
Copy Semantics	197
Lifetime	198
Creation and Destruction of Instances	199
Actions	199
Constructors	206
Creational Object Structures	210
Destructors	234
Propagated Destruction of Objects	236
Data Types	240
Primitive Data Types	241
Enumerations	242
Built-in and User-Defined Data Types	244
Chapter 9: Attributes	247
Attributes as Structural Features	247
Attributes as Multiplicity Typed Elements	248
Static Attributes	250
Read-Only Attributes	252
Frozen Attributes	255
Derived Attributes	255
Redefinition of Attributes	260
Actions on Attributes	263
Read Attribute Actions	264
Write Attribute Actions	268
The Symbol null	274
Freezing and Unfreezing Attributes	275
Iterations on Attributes	276
Access to Slots Through Reflection	277
Implementation in Other Detail-Level Languages	278
Chapter 10: Associations	281
Binary Associations	281
Binary Associations and Links	281
Association Ends and Properties	284
Semantics of Binary Associations and Association Ends	287
Special Characteristics of Association Ends	294
Actions on Binary Associations	319
N-ary Associations	331

Notion of N-ary Association	331
Semantics of N-ary Associations and Association Ends	333
Multiplicity of N-ary Association Ends	335
Specific Rules for N-ary Association Ends	338
Actions on N-ary Associations	339
Conceptual Modeling Issues	340
Association Classes	343
Notion of Association Class	343
Uniqueness of Association Classes	345
Actions on Association Classes	347
Conceptual Modeling Issues	351
Chapter 11: Constraints	353
Constraints as Model Elements or as Objects	353
Constraints as Model Elements in Standard UML	353
Constraints as Model Elements in OOIS UML	357
Constraints as Objects in OOIS UML	366
Object Constraint Language	369
Relation to the UML Model	369
Operators and Expressions	372
Tuples	374
Collections	375
OOIS UML Dialect of OCL	386
Chapter 12: Querying	391
Queries as Model Elements or Objects	391
The Semantics of Queries in OOIS UML	391
Queries as Model Elements in OOIS UML	392
Queries as Objects in OOIS UML	394
Object Query Language	398
Semantics of OQL Queries	399
Navigation in the FROM Clause	402
Selection in the WHERE Clause	405
Projection in the SELECT Clause	407
Ordering and Grouping	409
Unions	410
Parameterization and Nesting	411
Inline OQL Queries	413
Pattern Object Structures	414
Pattern Object Structure Specifications	414
Creating Pattern Object Structures by Demonstration	420

Contents

Chapter 13: Operations and Methods	423
Operations	423
Operations as Behavioral Features	423
Parameters of Operations	426
Operation Invocation	429
Preconditions and Postconditions	440
Methods	444
Methods as Implementations of Operations	444
The OOIS UML Native Detail-Level Language	445
Exceptions and Exception Handling	465
Notion of Exception and Exception Handling	465
Exception Types	467
Throwing and Catching Exceptions	468
Declaring Exceptions Thrown by Operations	470
Concurrency and Fault Tolerance	472
Concurrency Model in OOIS UML	472
Concurrency Control	474
Fault Tolerance and Transactions	477
Chapter 14: State Machines	483
Introduction to State Machines	483
Motivation	483
State Machines, States, and Transitions	487
Guards and Effects	490
Semantics	493
Advanced Concepts	495
Composite States and History	495
Pseudostates and Final State	499
Entry and Exit Behaviors	502
Semantics	504
Entry and Exit Points	511
Submachines	513
Design Considerations	515
Chapter 15: Collaborations and Interactions	517
Collaborations and Interactions	517
Motivation	517
Collaborations	521
Interactions	524
Semantics of Interactions	527
Messages	532

Fragments	534
Interaction References	543
Chapter 16: Commands, Presentation, and Architecture	547
Commands	547
Class Command	548
Built-In Commands	552
Presentation	563
The Presentation Layer Architecture	563
GUI Style Configuration	570
GUI Components and Widgets	573
GUI Component Library	576
Application Architecture	586

Part IV: Method

Chapter 17: About the Method	593
Activities and Artifacts	593
Development Activities and Artifacts	593
UML Analysis and Design Models	595
Requirements Engineering	598
Activities and Artifacts of Requirements Engineering	598
Requirements Specification Document	601
Chapter 18: Conceptual Modeling	609
The Process of Conceptual Modeling	609
Identifying Concepts and Relationships	612
Identifying and Specifying Classes and Attributes	612
Identifying Generalization/Specialization Relationships	616
Identifying Associations	620
Modeling Type-Instance Relationships	623
Chapter 19: Modeling Functional Requirements	631
Actors and Use Cases	631
Actors	631
Use Cases	633
Relationships Between Use Cases	637
Specifying Use Cases	644
Managing Use Cases	646

Contents

Business Processes and Use Cases	647
Discovering and Engineering Use Cases	651
Planning Iterations	658
 Part V: Supplemental	
Chapter 20: Characteristics of Information Systems	663
Domain-Related Characteristics 663	
Complexity	663
Conceptualization	664
Large-Scale Dynamic Instantiation	665
Functionality	665
Evolution	666
Usability-Related Characteristics 667	
Interactivity	667
Appropriateness	667
Timeliness	668
Availability and Location Independence	668
Security	668
Ease of Manipulation	669
Source Versus Derived Information Trade-offs	669
Deployment-Related Characteristics 671	
Diversity and Quantity of Data	671
Scalability	671
Persistence	672
Concurrency Control	673
Distribution	676
Fault Tolerance	682
Portability	684
Chapter 21: Process and Principles of Software Development	685
Project Management Process Models 685	
Objectives and Principles 688	
Objectives	688
Principles	690
Chapter 22: The Relational Paradigm	695
Introduction 695	
Fundamental Concepts 696	
Mathematical Foundation 701	

Actions Upon Structure	703
Advanced Concepts	706
Views	706
Referential Integrity	707
Triggers and Stored Procedures	707
Indices	708
Normalization	709
SQL	710
SELECT Statements	711
Data Modification Statements	720
DBMS Support	721
Development Tools Support	722
Chapter 23: Structured Analysis	727
Entity-Relationship Modeling	727
Basic Concepts	727
Advanced Concepts	729
Actions Upon Structure	732
Mapping to Relational Model	732
Data Flow Modeling	734
Chapter 24: Introduction to the Object Paradigm	737
Fundamental Object-Oriented Concepts and Principles	737
Abstraction and Abstract Data Types	738
Encapsulation	740
Inheritance and Subtyping	741
Polymorphism	742
Object-Oriented Decomposition	743
Object-Oriented Programming	745
Abstract Data Types and Instances	745
Encapsulation	746
Object References	746
Inheritance	747
Action Language	747
References and Bibliography	749
Index	753

Preface

Logical complexity of software systems is one of the main factors causing problems and errors in their planning, design, development, testing, deployment, maintenance, and use. There is a common understanding that building complex software systems requires careful planning, good architectural design, and well-controlled development processes. Many good books and papers, as well as all software engineering curricula, address this issue and yet, many software projects fail, miss their deadlines, or exceed their budgets. Building or maintaining a complex system (be it software or not) is always connected to a risk of mistakes and missed requirements, because humans (who are supposed to build the system) are intrinsically prone to errors when handling too many details and interrelated components at a time.

However, logical complexity is not completely inherent to software systems. On one hand, there is an inevitable component of complexity that is inherent to the very problem domain a software system deals with. The term *essential complexity* refers to that part of logical complexity inherent to the problem domain, and not introduced by a solution or the implementation technology used for it. Essential complexity is, thus, the “natural” part of complexity that cannot be removed and will exist in every solution to a problem, simply because a simple solution to the problem does not exist. However, essential complexity stands in contrast to *accidental complexity*, which arises purely from the implementation technology, tools, and methods applied in a solution. While essential complexity is unavoidable by any approach chosen to solve a problem, accidental complexity is caused by that very approach.

One of the main tasks in software engineering as a discipline is to discover means to minimize accidental complexity. Accidental complexity is to be minimized in any good software architecture, design, and implementation.

Sometimes, accidental complexity can be caused by mistakes such as ineffective planning or project management, or a low priority placed on a project. However, some accidental complexity always occurs as a result of solving any problem. For example, the complexity caused by out-of-memory errors in many programs is an accidental complexity that occurs because someone decided to use a computer to solve the problem [Wiki].

Another significant cause of accidental complexity is a mismatching or immature technology or process selected for the development of a software system. If the available technology (including the language used for software development) requires the developer to write more words or perform more actions to specify a design decision than is really necessary, the artifacts of the development will be accidentally complex. Using an assembly language to implement a non-trivial algorithm, and using a file system interface to build a database application are simple, extreme examples of such mismatching technology. A less obvious example of such accidental complexity is when some code has to be written to specify that a relationship between two objects has to be established when one object is dragged and dropped on the other object. This can be done in an easier and more direct way, by demonstration.

For that reason, raising the level of abstraction of the technology used for development of software systems, and doing so in a way that it better matches the problem domain of those systems, is one of the basic means for guarding against accidental complexity. Raising the level of abstraction is one of the main

Preface

characteristics of the evolution of software engineering as a discipline. As Bran Selic once said, “There has been no revolution in software engineering since the invention of a compiler.” In other words, once we understood that we did not have to talk to the computer in the language its hardware understands, but rather we can do it in a language that is more suitable for us, and which can be automatically translated into the language of the machine, we made the most significant breakthrough in software engineering. Everything since then has basically been all about raising the level of abstraction of the language used to program machines.

The point of raising the level of abstraction is to achieve better *expressiveness*. By using a language that better matches the problem you want to solve, you can say more “facts” in fewer “words.” This also means that you can do more with less work. In addition, written words do not have to be the only way to communicate with the machine. Pictures (such as diagrams), motions, and sounds (for example, spoken words) have already been a mode of communication between humans and computer programs, so they can be in software development, too.

This book contributes to the technology of developing one of many kinds of software systems, and proposes a technique that can improve development efficiency by raising the level of abstraction and reducing accidental complexity.

Model-driven development is one approach to raising the level of abstraction that has been successfully exploited for more than a decade. Its basic premise is to use models instead of (solely) code to specify software. Models are generally nonlinear forms, as opposed to code that is inherently linear.¹ *Non-linear* means that models consist of elements that are interrelated in a manner that is freer than a simple sequence where each element can have (at most) two adjacent elements. For that reason, models are usually rendered using visual notations, such as diagrams, instead of pure text.

The software development approach described in this book is model-driven.

The Unified Modeling Language (UML) is a standard language that is used for modeling software. It was proposed in the mid-1990s, and was first standardized in 1997. It is a general-purpose language aimed at modeling all kinds of software systems.

The approach described in this book uses UML as the modeling language.² The book follows the definitions and specifications given in the reference [UML2].

However, the scope of this book does not cover all kinds of software systems. Instead, it is limited to one special kind of software systems known as *information systems*. The introductory part of this book defines what is precisely meant by this term. In short, this book focuses on all those applications that have the following properties:

- A complex conceptual underpinning** — The applications rely on rather rich sets of concepts, properties, and relationships from their problem domains.

¹Note that code is a sequential form, because it represents a string of characters. Machines and humans read code in a sequential order, one character after another. To improve its readability, machines render code in two-dimensional viewports, but it is still inherently sequential.

²As of this writing, the latest UML standard is version 2.2. This book describes this version of UML and is based on the reference [UML2].

- ❑ **Large-scale dynamic instantiation** — During exploitation, the applications manipulate large spaces of instances of their concepts and relationships. These instances are dynamically created, modified, retrieved, queried, presented, and deleted. They are traditionally referred to as *data objects*.
- ❑ **Persistence of the run-time space** — The applications rely on what is conventionally called a *database* behind.
- ❑ **Interactivity** — The applications intensively interact with users and/or other systems to accomplish their purpose through user or machine interfaces.

This book focuses on model-driven development of information systems using UML.

UML is not, however, a fully formal language. This means that its semantics are not defined in an unambiguous way in all its elements. For that reason, UML cannot be used as a language in the same way as traditional programming languages, in which a specification of a software system can be unambiguously interpreted by machines (for example, compiled and executed). In order to be such, a language must have formal, unambiguous semantics — that is, a unique interpretation of each of its concepts that is supposed to have run-time effects.

In addition, UML is a general-purpose modeling language that can be *profiled* for a specific domain of problems. For example, standard UML leaves many so-called *semantic variation points*, which allow a profile to interpret certain language concepts in several ways. A profile can also reduce the set of the language concepts used in a particular problem domain, or extend the semantics of the concepts in a controlled way. This way, a profile can customize the standard language so that it becomes fully formal and, thus, executable. A model built in such a profile represents the implementation of the software at the same time, and, because it can be executed, is not just an informal sketch of the design.

This book proposes and describes one new executable profile of UML for the described application domain. It is but one of several existing profiles of UML with formal and executable semantics, specifically tailored for the domain of information systems.³

On one hand, the relational paradigm has been proven and widely accepted as the underpinning technology for building information systems. On the other hand, as another software-development paradigm with significantly more abstract and expressive concepts, object orientation has been successfully used for decades in programming. UML is one of the languages based on the object paradigm.

The marriage of object orientation with information systems development has been predominantly accomplished by using object-oriented programming (OOP) languages to implement behavior (or the so-called *business logic*) upon the underlying relational database, possibly accessed through a data persistence layer that performs object-to-relational mapping. This approach has partially replaced the use of fourth-generation programming languages that directly fit into the relational paradigm. At its current stage of technical development, this widely used approach suffers from discontinuities in development caused by incomplete or informal coupling of the object with the relational paradigm.

³For that reason, the term “executable UML” does not refer to any particular executable version of UML, but is rather a generic term that denotes any formal and executable specialization of standard UML. One such executable specialization of standard UML is presented in this book.

Preface

This book discusses the problems of these technologies, how they affect development, and how they can be overcome.

In short, this book explores the following:

- A technology for rapid development of one kind of applications referred to as information systems
- The use of the object paradigm and model-driven development of information systems
- One executable profile of UML for model-driven development of information systems

Following are the goals of this book:

- To provide an in-depth tutorial on model-driven development and UML for building information systems
- To show how information systems can be understood better and developed more efficiently by using the object paradigm, model-driven development, and a profile of UML that is formal and executable (rather than the relational paradigm or its incomplete coupling with object orientation)

Note that this book is *not* any of the following:

- A tutorial on, a reference specification of, or a textbook about the entire general-purpose UML** — This book does cover a major part of UML, but there are still parts of UML that are not covered. Instead, the book focuses on the concepts and parts of UML that are most likely to be needed in building information systems.
- A complete tutorial on the object paradigm or any traditional OOP language** — However, this book does describe the fundamental concepts of object orientation.
- A complete tutorial on information systems or all the related technologies** — Part V of this book does, however, provide a condensed recapitulation of the main facts about information systems and the technology of their building, including their architectures, the relational paradigm, entity-relationship, structured analysis, and SQL.
- A complete textbook on the development process of software systems in general, and information systems in particular** — Part IV of this book does, however, provide a quick practical guide to the proposed development method.
- A book that describes patterns or other techniques and building blocks for building information systems** — This book does not teach how to build information systems through the use of complex, integrated examples and case studies. Instead, it teaches concepts and principles, using many small, simple, and particular examples for illustration.

Whom This Book Is For

This book will be useful to software practitioners who analyze, specify, design, model, develop, or test information systems. This book is for those who want to improve their knowledge and productivity by exploiting model-driven rapid application development with an executable profile of UML. Readers who might benefit include system analysts, system and software architects, designers, developers, and testers.

The book will also be interesting to researchers who want to explore new software development strategies, methods, and metaphors, especially model-driven development and programming by demonstration. The book introduces some new concepts and ideas that could be interesting to explore further.

This book can also be used as a textbook for higher-education courses on information systems, model-driven software engineering, and UML.

The reader's prior knowledge of the object paradigm or any of the OOP languages is a plus, but is not necessary. This book gradually introduces the basic concepts and principles of object orientation.

Similarly, prior knowledge of the relational paradigm or any of the relational database management systems (RDBMSs) and SQL is not essential, although it is desirable. Part V of the book summarizes these topics for those who are not familiar with them. On the other hand, readers familiar only with these topics will experience a paradigm shift.

Finally, prior knowledge of UML is not needed at all. The book is a complete beginner's tutorial of (a profile of) UML. However, experienced users of UML will also benefit from clarification of many vague concepts of UML and their semantics.

The prerequisite for reading this book is general knowledge of programming. Knowledge and experience in building information systems is a plus, although not essential.

How This Book Is Structured

The book is divided in the following parts:

- “Introduction” (Part I, Chapters 1–3) — This part quickly introduces information systems. It then elaborates on traditional technologies of development of information systems and their advantages and drawbacks. This part clearly indicates the main issues with the widespread use of traditional paradigms for building information systems (most notably, relational modeling or entity-relationship modeling), or with incomplete coupling of object orientation (and OOP languages) with relational modeling. The analysis provides the motivation for the approach presented in the book.
- “An Overview of OOIS UML” (Part II, Chapters 4–6) — This part is a quick overview of the executable profile of UML proposed in this book, referred to as the *OOIS UML profile*. This part quickly presents the main concepts and ideas that will be described in more detail later in the book.
- “Concepts” (Part III, Chapters 7–16) — This central part of the book thoroughly explains the concepts of OOIS UML and their semantics.
- “Method” (Part IV, Chapters 17–19) — This part provides a quick guide to the proposed method for applying the OOIS UML profile for building information systems.
- “Supplemental” (Part V, Chapters 20–24) — This part provides auxiliary tutorial material for the traditional technology that is widely used for building information systems nowadays, and that is not essential for understanding the main parts of the book. The supplement includes a summary of the general characteristics of information systems, some basics of software engineering processes, the relational paradigm, entity-relationship modeling, structured analysis, and general principles of the object paradigm. These tutorials are provided for the convenience

of the interested readers who are not familiar with these topics and traditional technologies, or as quick reminders for those who are experienced with them.

If you are familiar with the notion of information systems and the traditional technology of their development (including relational databases and entity-relationship), you can simply read the book from its beginning. In the first three chapters of the book, you will find an analysis of the issues that you have probably faced in your work. You will also find explanations of the causes of the issues, while the central part of the book will provide solutions.

If you are not familiar with these traditional technologies, you can still start reading from the beginning. However, you can also skip to the supplement to gain some basic knowledge of the technology you do not know well. However, this knowledge is not essential for understanding of the main part of the book.

Finally, if you are just eager to see what this book is all about, and the new and original information contained herein, simply read Part II and you will get the main idea. Then you can go back or forward as you like.

About the Supporting Software and the Accompanying Site

The method described in this book can be applied even without full-fledged tool support. The author has taken part in several successful industrial projects where only customized off-the-shelf or ad hoc developed tools were used to partially support some activities in the approach (such as UML modeling tools, customized code generators, and object-to-relational mapping frameworks). Even without full-fledged tool support, the proposed approach can boost the productivity and improve the quality of the produced software because of the raised level of abstraction, better expressiveness of the modeling language and its semantics, clear architecture of the software system, and a well-controlled development method.

However, obviously, the full benefit of the proposed approach can be reaped only with strong and full-fledged support of computer-based tools. There can be many different implementations of the proposed UML profile with the appropriate tool support. The author is the inventor and has served as the chief architect of one such tool, named SOLoist⁴, which has been developed for and successfully applied to a wide variety of industrial projects since 2000, and which supports many concepts described in this book.

See www.ooisuml.org for more discussion about the profile and the method presented in this book, their open issues and further improvements, as well as their implementations and applications to real-world projects.

Conventions

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

⁴SOLoist is a trademark of Serbian Object Laboratories d.o.o. (SOL)

- Each section of the book ends with a summary enclosed in a box like this.

As for styles in the text:

- We *highlight* new terms and important words when we introduce them.
- We show keyboard strokes like this: Ctrl+A.
- We show filenames, URLs, and code within the text like so: `persistence.properties`.
- We present code in two different ways:

We use a monofont type with no highlighting for most code examples.

We use gray highlighting and underlining to emphasize code that is of particular importance in the present context.

Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books (such as a spelling mistake or faulty model fragment), we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration and, at the same time, you will be helping us to provide even higher quality information.

To find the errata page for this book, go to <http://www.wrox.com> and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list (including links to each book's errata) is also available at www.wrox.com/misc-pages/booklist.shtml.

If you don't spot "your" error on the Book Errata page, go to www.wrox.com/contact/techsupport.shtml and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

p2p.wrox.com

For author and peer discussion, join the P2P forums at p2p.wrox.com. The forums are a Web-based system for you to post messages relating to Wrox books and related technologies, and to interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At <http://p2p.wrox.com>, you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

- 1.** Go to p2p.wrox.com and click the Register link.
- 2.** Read the terms of use and click Agree.
- 3.** Complete the required information to join, as well as any optional information you wish to provide, and click Submit.
- 4.** You will receive an e-mail with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P. However, in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum emailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works, as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

Part I

Introduction

Chapter 1: Information Systems Modeling

Chapter 2: Traditional Approaches to IS Development

Chapter 3: The Object Paradigm

1

Information Systems Modeling

To provide a foundation for the discussions throughout this book, this chapter begins by defining what is actually meant by the term *information system*. The focus is on model-driven engineering of the software component of information systems. This chapter also introduces and describes the very notion of modeling. The chapter concludes with a brief discussion about software engineering processes, an important aspect of building successful information systems.

Definition of Information Systems

Information systems have played a key role in the history of computers and their use in everyday human activities. It is hard to imagine even a small company, institution, or organization that does not have a need for storing and using information of a different kind. We are all witnesses of the tremendous improvement of computer and communication technology, which support ever-increasing demands of human community for interchanging and utilizing information. It is not necessary to discuss the role and importance of information systems such as healthcare systems, enterprise systems, banking and financial systems, educational information systems, customer-support systems, governmental systems, and many other kinds of information systems (see Figure 1-1).

Trying to catch up with the importance and ever-emerging demand for improved functionality and performance of such systems, the hardware and software technology for their implementation seem to constantly stay behind. It has always been a question of how to improve the technology (especially that for software development) to meet the users' needs. Because this question defines the scope of this book, before trying to answer it, it is necessary to set up the context and define what information systems really are.

First and foremost, information systems are *systems*. A system is a set of elements organized to cooperate in order to accomplish a specific purpose. Elements of a system collaborate synergistically, in a manner that provides a behavior and quality bigger than the sum of its parts. It is likely that a

Part I: Introduction

particular part of a complex information system (such as a piece of hardware or software) may be of no use if it is taken without the other parts. Only a complex interaction of such parts accomplishes the real purpose of the entire system.

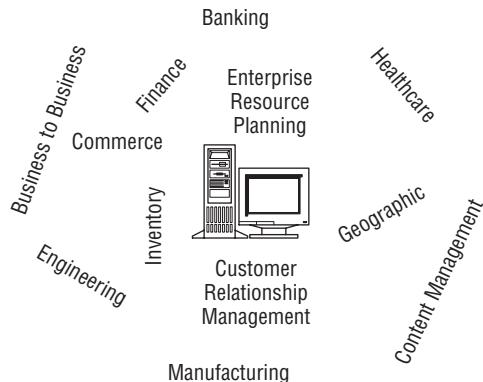


Figure 1-1: Information systems

Moreover, information systems are *computer-based* systems, meaning that their elements encompass hardware, communication equipment, various software subsystems, information, and users. The software of almost every information system is also a complex subsystem, comprised of communication software, the operating system, the database management system, different application modules, presentation modules, and so on. Because information systems generally assume storing and processing of large amounts of data, computer technology is irreplaceable for their implementation.

Next, information systems deal with *information*. Avoiding more complex and abstract definitions and focusing on computer-based information only, information can be understood as a piece of data that is structured, stored, transferred, processed, and presented in a proper manner, and at a right time, so that it has a certain meaning and accomplishes a specific purpose for its users. Therefore, information is not just a simple fact, but a piece of data that is shaped properly and provided timely to meet the specific user's needs.

For example, the fact that a certain patient has blood type A has no importance to the accountant using a hospital information system. To the accountant, it is much more important to present the total quantity of blood type A used for transfusions in a certain period. On the other hand, the same fact is key for a doctor who must organize the transfusion for that patient. However, if that fact relates to a patient that was hospitalized 50 years ago, it may be of no importance even to the doctor. If, however, the fact relates to a patient who just arrived in the Emergency Room, it has the crucial importance and represents the right information at the right time.

In summary, an information system is a computer-based system primarily dealing with pieces of data that are structured, stored, transferred, processed, and presented in a proper manner, and at the right time, so that they have a certain meaning and accomplish a specific purpose for the system's users.

For a more in-depth discussion of the variety of characteristics of information systems (characteristics that can more precisely describe the nature of information systems, as well as their engineering aspects), see Chapter 20 in the Supplement.

Section Summary

- ❑ An information system is a computer-based system primarily dealing with large amounts of data that are structured, stored, transferred, processed, and presented in a proper manner, and at a right time, so that they have a certain meaning and accomplish a specific purpose for the system's users.

Models and Modeling Paradigms, Languages, and Tools

Model-driven software engineering is a modern discipline that covers the building of software based on *models*, using modeling languages and tools. However, the notions and principles of this discipline have roots in other engineering approaches. This section further explores these topics.

Modeling

The engineering disciplines that are much more mature than software engineering (such as civil, mechanical, or electrical engineering) require that the designers of a new system build simplified representations of the system under construction in order to analyze the future system's characteristics before it is built, as well as to convey their design to the implementers. Such a simplified representation of the system under construction is called a *model* of the system, and the process of its creation is called *modeling*.

Models can be expressed in abstract terms, such as, for example, mathematical models, schemas, or blueprints. On the other hand, models can be physical, usually small copies made of plastic, wood, or other materials. Models allow the designers to experiment with the design and test its characteristics before the system is actually built. Such an approach reduces the risk of faulty construction of the real system with potentially disastrous or costly consequences.

There is no reason why software engineering would not follow these good practices of other engineering disciplines and exploit models and modeling in building complex software systems. One crucial difference is that in software engineering, the “material” that models are made of can be the same as the ultimate system — the model can be a formal and unambiguous specification that can be either directly interpreted by another software system, or transformed into a form that can be executed by hardware or interpreted by another software system. In that case, the model is *executable* and represents, when complete, the very system under construction.

A model is a simplified representation of the real world. It is built to provide a better understanding of a complex system being developed. Models of complex systems are built because such systems cannot be comprehended in their entirety. By modeling, four aims are achieved [Booch, 1999]:

- ❑ Models help to visualize a system as it is, or as designers want it to be (*visualization*).
- ❑ Models permit the designer to specify the structure and behavior of the system (*specification*).

Part I: Introduction

- ❑ Models give templates that guide in constructing systems (*construction*).
- ❑ Models document the design decisions that have been made during development (*documentation*).

Basically, programs created in traditional programming languages may be treated as models because they specify the software system in a formal and executable way. However, programs created in traditional programming languages are not considered as being models, because of the following:

- ❑ Models basically deal with more abstract notions, while programs deal with subtle implementation details.
- ❑ Models may often be incomplete or imprecise, especially in early analysis phases, although this does not prevent them from being executable. Programs are ultimate artifacts of software development.
- ❑ Models are usually specified in visual (diagrammatic) notations, combined with textual (sequential) parts, while traditional programming languages predominantly presume textual (sequential) forms.

The software engineering discipline in which models are the central artifacts of development, which are used to construct the system, communicate design decisions, and generate other design artifacts, is called *model-driven engineering (MDE)*, while the development of systems that exploit MDE is called *model-driven development (MDD)*.

Modeling Languages

To build models, developers must know the vocabulary that can be used in modeling. The vocabulary available for modeling (that is, the set of concepts, along with the semantics of these concepts, their properties, relationships, and the syntax) form the definition of the *modeling language*. Using a certain modeling language, developers build sentences in that language, creating models that way. Therefore, a modeling language is the key tool available to developers for building models because there is no purpose in making models without understanding their meaning. Hence, a modeling language provides a common means for communication between developers.

To be successful, a modeling language must be carefully balanced to meet somewhat opposing demands.

On one hand, a modeling language must be simple enough to be easily understandable and usable by modelers. If the modeling language were too complex to be comprehensible, it would not be widely accepted by the software community. Additionally, it is very desirable that users who pose the requirements also understand the modeling language, at least the part that is used in specification of requirements. This property may significantly reduce the risk of misunderstandings between users and modelers during requirements specification.

On the other hand, a modeling language should not be too simple or too specific. If the modeling language were not general enough, it could not cover all situations that could arise in the real world.

In addition, a modeling language must be abstract enough to be conceptually close to the problem domain. A modeling language that is not abstract enough suffers from a large conceptual distance

from the problem domain. In that case, the development requires a big mental effort because the modelers must take the conceptual mapping from the problem domain into the model. In other words, the language must be *expressive* enough.

Expressiveness is a descriptive property of a language that measures the “quantity” of meaning of certain language concepts. An expressive language consists of concepts that have rich semantics and, thus, have plentiful manifestation at the system’s execution time. Models in an expressive language may be concise, and yet provide lots of run-time manifestations that correspond to the posed requirements. Expressiveness is, in other words, a measure of conciseness with which a particular logical design may be expressed in a certain language. Put more simply, it measures how many (or few) words you must say in the given language in order to express some logical design or intention.

Therefore, the modeling effort and development cost is directly affected by expressiveness. If the language is expressive enough, models consist of smaller sentences, and developers make less effort to specify systems under construction, and vice versa.

Additionally, a modeling language should allow informal, incomplete, or inconsistent models because the process of modeling in the early phases of requirements specification and system conception often assumes such models. During the early phases of requirements engineering and system conception, system analysts and designers have vague and sometimes incorrect visions of the systems being specified and constructed. Therefore, the analysts should be able to make sketches in the modeling language, in a way that allows their later refinement. If the modeling language does not allow such imprecise modeling, it could not be used in the early requirements specification and conceptualization phase. However, it is very useful if the same language is used in the entire process of system development, but not just during requirements specification or just during design.

On the other hand, a modeling language should be simple and primitive enough in order to have precise semantics that allow transformation of models into an implementation form. Highly abstract models are usually transformed into lower-level forms that can be either interpreted by other software systems, or further transformed into even lower-level forms, or ultimately executed by hardware (which also interprets the binary code in some way).

For example, source code in a traditional textual programming language such as C++, Java, or C# is transformed (compiled) by other software systems (compilers) into either a binary executable program (for C++), which is executed by hardware, or into an intermediate form (as in Java or C#), which is interpreted by other software systems (virtual machines), to provide the running application. Similarly, a model made in a visual modeling language, such as the Unified Modeling Language (UML), which is a focus of this book, may be transformed into the target programming language code (for example, C++, Java, or C#).

The described transformation into a lower-level form can be performed manually, semi-automatically, or completely automatically, using the corresponding transformer. It is very useful if a highly abstract model of a system, specified in a modeling language, can be transformed completely automatically (in one or several steps) into a form that can be executed or interpreted in a way that provides the running application, following the semantics of the initial model. In that case, the modeling language can be

Part I: Introduction

treated as executable, although it is not directly executed by hardware. This is because the model implies an automatically achievable form that can be ultimately executed by hardware.

In this chain of transformations and intermediate models (for example, source code, intermediate code, executable code, and so on), the form that can be interpreted by another software system to result in the running application (such as Java byte code or SQL statements), or that can be automatically compiled into an executable binary code (such as source code in C++), is referred to as the *implementation form*, and the language it is made in is referred to as the *implementation language*.

In order to be executable, the modeling language must have precise and completely formal semantics, and the model that should be executed (or, more precisely, automatically transformed into an implementation form), must be formal, unambiguous, and consistent. These are the most important characteristics of the modeling languages that claim to be useable and useful for modeling of any complex software system in general, and information systems in particular.

The concrete notation of modeling and implementation languages can be textual or visual. Textual languages allow you to create inherently *sequential* models. Although textual models (that is, programs or scripts or sentences written in textual languages) may be visually perceived by the developer in two dimensions, they are inherently sequential because the transformers parse them as sequences of characters.

Conversely, visual languages assume making models in two or even three dimensions, most often using diagrammatic notations combined with textual fragments. It is usually the case that a pictorial, diagrammatic notation is much more descriptive, intuitive, and easier to perceive than the equivalent sequential (textual) form. This is why modeling languages with diagrammatic notations are more popular for abstract modeling.

However, it is not always true that visual languages are more usable than textual ones. Some intentions can be much more effectively expressed in a textual form than in a visual form. For example, to specify a simple loop or an expression such as a regular expression or a simple addition of two operands, a textual form may be much more convenient and concise than an equivalent graphical notation.

In short, both textual and diagrammatic languages have their advantages and drawbacks. Usually, a combination of both is most efficient in modeling.

Figure 1-2 illustrates the comparison of abstract versus low-level modeling, and visual versus textual modeling languages. A small piece of a model in a highly abstract, visual language is shown in Figure 1-2a, whereas the equivalent model in a low-level, textual language is shown in Figure 1-2b. Both examples model the same simple fact from the problem domain that a person (described with name, social security number, and address) may work for at most one company (described with name and address), and a company may employ many persons.

The example shown in Figure 1-3 illustrates a yet more dramatic difference between the abstract visual model in Figure 1-3a, and its semantic equivalent in a lower-level, textual programming language in Figure 1-3b. The figure shows the lifecycle model for a course enrollment in an imaginary educational institution. The diagram in Figure 1-3a depicts how a student's application for a course is processed until it gets either rejected or accepted, or suspended (and later resumed) or canceled.

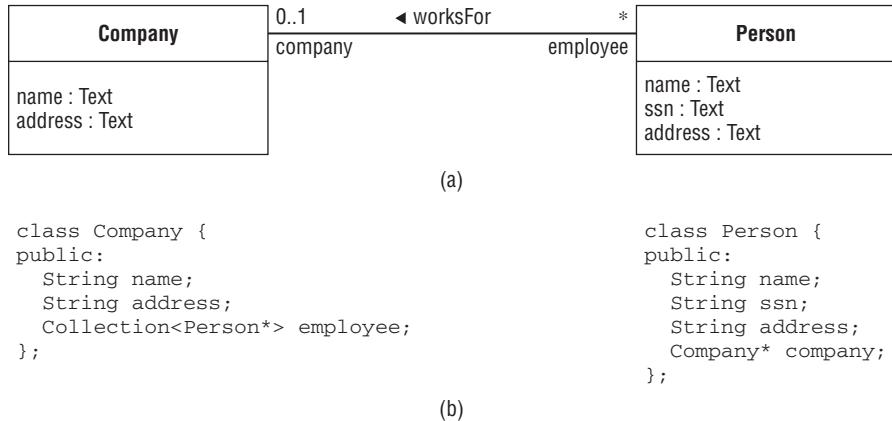


Figure 1-2: Abstract vs. low-level modeling and visual vs. textual modeling languages.
 (a) A model in a highly abstract, visual language. (b) A model in a low-level, textual language. Both examples model the same simple fact from the problem domain that a person (described with name, social security number, and address) may work for at most one company (described with name and address), and a company may employ many persons.

Modeling Tools

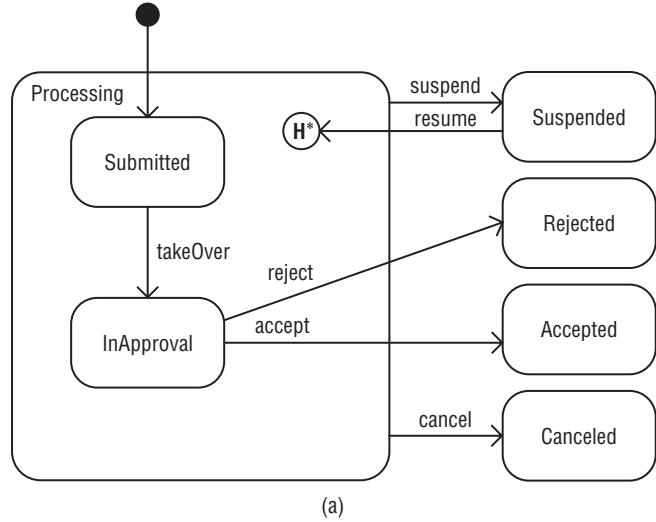
As in any other engineering discipline, tools are key factors to successful production. In the context of modeling, different kinds of software systems are used as modeling tools. The system that is used as an environment for creating and editing models (and that is responsible for their specification, visualization, and consistency checking) is called the *model editor*. The tools that transform the models into lower-level forms are generally called *transformers* or *translators*, or specifically *generators* or *compilers*, depending on their target domain.

For example, a tool that transforms a higher-level, abstract, and visual model into a textual source code in the implementation language (for example, C++, Java, or C#) is often referred to as a *code generator*, although it is sometimes called a model compiler.

Finally, the tool that transforms a program written in a textual language (such as C++, Java, or C#) into the executable (binary) code, or interpretable (intermediate) code, is often called the *compiler*. Many other kinds of modeling tools are also used for specific tasks in development, such as model analyzers, model comparators.

Modeling Paradigms

There are modeling languages that are particularly appropriate for (or the same) problem domains, such as those that analyze or compare models, and others. These languages, based on the same set of fundamental concepts, form a *modeling paradigm*. The languages of the same paradigm may differ in more or less detail or variations.



(a)

```

switch (state) {
    case Submitted:
        switch (event) {
            case takeOver: state = InApproval; break;
            case suspend:  prevState = state; state = Suspended; break;
            case cancel:   state = Canceled; break;
        };
        break;
    case InApproval:
        switch (event) {
            case reject: state = Rejected; break;
            case accept: state = Accepted; break;
            case suspend: prevState = state; state = Suspended; break;
            case cancel: state = Canceled; break;
        };
        break;
    case Suspended:
        if (event==resume) state = prevState; break;
}

```

(b)

Figure 1-3: Abstract vs. low-level modeling and visual vs. textual modeling languages. (a) A model in a highly abstract, visual language. (b) A model in a low-level, textual language. Both examples model the lifecycle of an application for a course.

For example, the relational modeling paradigm is based on the fundamental concept of mathematical relation, which is often represented by a table with columns (or fields, properties, attributes) and rows (records or tuples). Although relational database management systems (RDBMSs) often support somewhat different languages, varying in other concepts built upon the basic relational concepts, they are still based on the same paradigm. For additional information about the relational paradigm and DBMSs, see Chapter 22 in the Supplement.

Similarly, the procedural programming paradigm assumes some common concepts such as data type, variable, expression, statement, loop, subprogram (procedure or function), argument (formal and actual),

subprogram invocation, recursion, and so on. Many programming languages that fall in this category (for example, Algol, Pascal, C, Fortran, and so on) support most of these concepts in very similar ways, but differ in many other less relevant details.

This book is devoted to a standard modeling language for general software modeling, called the *Unified Modeling Language* (UML). This language supports the object paradigm, significantly different from the relational paradigm, for example. The basic concepts of the object paradigm are summarized in Chapter 24 of the Supplement, but will also be explained from the beginning in Part II.

Section Summary

- ❑ A *model* is an abstract, simplified, or incomplete description of the system being constructed.
- ❑ The language used for creating models is called the *modeling language*. It encompasses concepts used for creating models of systems, their semantics, properties, and relationships, along with the syntax for creating correct models.
- ❑ A modeling language should be simple enough to be easily comprehensible and usable by modelers. On the other hand, the language should not be too simple or too specific.
- ❑ A modeling language must be expressive and abstract enough to be conceptually close to the problem domain. However, the language should be simple and primitive enough to have precise semantics that allow transformation of models into implementation forms.
- ❑ If the concepts of a modeling language have formal semantics that enable models to be transformed completely automatically into forms that can be executed or interpreted in a way that results in running applications, the language is called *executable*.
- ❑ The form that can be interpreted by another software system that provides the running application, or can be automatically compiled into an executable binary code, is referred to as the *implementation form*, and the language it is made in is called the *implementation language*.
- ❑ The concrete syntax of modeling languages can be visual (diagrammatic) or textual (sequential).
- ❑ The software engineering discipline in which models are the central artifacts of development (which are used to construct the system, communicate design decisions, and generate other design artifacts) is called *model-driven engineering* (MDE), while the development of systems that exploits MDE is called *model-driven development* (MDD).
- ❑ The tool that is used for creating and editing models, responsible for their specification, visualization, and consistency checking, is called the *model editor*.
- ❑ The set of languages based on the same set of fundamental concepts form a *modeling paradigm*.

Processes and Methods

A modeling language is just a vehicle for modeling. It only specifies the concepts that can be used in modeling, as well as their semantics and the rules for forming correct models. However, the language itself does not give any direction *how* to specify requirements, create models, or develop a system. In other words, the *process* of development is not defined by the language itself.

In general, a process is a set of partially ordered steps intended to reach a goal. In software engineering, the goal is to efficiently and predictably deliver a software product that meets the users' needs [Booch, 1999].

In software production, at least two types of processes can be distinguished, depending on their scale and focus.

First, there is a higher-level *project management* process that deals with planning, organizing, supervising, and controlling phases and iterations during the entire project, interacting with the customers, managing resources, as well as defining milestones and artifacts of development phases. It is mostly independent of which modeling language or paradigm is exploited.

On the other hand, there is a lower-level *design process* that deals with how the modeling language and other technology is used and applied in different situations, as well as other things such as selecting or inventing suitable design patterns for given design situations, refactoring techniques, ways of using tools, and the like. To successfully build systems, designers must follow a proper design process that, in conjunction with the used modeling language, forms the *modeling* (or *design*) *method*.

For more information about the characteristics of project management process models, see Chapter 21 in the Supplement section. Part IV describes some elements of the design process proposed for the modeling method described in this book.

At this point, let's just briefly summarize some of the key properties of the proposed design method.

First, the proposed design method is based on the object paradigm in all its phases and all its artifacts. Second, it is oriented to production of abstract models instead of paper documents, as the principal artifacts of software development.

In addition, it encourages reuse of different artifacts and relies on a formal, executable modeling language as much as possible, and as early as possible. Such orientation can improve software production dramatically. Namely, if an executable model of the system being constructed can be obtained early enough, as soon as the requirements are captured, but without exhausting conceptual mappings, the ultimate implementation can then be reached automatically and rapidly. As already stated, the basic precondition for this is a highly abstract, but formal and executable modeling language, as well as a design process that supports its use. This is also one of the key focuses of this book.

Section Summary

- ❑ A process is a set of partially ordered steps intended to efficiently and predictably deliver a software product that meets users' needs.
- ❑ The *project management* process deals with planning, organizing, supervising, and controlling phases and iterations during the entire project, interacting with the customers, managing resources, as well as defining milestones and artifacts of development phases.
- ❑ The *design process* deals with how the modeling language is used and applied in different situations, as well as other things such as selecting or inventing suitable design patterns for given design situations, refactoring techniques, ways of using tools, and the like.
- ❑ The design method proposed in this book relies on using formal and executable models, as well as producing such models as early as possible.

2

Traditional Approaches to IS Development

This chapter provides a brief overview of the advantages and disadvantages of two prevailing paradigms for modeling data in information systems in the past — the *relational* paradigm and the *entity-relationship* (ER) paradigm. Although they have been supplanted by the object paradigm in the last decade or so, they are still widely used in practice.

You do not need extensive knowledge of these two paradigms to understand the concepts presented later in this book because these concepts will be introduced in detail. However, for the convenience of the interested readers without background knowledge in these two paradigms, Part V of this book provides quick tutorials on these topics.

In particular, Chapter 22 (in the Supplement) provides an overview of the relational paradigm and the related technology that are still commonly used for building information systems, such as SQL and the support of database management systems (DBMSs) and development tools. Readers who do not have a basic knowledge of the relational paradigm are encouraged to go through that tutorial and then return to this chapter.

Chapter 23, in the Supplement, contains a quick overview of the basic concepts and characteristics of the ER paradigm. UML (which is the modeling language that is the focus of this book) has inherited many concepts of ER and adapted them to the object paradigm. Some of the concepts of ER have been superseded by more appropriate solutions, however. Of course, the concepts that have been adopted by UML will be explained in detail later in this book. Readers who are interested in the basics of ER and the characteristics of modeling using that paradigm can go through that tutorial.

The analysis of the issues of information system development based on these paradigms provides a motivation behind the approach described in this book. Readers who are not interested in these two paradigms and the motivation behind the approach presented in this book can simply skip this chapter. However, the author's everyday experience in working on industrial projects and with different development and management teams shows that there is a common feeling among

the practitioners that there is a major issue with development using these paradigms that affects efficiency. It seems that there is a pervasive unease that the use of models does not bring much benefit to the development. This chapter provides explanations for the causes of these issues, while the rest of the book will provide the solutions.

Characteristics of Traditional Modeling Paradigms

In the history of databases and information systems, by far the most influential breakthrough was the emergence of the *relational paradigm*, or the so-called *relational database model* [Codd, 1970]. (According to the definitions provided in Chapter 1, the term *model* refers to a representation of a concrete system or application, whereas the term *paradigm* encompasses a set of concepts used for modeling. For this reason, the term *relational model* will be used only if it refers to a concrete application, and not to the entire paradigm.) Today, the vast majority of information systems rely on the underlying relational database model because it is easy to understand and use, the support of database management systems (DBMS) is stable and standardized, and the entire paradigm has been approved as suitable for successful systems deployment.

Although the conceptual simplicity of the relational paradigm is an advantage for its comprehension and implementation, it is a significant drawback if the paradigm is directly used for conceptual modeling. In that case, the modeler must map a concept from the problem domain into the relational model. This is because its simple fundamental concepts are not expressive enough for conceptual modeling.

For example, in a school information system, the fact that people (modeled with a table named `Person`) may attend courses (modeled with a table named `Course`) is modeled with a separate low-level inter-connecting table `PersonAttendCourse`, with the foreign keys referring to the tables for courses and people. This solution is rather technically oriented, too peculiar, and too far from the natural relationship between the abstractions of people and courses. Namely, observing from the perspective of the relational paradigm, the table `PersonAttendCourse` is by no means different from the tables `Person` and `Course`, although these tables represent real-world concepts of significantly different kinds (abstractions and their relationship). This difference is not directly noticeable in the relational model. In other words, the relational model may lose some important semantic information about the real world.

This is the reason why a more abstract paradigm was invented, called the *entity-relationship* (ER) paradigm [Chen, 1976]. It supports conceptual modeling more efficiently by differentiating entities (modeling abstractions from the problem domain) from relationships (modeling their conceptual associations). A model developed using the ER paradigm may be automatically and easily mapped to the corresponding relational model because these two paradigms are conceptually close.

ER addresses the modeling of structure — that is, of pure data. For modeling behavior (that is, functionality) of systems, practitioners sometimes combine other modeling paradigms with ER. The most commonly used traditional paradigms for modeling functionality are based on variations or derivatives of classical flowcharts and activity diagrams with different syntax and semantic flavors. Chapter 23, in the Supplement section, briefly describes one such paradigm, *data flow (DF) modeling*, which is based on processes that perform some transformations of the data packets that flow between the processes.

Although the history of computing includes lots of other languages, paradigms, processes, and methods for database design, conceptual modeling, and information system analysis and design, the described

paradigms have had the major influence in practice. This is why other traditional approaches will not be addressed in this book.

This section discusses the advantages and disadvantages of these traditional approaches. The discussion will consider two perspectives:

- ❑ **The perspective of the users** — That is, how users see the systems developed with traditional paradigms, including the aspects of usability.
- ❑ **The perspective of the developers** — With development aspects and technology support

Usability Aspects

From the user's point of view, information systems based on the traditional paradigms have several key advantages. First, the graphical user interfaces (GUI) of information systems based on the relational paradigm are typically oriented to *forms*, as shown in the example in Figure 2-1. Such common forms-oriented user interfaces usually expose the relational data model to users, which is easy to understand. Namely, regardless of how the forms are really designed, they somehow ultimately and inevitably build an impression that the data is organized into tables behind the façade. Users quickly get used to that perception. This is easy to explain because the concept of record set (that is, relation) fully fits into the intuitive human understanding of a table.

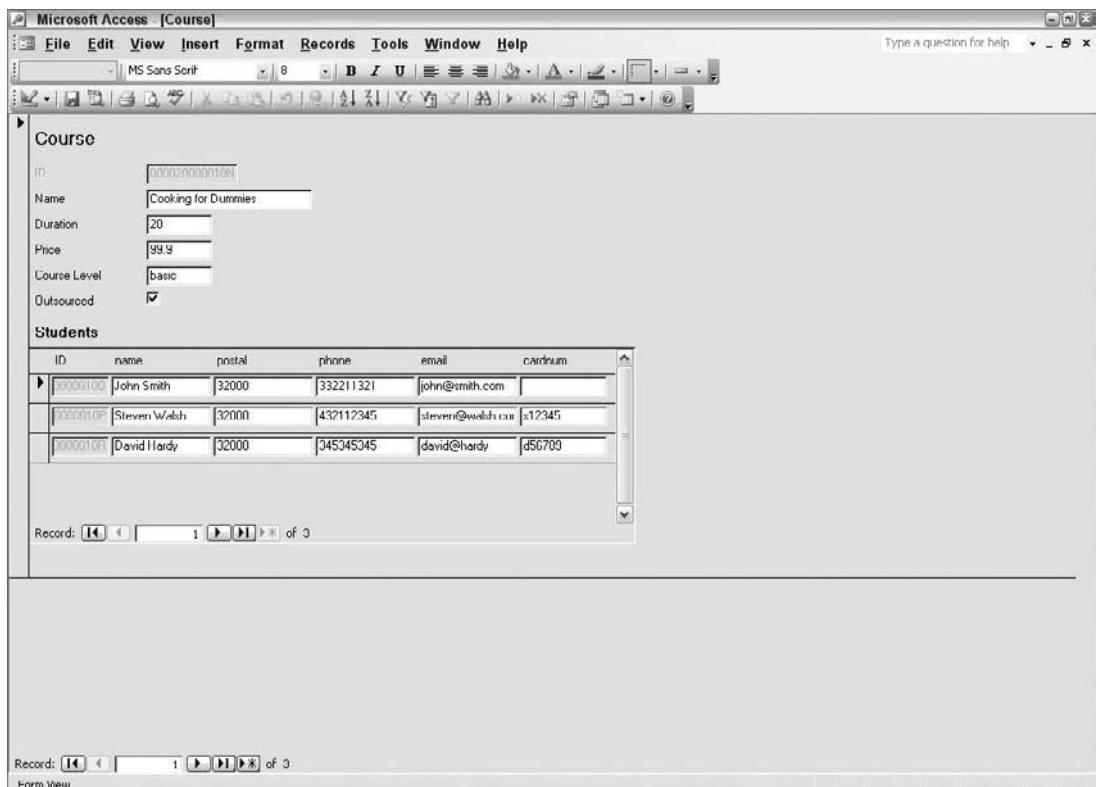


Figure 2-1: The appearance of traditional user interfaces based on the relational paradigm

Part I: Introduction

Second, dealing with forms is usually quite suitable for certain application domains. People are generally used to filling in various paper forms, and forms-oriented user interfaces are very simple electronic generalizations of their paper counterparts. This is why applications with such interfaces do not impose a steep learning curve. There are a number of examples where forms-oriented manipulation of data perfectly fits into the human's understanding of information processing. For example, filling in a form with grades for the students that took an exam, or an order for money transfer, are probably the easiest ways to specify the corresponding tasks to the computer.

On the other hand, the relational representation (whereby the data is organized strictly into tables) is not always the way users want to see the data. This is especially the case where relationships among entities are concerned. For example, the fact that a student attends three courses is an explicit piece of information that users want to see directly, and not through a set of interlinking records with the pairs of foreign keys with the student and the course IDs. This is why user interfaces always hide such implementation details by more explicit representations, such as the so called *master-details* forms (see Figure 2-1). In a master-details form, the master form renders the data about one entity (for example, a course in the upper part of the form in Figure 2-1), whereas the details subform renders the data about multiple entities related to the entity in the master form (for example, the students in the course, as in the lower part of the form in Figure 2-1). Note, however, that this representation is not a direct counterpart of the actual underlying implementation, and that there is a semantic gap between the two, which must be handled by the developer or by the development tool.

Some more complex structures of entities are even more inconvenient for representations available in traditional approaches. For example, consider a tree-shaped structure of entities: a structure of folders and files on a disk or a hierarchical organizational structure of departments in a company. The traditional forms-oriented user interfaces are capable of presenting only limited parts of the structure. For tree-shaped structures of entities, the traditional master-details form can present only one node in the master form and all its direct sub-nodes in the details subform, offering only one-step navigation to either another current node by moving the master form to a sub-node or to a parent node.

This is actually equivalent to the old-fashioned command line-oriented shells of file systems, where the user lists only the files in the current directory and moves to another current directory by specific commands. It is like you are looking at the structure "through a peephole," or "sipping the data through a drinking straw."

A better alternative is now available — the usual tree-view controls (as in Windows Explorer, for example) offer a far wider perspective and better navigation over tree-shaped structures. It is obvious that the same representation should be used for a hierarchical organizational structure of a company, or a compositional structure of a complex mechanical device. The difference between a tree-view representation of these structures and their representation through a master-detail form is the same as viewing the contents of a disk by Windows Explorer or by an old-fashioned command-line shell. In other words, it is like viewing the landscape from an open balcony instead through a narrow window, or drinking water from a glass instead of through a straw.

Furthermore, because of the described gap, forms-oriented applications sometimes require from users more data manipulation than is actually necessary. For example, in order to assign a student to a new course, the user must activate a certain application function, get to the form dedicated to that operation, probably select the student either from the list of offered students or by entering the student's unique ID, and then specify the attended course again by selecting it from a list or by entering its identifier.

In addition, applications sometimes unnecessarily burden users with entering keys to specify certain conceptual relationships or select entities. Although this approach is sometimes inevitable, it is not

always necessary. For example, it is much easier to specify that a student attends a course by dragging an icon representing that student and dropping it onto the icon representing the course, regardless of how and where in the application the two icons appeared (for example, in a result set of a search, in a list of enrolled students, or elsewhere). What may be even worse for the application usability is a lack of criteria for usage of patterns of interaction for accomplishing certain operations within the system (for example, selecting an entity). This may cause a highly inconsistent usage of different patterns by different developers across the same application.

Unless usability experts strictly control this issue, it may lead to confusing user interfaces that are difficult to fix later when the defects are revealed. For example, in one part of an application made by one developer, a function for enrolling a student in a course can require that users enter the student's ID, while in another part a student can be identified by other means (for example, searching by name).

Section Summary

- From the user's point of view, the relational paradigm is good because of the following:
 - It is more or less exposed to users and is easy to understand.
 - Forms-oriented GUIs are simple enough, perfectly suitable, and easy to use for some applications.
- On the other hand, the relational paradigm is not always good because of the following:
 - It is not always the way users want to see and manipulate the data.
 - Forms-oriented GUIs are tedious for some applications or tasks, and require from users more manipulations than really necessary (entering keys and so on).

Development Aspects

The development methods based on the relational paradigm have many advantages from the developer's perspective. First, like users, developers do not have much difficulty in understanding the simple concepts of the relational paradigm. This is probably one of the major reasons for its wide acceptance in the information systems development community.

Apart from that, the relational technology is mature and strongly supported by DBMSs. All the features of DBMSs that support distribution, multiuser concurrent access and concurrency control, security, and fault tolerance have had a very important role in the applicability of the relational paradigm. Among them, the support for the generic operations (that is, actions) upon the abstract data model used by developers is important, which hides the implementation details of the internal data representation.

Part I: Introduction

Finally, the precise mathematical formalism that lies beneath has helped in establishing strong and clear semantics of the relational model, both of its structural (the data model) and behavioral (the actions upon data) parts. This is very important for avoiding misunderstandings and faults in development.

As already stated, however, the core disadvantage of the relational paradigm is its low abstraction level and poor expressiveness, which means it is not suitable for conceptual modeling. This is why more abstract modeling approaches have been invented (such as the ER modeling). They provide concepts more closely to the problem domains and, thus, require less effort in system modeling because of their better expressiveness.

However, even these modeling paradigms contain numerous shortcomings that can be categorized into three kinds of *discontinuities* described by Selic et al. [Selic, 1994]:

- Scope discontinuities
- Semantic discontinuities
- Development phase discontinuities

These three kinds of discontinuities, as well as their implications, are explained in the sections that follow.

Scope Discontinuities

These discontinuities are caused by a lack of formal coupling between the representations of different *levels* of detail. The principal representations in traditional high-level modeling approaches attack high-level details, and the associated schemes for representation of low-level details usually follow one of three patterns:

- No specific representation is provided for such details.
- The representation provided is informal.
- The representation provided is formal but is not formally integrated with the high-level representation [Selic, 1994].

In the context of ER combined with DF or another paradigm for modeling functionality, for example, this refers to the levels of detail encountered in a model, from high-level details (such as elements of ER and DF diagrams) and low-level implementation details (such as database tables, referential integrity rules, primitive data types, manipulation of data values, and implementation code in the target language). Although the structural perspective does not suffer severely from this kind of discontinuity, because there is a direct mapping from ER diagrams into the relational model, the functional (DF) part does. Namely, the roles of the elements in DF or other kinds of diagrams are well-defined, but a complete model also requires a description of the logic carried out by each atomic process or activity in a diagram.

The traditional approach in DF and other similar approaches for this purpose is “structured English,” an informal language for logic definition. It is obvious that there is no general formal way to map such specification to a formal implementation programming language. Moreover, there is no formal way to integrate the data-manipulation capabilities of the code with the data-decomposition syntax (for example, how to refer to the parameters of a process or to the elements of a data store from its code).

Semantic Discontinuities

These discontinuities are caused by a lack of formal coupling between the representations of different *kinds* of related detail. It refers to the distinction between details of one kind (such as data structure) and details of another kind (such as functional definition).

In the context of ER combined with DF or another paradigm for modeling functionality, this means that the relationships among the elements of the ER diagram on the one hand, and among the elements of the DF or activity diagram on the other, are well-defined. However, no commonly accepted set of relationships exists between the concepts from these two kinds of diagrams. It is not clear, for example, how the attributes associated with a data store in a DF or activity diagram are related to the attributes of an entity set in the ER model.

Or, even worse, although action languages for ER exist (that is, the languages that provide generic operations that directly work with the entities and relationships modeled by the ER model), they are not used or not available in practical projects. Instead, developers often use ER for conceptual modeling, and sometimes DF or another kind of diagram for designing and documenting functionality at higher abstraction levels. Then, when implementing the behavior, they use their mental potential to conceptually map the semantics of imagined actions upon the ER model to the actions upon the target relational model obtained from the ER model. In fact, they write SQL data modification and SELECT statements to modify and access the relational database in order to implement the specified behavior.

For example, in order to implement the procedure that assigns a student to a course (that is, that creates a relationship between two entities), the developer would think in terms of tables and records, instead of entities and relationships. In other words, the developer would write an SQL query that “inserts a record into the interconnecting table `PersonAttendsCourse`, having the values of the foreign keys equal to the IDs of the given student and course,” instead of saying directly “create a relationship between this student and that course.” Or, similarly, to retrieve all students that attend a course, the developer must write an SQL query that joins three tables over two pairs of foreign and primary keys, as shown earlier, instead of saying this more explicitly.

It should be emphasized how poor this approach is, and how it adversely affects development productivity. First, it is mentally intensive, time-consuming, and error-prone. You can see, even from the given simple example, how much effort and time developers must invest in implementing lots of functions in complex applications applying the given mapping. Additionally, it leads to code that is difficult to understand because of the semantic discontinuity. ER is used for modeling the structure, and the relational paradigm is used to implement the behavior upon the relational model obtained from the initial ER model! Finally, such code is difficult to maintain and modify for the same reasons.

Although this observation may seem strange and overemphasized, it is not. This is really the major drawback of the traditional IS development approaches. To support this claim, it is interesting to compare the described situation with the analogous imaginary situations in some other contexts. Imagine that you were able to define a relational data model in an easy and intuitive way, by defining tables, fields, indices, keys, and so on. But you lack a formal action language for it. For example, in order to insert a record into a table, you must apply a complex calculus of the position of the record on the disk, or perform other low-level data-access operations to store the data. (Actually, this was exactly the problem with early systems that claimed to be relational, but actually were just existing products repackaged with a relational veneer.)

Part I: Introduction

Or, similarly, think about a procedural programming language that allows you to declare a floating-point local variable by a simple and intuitive declaration, but forces you to perform stack calculus and machine-level arithmetic in order to add two such variables! These analogous examples also show that the problem does not need to exist, and indicate the way it can be overcome.

Development Phase Discontinuities

These discontinuities are caused by a lack of formal coupling between representations used in different phases of system development (requirements, design, and implementation). It may be manifested by the use of different notations for requirements definition, design, and implementation. For example, DF diagrams use abstract models of data flow communication, whereas the traditional implementation programming languages use procedure calls as the model of communication. It must designate a calling module and a called module for each data exchange, and the direction of the call may or may not be the same as the direction of the data flow.

Implications of Discontinuities

Because of the described discontinuities, engineers use different paradigms in the development phases. Typically, they use ER alone, or ER combined with DF or another kind of models in the early phases of requirements specification and design phases, and switch to the relational paradigm in the implementation phase. Such an approach, however, has a number of shortcomings.

A significant effort is needed to map the high-level design model into the low-level implementation model. Because of the scope and semantic discontinuities, it is not always possible to perform the mapping automatically. Manual mapping is, however, time-consuming and error-prone.

Consequently, developers also exhibit the “rush-to-code” syndrome: “a pervasive unease during the early development phases, a prevailing attitude among developers that requirements definition and design models are ‘just documentation,’ and a conviction that the ‘real work’ has not begun until code is being written” [Selic, 1994]. Although these developers would claim that requirements and design models provide useful insights into the nature of the system being developed, the more time is invested in building such models, the more uncertainties multiply. This is caused by the lack of objective evidence that the developed model is correct and complete.

As a result, once the development passes to the implementation phase, the requirements and design models remain as documentation artifacts only, without executable semantics or any guaranteed effect on the ultimate executable system. Because the development process is iterative and incremental, the process often calls for modifications. Such modifications on the initial design model do not affect the running system and, therefore, developers are not forced to update it properly. Instead, they update what they merely have to — the implementation (that is, the relational data definition code and the program code), which directly affects their running system. This ultimately leads to inconsistencies between the design model and implementation, which turns the design model into incorrect and, thus, dangerous (or at least useless) documentation. This often forces the development teams to discard the models and stick only to the implementation in later iterations.

User-Interface Development Problems

Although supported by tools featuring automatic form generation and “What You See Is What You Get” (WYSIWYG) design, traditional GUI development also creates a number of problems.

First, typical complex applications often consist of tens or hundreds of forms. Generally, tools automate their design only on a per-form basis, and thus the design of a complex GUI becomes a repetitive, tedious, and time-consuming task.

Another issue is that tools poorly (if at all) automate the design of the repetitive patterns in GUIs. Such patterns are typical for the generic navigation and modification of the underlying structure. A good example of such generic navigation is the navigation to all entities related to the given entity. The traditional master-detail form is a GUI pattern that supports such navigation. However, although tools often support automatic generation of a master-detail form for the given relationship, they do not generate such forms by default, and massively for all, or a subset of relationship sets from the conceptual model. The same holds true for other kinds of concepts used in the underlying structural model.

As a result of the lack of massive automatic generation of forms and other parts of GUI from the underlying structural model, these must be developed manually and usually by teams of developers. Unless strict rules for designing GUI forms are instituted and a detailed GUI design is created in advance, and the application's usability is thoroughly verified afterwards, the members of the development team will diverge in the design, which may lead to a problematic application. For example, some members may easily forget to provide the means to navigate from one entity to other entities over all available relationships in all places in the application, or do it in different (or less intuitive) ways in some places.

Another consequence of all the described problems is that the application's GUI becomes more difficult to test than necessary. The testing is less reliable and also requires more time and effort because the GUI is not uniform and is not generated automatically, so errors may occur in many different places.

Finally, the GUI obtained this way is vulnerable and inflexible. If the underlying conceptual model is modified, the GUI must be updated accordingly in all places, without a systematic and automatic support, or else the application becomes incorrect. More recent software architectures and technologies simply neglect or hide this deficiency. They encourage multitiered software architectures, whereby the business logic (which consists of the domain conceptual model and functionality) is kept clearly separated from the GUI.

Although it is undoubtedly true that this leads to a business layer independent of the GUI and to the flexibility of the GUI, it does not ensure the independence of the GUI from the business layer or reduced vulnerability. Because the GUI is a product of significant human effort and specified through a huge amount of development artifacts (code, forms, and so on), sometimes a slight modification of the underlying conceptual model may require significant work to update the GUI. This problem becomes even greater in the contemporary systems where different GUIs coexist (for example, a thick-client GUI, a thin-client Web-oriented GUI, a PDA GUI, and so on).

Section Summary

- From the developer's perspective, the relational paradigm is good because it is simple, formal, mature, and well supported by DBMSs.
- On the other hand, the relational paradigm is conceptually too simple and far from the problem domain.

Continued

- ❑ The problems with the traditional high-level modeling paradigms such as ER combined with DF, activity, or another functional modeling paradigm, are three kinds of discontinuities:
 - ❑ Scope discontinuities are caused by a lack of formal coupling between the representations of different levels of detail.
 - ❑ Semantic discontinuities are caused by a lack of formal coupling between the representations of different kinds of related detail.
 - ❑ Development phase discontinuities are caused by a lack of formal coupling between representations used in different phases of system development (requirements, design, and implementation).
- ❑ The discontinuities cause the “rush-to-code” syndrome, which turns high-level models into incorrect documentation of the low-level implementation.
- ❑ Development of user interfaces also offers its share of challenges:
 - ❑ Although supported by tools at the level of a single form, development of tens and hundreds of forms in typical complex applications is a repetitive, tedious, and time-consuming task.
 - ❑ Tools and frameworks do not automate creation of forms and GUI controls that support repetitive patterns such as generic navigation and modification of the underlying structure. All of these must be developed from scratch.
 - ❑ The GUI developed is often vulnerable and inflexible. When the conceptual model is modified, the GUI must be modified manually in all affected places.
 - ❑ This approach does not enforce uniform and intuitive use, but it must be ensured by strict usability policies and verification.
 - ❑ Testing of a GUI is more complex and less reliable than necessary.

3

The Object Paradigm

Although it emerged more than 40 years ago, the object paradigm (and the supporting technology) has become the mainstream in software development in the past decade or so. It has been successfully applied to almost all application domains, and many people find it to be better suited to certain domains than other paradigms. However, apart from the evident advances, the current object technology for building information systems suffers from mostly the same limitations as has been described for traditional approaches.

I expect that many (but certainly not all) readers will be familiar with the fundamental concepts of the object paradigm, having at least some (or even very extensive) experience in object-oriented (OO) programming with one or more OO programming languages. All of these fundamental OO concepts and principles (but with their concrete meaning tailored for the context of this book) will be thoroughly explained later. However, those readers without prior knowledge in object orientation, or without experience with any OO programming language, are encouraged to read the tutorial in Chapter 24 of the Supplement. That tutorial briefly describes the fundamental principles of the object paradigm, and provides a quick overview of how popular OO programming languages support these principles.

This chapter introduces OO modeling as the approach of a higher level of abstraction than programming in a classical OO programming language, bringing answers to some of the issues that may arise in OO programming. It then introduces UML, the modeling language central to this book. Finally, it discusses the characteristics and issues of the current state-of-the-art of OO technology in building information systems, the characteristics and issues motivated the approach presented in the book.

Object-Oriented Modeling

Using only OO programming as the means to develop software has turned out to be insufficiently effective for engineering complex systems. Some of the most important reasons for this include the following:

- OO programming does not offer effective means for specifying, organizing, and documenting complex software systems. More expressive concepts are needed for decomposition of software systems.

Part I: Introduction

- ❑ The concepts of OO programming are good, but in some cases not conceptually close enough to the problem domain. In other words, they are sometimes still too close to the implementation technology (that is, computers), which makes them insufficiently expressive to specify problem domain concepts. First, more expressive relationships among classes than those supported by OO programming languages may be useful. For example, the notion of relationship from ER, as a bidirectional link between entities (or objects), is not directly supported in OO programming languages. Instead, in order to implement a bidirectional conceptual relationship, you must deal with unidirectional references or collections of references in the classes at both sides and keep them synchronized. (That is, if a reference from an object *a* points to an object *b*, there should also exist an opposite reference in *b* pointing to *a*). This is a common source of errors, or at least a policy that must be pursued by the programmers. Furthermore, the scenarios of interactions among objects are too deeply hidden and spread in operation bodies to be easily visible and understandable. You must trace through the textual code of usually many operation bodies in order to understand a mechanism of object interaction. Furthermore, there are a number of useful OO concepts that are not supported directly by existing OO programming languages.
- ❑ Textual representations of software systems are often less clear than visual (diagrammatic, pictorial) ones.
- ❑ OO programming does not fully support all phases of software development, especially not the earlier phases (requirements capturing and design). Consequently, it suffers from the phase discontinuities because transitions from earlier to later phases are not smooth.
- ❑ OO programming does not provide good enough instruments for documenting design decisions and software architecture.

As a result, methods for OO modeling have emerged as a response to the listed problems of OO programming. Generally, the methods for OO modeling include the following characteristics:

- ❑ Developing models of software systems at a higher level of abstraction, closer to the problem domain, and using more abstract and expressive concepts than those supported by some OO programming languages. Again, as before, OO modeling is not a revolutionary approach that has replaced the core concepts from OO programming. On the contrary, OO modeling approaches retain all core concepts from the object paradigm, but also introduce some new ones, unknown to OO programming.
- ❑ Model specification using visual (graphical, diagrammatic) notations, combined with text.
- ❑ Automated transformation of abstract models into implementation forms (that is, transformation supported by computer-based automation, whether it be complete or partial). The implementation forms include (but are not limited to) the code in OO programming languages. Depending on the OO modeling language, the transformation may or may not be performed completely automatically.

Section Summary

- ❑ Using only OO programming as the means to develop software has turned out to be insufficiently effective for engineering complex systems.

- ❑ Generally, methods for OO modeling assume the following:
 - ❑ Developing models in more abstract languages that support the fundamental OO concepts, as well as concepts not supported by common OO programming languages
 - ❑ Specifying models using visual notations combined with text
 - ❑ Automated transformation of models into implementation forms, most notably into code in OO programming languages

The Unified Modeling Language

Starting from the mid-1970s and during the 1980s, researchers experimented with different languages and methods for visual modeling of OO systems. Because they understood the shortcomings of the OO programming approach, they invented a lot of useful concepts, proposed notations, and established processes for designing complex OO systems. Many of them had predominantly in mind the development of information systems. The development of the methods ended with more than 50 different approaches to OO analysis and design in the early 1990s. The authors of new methods promoted them through books that described the methods, notations, and processes for developing OO software systems.

Although the diversity of methods had a useful impact on the evolution of OO technology, the software engineering community appealed for unification of the ideas and notations of the proposed methods, and for a standard approach that could be widely used in different problem domains. This demand was quite understandable. The lack of unification caused severe problems in learning and applying OO modeling techniques, interchanging ideas among researchers and practitioners, and in support and interoperability of development tools. The situation threatened to jeopardize further development and application of OO modeling languages and methods.

As a result of these pressures, a unified OO modeling method emerged in the mid-1990s. It was based on the methods and work of three authors (with their coauthors): *OO Analysis and Design* by Grady Booch [Booch, 1994], *Object Modeling Technique (OMT)* by James Rumbaugh and his colleagues [Rumbaugh, 1991], and *OO Software Engineering (OOSE)* by Ivar Jacobson and his colleagues [Jacobson, 1992]. The “three amigos” (as the three distinguished authors became known) gathered in Rational Software Corporation and started working on the *Unified Method*. However, they quickly realized that a method should consist of a modeling language, which should be standardized as a common means of communication, and a process of developing software, which should not be homogenized, but should be customizable for particular problem domains, applications, organizations, development teams, and so on. This is why they decided to work only on the language itself, which they named the *Unified Modeling Language (UML)*.

The first version of the language (UML 0.8) was published in 1996. The response was collected from a very wide industrial community, and important companies took part in polishing the next version of UML. The UML consortium, established for this purpose, submitted UML 1.0 to the Object Management Group (OMG) for standardization. OMG adopted UML 1.1 as a standard in November 1997. From then on, UML has been a public standard under supervision of OMG.

Version 1.x of UML suffered from a significant lack of formalization and preciseness of its semantics. This was recognized as the major shortcoming of UML. One major step in curing this problem was the definition of the semantics of actions in UML 1.4 released in September 2001. However, this was not good enough. The community appealed for a major revision of UML, especially its semantics.

Part I: Introduction

As a result of that campaign, a major revision appeared in UML 2.0, which was finalized in October 2004. After that, several minor revisions were published by OMG. As of this writing, the last adopted version of UML is 2.2, released in May 2008.

Section Summary

- ❑ UML is a standardized, object-oriented, visual language for modeling software-intensive systems.
- ❑ UML is used for specifying, visualizing, constructing, and documenting software systems.
- ❑ UML emerged after the wide appeal from the software engineering community to unify numerous OO modeling approaches that had previously existed.

Characteristics of UML

In summary, UML is an object-oriented, standardized, and widespread modeling language intended predominantly (but not exclusively) for modeling software systems. As such, it has many important advantages:

- ❑ **UML is standardized and widely accepted today.** It is a common language for developing and interchanging models of software. Most researchers and practitioners use it to describe their ideas and designs. It is taught in virtually all software engineering curricula at universities worldwide. Additionally, it is now supported by most available commercial software development tools.
- ❑ **UML is conceptually rich.** Many interesting concepts have been incorporated into UML as a result of practical needs. This is why UML can be used for very different application domains, such as information systems, interactive desktop applications, command-and-control systems, telecommunication systems, embedded and real-time systems, Web applications, banking and financial applications, business systems, scientific applications, and many others. UML can even be used for modeling hardware configurations or business processes. However, it is predominantly targeted for modeling software. In all, UML is a general-purpose language for modeling software-intensive systems.
- ❑ **UML is object-oriented.** It supports all fundamental concepts of the object paradigm. It also avoids most drawbacks of the OO programming level. Additionally, it has a diagrammatic notation, which makes models possible to visualize.
- ❑ **UML is extensible in a standardized, controlled way.** It is possible to customize the language for each particular application domain or concrete problem. This is an important aspect of UML that enlarges its scope and makes it valuable for a longer time.

However, in spite of the evident important benefits, UML, in its current state, has several severe drawbacks:

- ❑ **UML is very complex and heterogeneous.** This is a result of the great interest of contributors from very different and contrasting domains in incorporating concepts needed for modeling in

different problem domains. On one hand, this is good, because, as stated, UML can be used for very different problem domains. On the other hand, however, this makes it too complicated for many people. To be precise, it is not the intention of UML to use all its concepts in every particular application domain. Of course, you should use only those concepts that you need for a particular application. However, in order to be able to read and understand models made by others who may likely use different concepts of UML, you must be familiar with the entire language. Although the inventors of the language claim that they have achieved a balance of the two opposite requirements (to make a conceptually rich and widely applicable language, yet to keep it simple enough), many people think that this balance wasn't achieved. An ambitious user of UML must surmount a steep learning curve to adopt the entire language.

- ❑ **A great (or, better to say, the larger) part of UML does not yet have precise semantics.** In other words, UML is still far from being completely formalized. There are a number of reasons for this; some of them are completely justifiable, but some of them are not:
 - ❑ Some parts of UML are not intended for precise, formal modeling. Namely, UML is aimed at all phases of software development, and also at early phases of requirements modeling, when analysts have only vague and incomplete ideas about the system they are constructing. For this reason, UML allows incomplete, inconsistent, and informal models to be created in early phases, which might be refined in later phases. Such parts of UML should not (and will not) have formal semantics.
 - ❑ Some areas of UML are too vague by their nature, so it is very unlikely that they can ever have formal semantics. They often have incomplete, inconsistent, ambiguous, derived, or completely void meaning. The usability of such parts and their survival in UML is highly questionable.
 - ❑ The semantics of some parts of UML are not yet defined, either because of the lack maturity or a complete understanding of their purpose, or because language designers want to keep them open for customization for specific domains or for specific mappings to implementation languages. Namely, one of the interests of the language designers was that code in the traditional OO programming languages can be generated from UML models. Because these languages differ in the semantics of some aspects, the semantics of the counterparts of those aspects in UML are deliberately left undefined. In other words, their semantics in UML are interpreted depending on the mapping to the target implementation. However, this is not good because it leaves the described semantic discontinuities in modeling, as it was with traditional IS development approaches. The semantics of such parts can and should be defined, either generally, or specifically for a certain application domain. The lack of formal semantics prevents UML from being executable.

A recent initiative of OMG is the work on the *Executable UML Foundation*, which was adopted in June 2008 and contains not only precise specification of a core subset of UML semantics (in effect, a UML virtual machine), but also a mathematical formalization of those semantics [EUF]. Its aim is to alleviate some of the previously described problems of UML.

The approach described in this book is a result of an independent and simultaneous work that is similar with the Executable UML Foundation in many aspects, most notably in its idea of the formalization of the run-time semantics of a selected part of UML. However, the approach presented in this book is tailored for one particular, yet very broad, domain of applications, and brings other concepts that are suitable for modeling in that domain.

Section Summary

- ❑ UML has several significant advantages:
 - ❑ It is standardized and widely accepted nowadays.
 - ❑ It is conceptually rich and applicable to many different problem domains.
 - ❑ It is object-oriented and visual.
 - ❑ It is extensible and customizable.
- ❑ The drawbacks of the current state of UML include the following:
 - ❑ It is too complex and has a steep learning curve.
 - ❑ Many parts of UML do not have precise semantics and, as a result, UML is not formal and executable.

Profiling UML

The described drawbacks can be mitigated substantially by *profiling*. A *profile* of UML is a specialization of standard UML for a certain problem domain, with the intention to do the following:

- ❑ Narrow UML by cutting out some concepts that are unlikely to be necessary in the problem domain, thus reducing its complexity and alleviating the learning curve.
- ❑ Extend UML in a controlled way by introducing some other concepts and libraries of models specially needed in the domain.
- ❑ Formalize the semantics of the added and existing parts of UML, as it is suitable for the domain.

The use of profiles may alleviate the three kinds of discontinuities otherwise present in modeling with UML. It allows UML to become executable and, thus, particularly useful for a problem domain, without jeopardizing the general character of the language and spoiling its native flavor or reducing its wide understandability. Making a profile of UML for modeling information systems is actually the core goal of this book.

Section Summary

- ❑ A solution to UML's drawbacks is offered by UML profiles. A UML profile is a specialization of the language for a certain problem domain, which cuts out some of its parts, extends it with specialized concepts, and defines formal semantics for some of its parts.

Traditional OO Development Approach

In its current state, UML still has all the described shortcomings — most notably, the lack of completely formal semantics of many of its important parts. The semantics of actions upon structure in UML, although already standardized, have not been widely supported by development tools and run-time environments. In such circumstances, a typical approach to developing OO software (including information systems) is much as it appears in the following discussion.

UML models are built as informal design specifications of software that will be constructed in later phases, rather than as executable artifacts that may serve as early and objective assessments of design decisions. In other words, instead of being the first implementation products of the development process, models are regarded as design sketches of the products of later development phases. This is partially a result of the fact that UML has its roots in the methods that regarded OO modeling just as a kind of “drawing pictures” of OO programs that designers have in mind and leaving the formal semantics to the implementation languages and particular interpretation. This approach is quite understandable. Engineers have fewer problems understanding new advanced concepts if they raise their abstraction level bottom-up, in small increments. This is why they very often design their models by “thinking in C++, Java, or C#, and drawing UML diagrams just as pictures of their thoughts and sketches of the design.” In other words, UML is used to “draw the code of an OO program” so that its design can be more easily understood.

Moreover, it is often the case that UML concepts are explained (in textbooks and courses on UML) by showing how they can be mapped into OO programming language concepts. This is again reasonable because people tend to understand new concepts more easily if they can conceptually map them into something that they already recognize.

When a model is filled with enough details, it is translated into an implementation form — that is, code in an OO programming language, a database scheme, or something else. Because of the lack of formalism, this translation can be only partially automated. Usually, the structural part of a UML model is mapped into the declarative part of the OO program and/or the database scheme. However, the behavioral part remains mostly unmapped. The programmers complete it manually in the target program.

This task is the toughest for developers. First, for example, the persistence of objects is usually supported by the programming framework such as Hibernate or Enterprise Java Beans (EJB). The developer must integrate the generated program into the framework with or without the help of the tool. Furthermore, the generic operations (actions) upon the higher-level structural concepts are usually not supported directly, but the programmer must implement them for each particular case. They are often implemented using the direct access to the (usually relational) database — that is, by operating upon data records, or at the target OO programming language level, without tight coupling with the source UML concepts in either case.

Although often similar, the concepts in UML and in the target OO programming language sometimes have different semantics and a logical mapping is needed. This is again performed by the developer. One typical example is the concept of association (the UML counterpart of a relationship set in ER), which must be implemented by synchronized unidirectional references in participating classes. Their synchronization (and a lack of it) is completely up to the developer, so it represents a source of errors.

In addition, object-to-relational persistence frameworks, such as Hibernate or EJB, spare the developer from having to think about mapping concepts of the OO programming language to the relational model,

Part I: Introduction

but often require the developer to think about or do more things than are necessary. For example, they often insist on defining primary keys (even when they are technical IDs irrelevant for the problem domain), unnecessarily emphasizing the fact that the programming language's object space is a working subset (or cache) of the entire space stored in the database, and making it explicit to the developer. For example, they might require the explicit load/store operations on objects to synchronize the two, and many other things.

Moreover, problems occur when some design decisions are incorrect or inadequate. Such conclusions may be drawn pretty late in development, as late as the first running implementation is constructed, and long after the model was created. Instead of going back to the source UML model, many developers tend to apply the modifications in the target program only. This leads to inconsistencies between the model and the code, and makes the model useless. Reverse-engineering methods that help discover changes in the implementation and update the model correspondingly have just a limited value, again because of the lack of formal coupling of the two levels of artifacts.

Because of the described problems, developers again exhibit the rush-to-code syndrome. As in the traditional non-OO approaches, the main work remains for the implementation phase. The assessment of the validity of the model is achieved too late, when the implementation is near completion, and long after the model was created. It is obvious, therefore, that *the rush-to-code syndrome cannot be mitigated by merely replacing a traditional non-object paradigm with the object one.*

Unfortunately, many problems with GUI development described for the traditional non-OO approaches still persist. The GUIs are still predominantly based on forms generated by the tools or made by hand one by one. Many of the popular tools still bind their form generators and GUI controls to the relational model instead of OO models. Even if they do not — that is, if a GUI control relies on object-oriented underpinnings (for example, a tabular or a tree view control expects a collection of objects in an OO programming language to render) — the developer must deal with loading the data from the database, creating the necessary object structure, and passing it as the input to the control. In other words, in order to render a set of data in a tabular form, the developer must transform the data returned by a relational query that is inherently organized as a table, into a collection of objects, which will then be rendered as a table by the GUI control.

This often looks like a completely unnecessary transformation of the data between structures organized in different modeling paradigms, just for the sake of “orthodox object orientation” of the GUI and application, introducing unnecessary development effort and execution overhead. It is sometimes really faster to introduce a GUI control that is directly bound to the relational data structure, instead of dealing with objects in memory that should be handled only for the purpose of resolving the impedance mismatching of the available database structure and the GUI control. This is why it takes an enormous amount of effort and takes a long time to build GUIs.

To some extent, the described OO development approach requires even more effort from developers than was necessary with the older ones (such as the relational paradigm). Now, developers still must be experts in relational databases, which is not completely transparent to them. And apart from thinking about the relational model, procedural code, and form-oriented GUIs, they now must take into consideration yet another layer, programmed in a traditional OO language. In effect, it is just as if the ER/DF modeling simply has been replaced with UML diagrams, but all other problems remain. In addition, because of the scope and semantic discontinuities, the different levels and kinds of details are intermixed in a confusing manner (for example, SQL statements that access the relational database are written within polymorphic operations of Java classes).

This is why many practitioners that design and build information systems have been disappointed with the OO technology and model-based approach, even after an initial fascination with it, and are getting back to the familiar relational development paradigm. However, they are wrong to be disappointed with the entire object paradigm. The object paradigm is truly beneficial and expressive, but the actual problem is with the supporting technology and the traditional development processes that are not mature enough. In fact, the described UML-based modeling approach also suffers from all three kinds of discontinuities that lie in the heart of the described problems:

- ❑ Scope discontinuities exist because the level of details (for example, implementation of operations) is usually incorporated in the target program and at the OO programming level, with the direct access to the database records, and not at the higher level of UML concepts.
- ❑ Semantic discontinuities exist because of the lack of formal semantics of many UML concepts.
- ❑ Development phase discontinuities (although much smaller because UML is used in both requirements engineering and design phases) still exist because the implementation is done in a traditional OO programming language, loosely coupled or often completely uncoupled with the source UML model.

Section Summary

- ❑ The traditional UML-based development approach also suffers from the three kinds of discontinuities:
 - ❑ Scope discontinuities exist because the level of details (for example, implementation of operations) is usually incorporated in the target program and at the OO programming level, with the direct access to the database records, and not at the higher level of UML concepts.
 - ❑ Semantic discontinuities exist because of the lack of formal semantics of many UML concepts.
 - ❑ Development phase discontinuities (although much smaller because UML is used in both requirements engineering and design phases) still exist because the implementation is done in a traditional OO programming language, loosely coupled with the source UML model.
- ❑ The rush-to-code syndrome cannot be cured by merely replacing a traditional non-object paradigm with the object one, without resolving the discontinuities.

Desired Characteristics of Object-Oriented Information Systems

As a conclusion from the discussions on advantages and drawbacks of the traditional relational and object-oriented development approaches, let's summarize the characteristics of information systems that

Part I: Introduction

are desired to alleviate the identified shortcomings. In essence, the desired approach should have the object paradigm at its core. The object paradigm should be exposed to the user in a proper manner. The modeling approach should also overcome the three kinds of discontinuities in order to be effective. As a summary, let's look at the desired characteristics again from the two perspectives: the user's and the developer's.

Usability Aspects

The application's look-and-feel should consistently expose the object paradigm to the user through the entire user interface, including its presentational and behavioral parts. Actually, the user interface of OO information systems may resemble the interfaces of other interactive applications (such as, for example, graphical editors, graphical shells of modern operating systems, e-mail clients and personal organizers, or file system explorers), which have already established modern standards of usability. An example is shown in Figure 3-1. Of course, whenever it is really suitable, the traditional GUI approaches may be used, too.

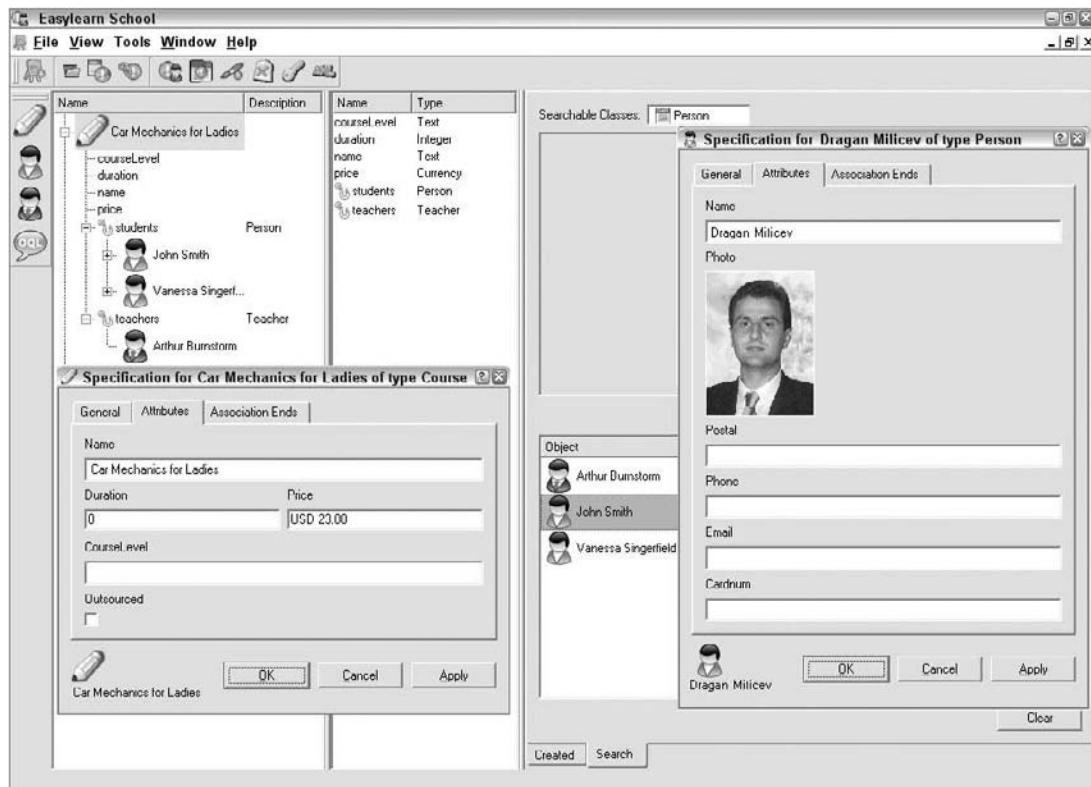


Figure 3-1: An example of an intuitive GUI that reveals the underlying object paradigm to the user.

But what does object orientation really mean from the perspective of users? Here are some manifestations of the object paradigm that can be visible to the user, as illustrated in Figure 3-1 for a sample school information system:

- ❑ The user should have a feeling that he or she works with *objects* instead of *data*, whenever appropriate. Objects, from the user's perspective, are tangible things that have some meaning, role, and purpose in the system, and directly represent some domain-specific notions. For example, objects may be represented with icons that can be clicked, double-clicked, or drag-and-dropped. In the sample school information system, courses and people may be represented with icons. However, other things in the system may also be represented with icons and exposed as objects. For example, a command to delete an object from the system may also be an object, or a query that returns all students in a course that satisfy a certain condition may be an object.
- ❑ Objects are things with properties that may be retrieved or modified. For example, the user interface may provide a way to show a form with all properties of a person (name, postal address, and so on), a form with input/output controls for all properties can be modified.
- ❑ There may be conceptual relationships among objects. For example, a person may attend a course. The relationships among objects should be established or discarded in an easy and intuitive way (for example, by dragging an icon of a person and dropping it on an icon of a course). Whenever possible and suitable, the user should be spared from entering foreign keys or selecting data from lists, if the required entity is already available on the screen (for example, as an icon).
- ❑ The user should perceive the inheritance relationship with its essential meaning. Whatever is applicable to objects of an ancestor type should also be possible for the objects of the descendant type. For the given example, all described manifestations for people should also hold for teachers, being special kinds of people.
- ❑ Objects are things that exhibit some behavior, that is, they are things on which some operations can be applied. For example, each object may have a list of appropriate operations that pops up on a right-click on the object's icon. The list of operations, of course, depends on the type of the object. As already stated, even operations may be presented as objects, and an icon of an operation can be simply dragged and dropped onto another icon, meaning that the operation should be applied on the target.
- ❑ Operations should be polymorphic. This means that the actual response to activation of an operation should depend on the concrete type of the object. For example, the response on a double-click on an icon of an object depends on the type of the object, or selections of the same operation from possibly the same list of available operations for a person and for a teacher may have different effects.

When all these manifestations are present, the ultimate effect is that users have a feeling that they work with *actual objects*, which are tangible things that have their properties and behavior, instead of *passive data*, represented with records of values that are just (sometimes artificial) extractions from the real problem. The users' impression of dealing with objects in the system ensures their proper association with the entities from the real world that are direct counterparts of the objects that represent them. More precisely, it is particularly important that the form in which the data is represented (icons, pictures, reaction on users' actions, and so on) enables more direct association with the domain-specific notions captured by the data.

Part I: Introduction

Of course, this is not an approach that must be exclusively used in all circumstances. As already mentioned, probably the quickest way to request a money withdrawal is to fill in a simple form by entering the bank account number and the money amount — drag-and-drop or icons are not needed for that. The point of this discussion is not to suggest complete replacement of traditional forms. On the contrary, they should be used whenever they are the best way to accomplish the task. The point is to indicate that the forms are not the best thing for all circumstances, and that there are many cases in which the GUI should provide the OO perspective to the workspace.

Section Summary

- ❑ In many circumstances, the application's user interface and behavior should expose the object paradigm to the user.
- ❑ Users should have a feeling that they work with *actual objects*, which are tangible things that have their properties and behavior, instead of *passive data*, represented with records of values that are just (sometimes artificial) extractions from the real problem.

Development Aspects

The development approach should support easy and efficient achievement of the listed manifestations. It would be very inappropriate if developers needed to program each particular application from scratch, using the traditional OO programming methods and frameworks, in order to obtain a powerful and intuitive application described in the previous section.

To obtain this, the development approach must fulfill several important requirements. First, the modeling language must be expressive, formal, and executable. The models built in the modeling language must be concrete executable products of the development process. The language must, therefore, have only the necessary concepts with completely clear semantics. The unclear concepts with imprecise semantics must be either removed from the language, or carefully isolated and used for particular purposes only (such as for early phases of requirements analysis).

In other words, the modeling language must resolve all three kinds of discontinuities:

- ❑ **Scope discontinuities** — The detail level must be carefully incorporated into and coupled with the higher-level modeling concepts.
- ❑ **Semantic discontinuities** — Different parts of the language used for modeling different aspects of the system must have clear semantics and must be formally coupled with one another.
- ❑ **Development phase discontinuities** — The same modeling language must be used throughout the entire process.

As a result, the development approach with such characteristics would allow *rapid prototyping*. The models built in early iterations may be executed to assess the design decisions and eliminate risks early enough. The described user interface should be realized very quickly, with least possible effort. For example, the generic operations upon the object structure should be supported generically, as DBMSs

do for the relational model. The conceptual model should carry enough information for such generic prototyping.

In general, the largest part of the GUI that is repetitive and is implied by the application's model must be obtained automatically from the model, and only some specific parts may be configured manually, but in an intuitive and efficient way. (This automated production of a skeleton of the application's GUI from its model is sometimes called *scaffolding*.) The development approach must also encourage an iterative and incremental process of development, especially of the GUI and behavior initiated from the GUI. The operations issued from the GUI must be configured in an intuitive way. The approach must also support efficient customization of a prototype or generic GUI in order to get the desired application quickly. The GUI obtained this way must not be too vulnerable to the modifications of the conceptual model.

Finally, the modeling language must be expressive enough to support all situations typical for information systems. It should treat concurrency and distribution in a way that would allow different implementations on present and future distributed architectures, whereby parts of an application and data may be deployed on different computers. It should also support fault tolerance in a proper manner. It should support preservation of the system's consistency, as well as querying of the object (not only data) space, again according to the object paradigm.

Section Summary

- The modeling language should be formal and executable, and should resolve all three kinds of discontinuities.
- The modeling language should allow rapid prototyping with early assessment of design decisions and elimination of risks.
- The modeling language should support all other important issues, such as concurrency control, distribution, fault tolerance, consistency checking, querying, and reporting, according to the object paradigm.

The Rest of This Book

The goals set forth in the few preceding sections are the subject of the method described in the rest of this book. The method is actually based on a profile of UML for object-oriented information systems (OOIS), called the *OOIS profile of UML* (OOIS UML for short).

The next part of the book provides a quick overview of some basic concepts and approaches used in OOIS UML. You will gain an overall idea of what OOIS UML consists of, how it mitigates the problems described previously, and how it significantly improves productivity of information system development.

Part III of the book provides a thorough and complete definition of the language itself (that is, of the concepts of OOIS UML). The concepts are described in detail, with their complete semantics.

The definitions in these parts are provided without relying on the semantics of elements of the traditional OO programming languages. Although this may seem strange to you, the idea is to make you forget

Part I: Introduction

about the semantics of traditional OO programming languages or the relational paradigm you might be familiar with (or at least to move them to the background). Although this knowledge may help, it is not necessary for understanding (and, in some cases, may actually impede it). Namely, the goal of this presentation is to introduce a complete and consistent logical system from scratch, without explaining the semantics of new concepts by explaining how these concepts can be mapped to the concepts of a lower level of abstraction. (That is, by explaining how they can be implemented with the existing technologies, such as OO programming languages or relational databases.)

Although it is completely natural for people to try to understand a newly introduced concept by trying to map its semantics to something already known, you are encouraged to try to grasp the presented concepts as they are given and explained, in a completely abstract and formal way. Such an approach will help achieve the main goal — accepting a new language with its complete semantics, so that the executable application can be imagined from the specified model and taken as the key manifestation of the semantics. Simply put, the modeling language presented in the book should *not* be treated as yet another way to “draw the design of a relational database or code to be written in an OO programming language,” which is the major misconception of the traditional approach.

Part IV of this book describes some elements of the design process proposed to be used with OOIS UML. This is not a cookbook recipe for how to get to a successful information system. It is just guidelines and suggestions that may help you achieve that goal.

Section Summary

- ❑ The goals set forth in the few preceding sections are pursued by the OOIS profile of UML described in this book.

Part II

Overview of OOS UML

Chapter 4: Getting Started

Chapter 5: Basic Language Concepts

Chapter 6: Interaction and Querying

4

Getting Started

This chapter quickly introduces OOIS UML by examining its main features and describing its organization and constituent parts.

Key Features of OOIS UML

Let's begin by taking a very quick glance at what can be done with OOIS UML. At this point, let's not dig in to any of the concepts or mechanisms of OOIS UML, or even try to understand how the things work. Instead, let's take a quick look at *what* can be done with OOIS UML, not precisely *how* it can be done. Of course, these details will be given later in this book. In particular, Chapters 5 and 6 provide an overview of the concepts and mechanisms. Throughout this part of the book, you will see a simple and illustrative example of a small information system for an imaginary school named Easylearn.

The idea of model-driven engineering is to develop software based on models. Figure 4-1 shows a simple UML model for the Easylearn school system. Without any additional explanations of the meaning of its elements, let's say that it models the key concepts from the problem domain, such as *courses*, *persons*, and *teachers*, as well as their properties, such as *name*, *duration*, *address*, and so on. In addition, it models the relationships between the key concepts, such as the fact that a person can *attend* courses, or a teacher can *teach* courses.

Without writing a single line of code, a running application with the results shown in Figure 4-2 can be obtained right from the model.

In particular, the application exhibits the following features:

- ❑ The user deals with *objects* that represent the physical or conceptual things from the problem domain. The objects are instances of the corresponding concepts from the model, such as students, courses, and teachers. They can be created and destroyed (for example, interactively from the GUI, or programmatically from the implementation of the specific business logic).

Part II: Overview of OOIS UML

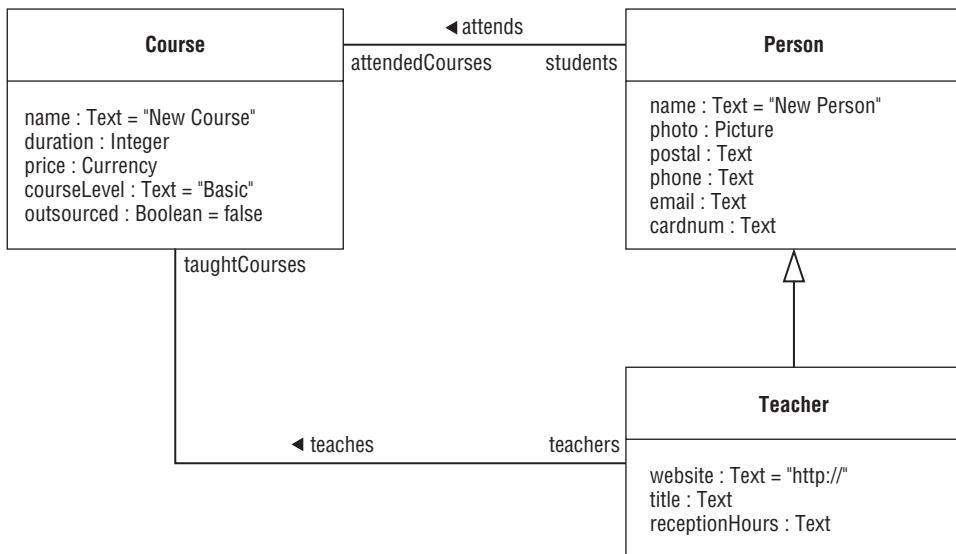


Figure 4-1: A sample UML model for the Easylearn school system



Figure 4-2: The appearance of the prototype application obtained from the sample UML model for the Easylearn school system

- ❑ The objects inhabit a potentially huge, inherently persistent object space managed by the system. Modelers or developers do not need to ensure their persistence by any explicit mechanism, nor do they need to be aware of the relational or any other lower-level database paradigm.
- ❑ The objects hold values of their properties, as specified in the model. These values can be retrieved and modified (for example, interactively from the GUI, or programmatically from the implementation of the specific business logic).
- ❑ The conceptual connections between objects can be established and removed in the application (again, interactively from the GUI, or programmatically from the implementation of the specific business logic). For example, a person can be associated with a course as a student.
- ❑ The objects can exhibit *behavior* that can be activated from outside — for example, from the GUI or by calls from other objects (in the implementation of the specific business logic).

Chapter 5 describes the basic OOIS UML language concepts and their semantics that provide such effects. The capability to get the running application right from the initial model, without the need for classical programming, represents the executable nature of OOIS UML.

The GUI obtained from the model represents a quick prototype (or a *scaffold*) of the target application. It can be further tailored for the particular purpose. For example, the appearance and behavior of objects can be customized at run-time, without any coding. That is, the icons used for objects, as well as the commands available in the right-click pop-up menu and their association with the application's behavior, can be specified by interactive configuration of the application, again without classical programming. In addition, the behavior activated on a user's drag-and-drop can be specified by simple *demonstration*. For example, in order to define that dropping an icon of a person onto an icon of a course would mean associating that person as a student of the course, the developer can simply do the same action, and the run-time environment will offer this option because it is implied from the particular relationship in the model.

Chapter 6 describes the basic OOIS UML concepts related to the presentation and interaction within the application, and the mechanisms of customizing the GUI at run-time and by demonstration. Chapter 6 also describes the OOIS UML capability of posing complex queries to search for sets of objects using the object paradigm.

Section Summary

- ❑ The idea of model-driven engineering is to develop software based on models.
- ❑ Key features of an application obtained right from an OOIS UML model include the following:
 - ❑ The user deals with *objects* that represent the physical or conceptual things from the problem domain.
 - ❑ The objects inhabit a potentially huge, inherently persistent object space managed by the system.

Continued

- The objects hold values of their properties, as specified in the model.
- The conceptual connections between objects can be established and removed in the application.
- The objects can exhibit *behavior* that can be activated from outside.

The Organization of OOIS UML

OOIS UML, in a broader sense, is a model-driven development method that consists of the following elements:

- Language** — The language includes the definition of the concepts, along with their properties, relationships, and semantics. It also includes the *abstract syntax* that defines the rules of how the elements of the language can be combined, as well as the *concrete syntax* that defines the (diagrammatic or textual) notation for the language. The OOIS UML language is defined as an executable profile of UML, as discussed in Chapter 3.
- Model library** — This is a library that consists of some reusable, general-purpose models, built in the OOIS UML language, available to modelers and applicable to all specific domains and applications. One of the notable parts of the OOIS UML model library is the model for configuring and customization of the GUI that will be described in Chapter 6.
- Run-time environment** — This is the execution engine that provides the services of maintaining the object space to the running application. It ensures the object persistence and provides *actions* upon the object space with the full OOIS UML semantics. It frees the modeler of the burden of dealing with the underlying database in the relational or another paradigm.¹ It does so by providing the application programming interface (API) to the domain-specific application (or, more precisely, to the implementation of the specific business logic). For this reason, it can be also thought of as an OOIS UML *virtual machine*.
- Design process** — This is a set of guidelines and hints of how to apply the language to building information systems. It will be briefly described in Part IV of this book.

¹Note that this observation does not prevent the run-time environment from exploiting the services of an underlying relational (or another kind of) database. In other words, the run-time environment has a role similar to a traditional object DBMS (although with specific OOIS UML semantics), but it does not have to be implemented from scratch. A concrete implementation can likely rely upon an underlying relational database, but this is a matter of implementation and is completely hidden from the modeler. The modeler sees the object space with the OOIS UML run-time semantics, while the run-time environment can use a relational database as the actual storage.

Figure 4-3 shows a block diagram of these components of OOIS UML. The components that are part of OOIS UML are filled in gray. The white components represent domain-specific parts — that is, the parts developed for a particular information system.

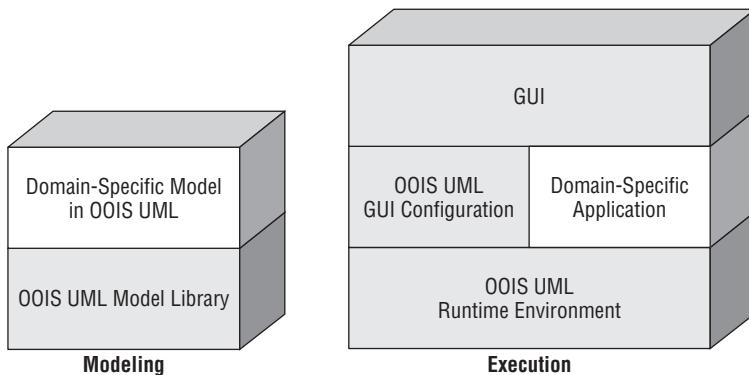


Figure 4-3: A block diagram of the components of the model and of the running information system build in OOIS UML.

On the left side are the components related to modeling. The model of the specific problem domain can (and typically does) use the concepts available in the OOIS UML model library. The elements of that model library are also sometimes called the *built-in* elements of OOIS UML.

On the right side of Figure 4-3 are the components of the executing system. As already explained, the OOIS UML run-time environment provides the services to the domain-specific part of the system, as well as to the GUI, especially to its part that enables the dynamic configurability of the GUI at run-time. The GUI can be fully generic (as obtained directly from the model and supported by the selected implementation), further tailored and customized, or completely developed from scratch for the particular system. The chapters that follow use one possible form of a generic GUI for demonstration. It is referred to as the *generic OOIS UML GUI*. A particular implementation, however, can provide a completely different appearance of the GUI, on a different platform (for example, the Web), or adapted for a particular third-party GUI framework.

The (major part of) models built in OOIS UML are formal and executable. When a model is developed, it can be executed within the run-time environment, after a possibly needed compilation, such as a standard computer program. (The compilation can — but does not have to — mean an automated transformation of the model into the code of a traditional OO programming language, which is then referred to as the *implementation* or *host language*. This is again a matter of implementation and does not, by any means, imply that the system is modeled using the semantics of that language.) As already described, the OOIS UML run-time environment provides a generic GUI that enables interactive manipulation of the object space specified by the model. Such manipulation helps the modeler to check the model against the user's requirements. The modeler can check whether the concepts from the problem domain are properly covered by the system, and whether the system provides all the information and functionality needed by

Part II: Overview of OOIS UML

the user. If it does not, the model can be refined and recompiled, while the development cycle from the model to the executable prototype is short. In other words, the generic GUI of the execution environment enables *rapid prototyping* of the system and a kind of *model debugging*.

Chapters 5 and 6 describe the details of the basic concepts and features of OOIS UML. Using a simple and illustrative running example of a small school information system, these chapters provide an informal and somewhat incomplete introduction to OOIS UML. They briefly present some core concepts used in the profile. Some of the concepts are inherited from the standard UML, but some of them are specific for the profile itself. The goal of these chapters is for you to quickly get acquainted with the formal and executable nature of the models developed using the profile, as well as with the main features the profile provides, without subjecting you to too much detail. In particular, Chapter 5 deals with the basic concepts of the OOIS UML language. Chapter 6 deals with the basic concepts and mechanisms of the GUI and its customization using the standard OOIS UML model library and run-time environment, as well as the querying feature.

The intention of this part of the book is not to describe and fully define the concepts in great detail, but to give an overview of some core parts of the profile. This is why the reader may have many open questions after reading the sections in Chapters 5 and 6. A “Frequently Asked Questions (FAQ)” subsection at the end of each section will list just a few. The FAQ subsection will give short answers to the questions and often refer the reader to the more in-depth explanations later in the book.

The manifestations of the OOIS UML concepts in the generic environment are also presented in the following chapters, in order to make the explanations of the concepts more intuitive. The manifestations that the concepts have in the GUI are called the *interactive manifestations* of the concepts. They define what can be seen and done within the system interactively, through the user interface, as a consequence of the specified model.

Interactive manifestations represent just one type of manifestation of modeling concepts at run-time, when the system is being executed. The other is the *programmatic manifestation*, which is the result of the execution of the behavioral part of the model (that is, the internally programmed logic of the system, not directly visible to the user). Both aspects comprise the *run-time* (or *dynamic*) *semantics* of the modeling concepts.

It must be emphasized that OOIS UML does not prescribe the concrete appearance of the GUI that has the defined interactive manifestations. Instead, it defines only *what* can be seen and done interactively at run-time, not precisely *how* it can be done in an ultimate system’s GUI. The descriptions given here present the GUI of the generic environment that can serve as an example of just one, among many possible appearances of the user interface that a modern OO information system may offer. The ultimate user interface of the system may be completely different. For example, it can be designed as a combination of form-oriented screens and those provided by the generic execution environment. But it must, however, ensure the defined run-time semantics of the modeling concepts.

The explanations use the Easylearn school information system as the running example. The motivation for the concepts introduced and explained in each section of the following chapters is presented alongside users’ requirements described at the beginning of the section.

Section Summary

- ❑ OOIS UML is a model-driven development method that consists of the following elements:
 - ❑ The language
 - ❑ The model library
 - ❑ The run-time environment
 - ❑ The design process
- ❑ The (major part of) models built in OOIS UML are formal and executable. When a model is developed, it can be executed within the run-time environment, after a possibly needed compilation, as with a standard computer program.

5

Basic Language Concepts

Using the simple running example of the Easylearn school system, this chapter introduces the basic concepts of OOIS UML language as an executable profile of UML. Each section begins with the requirements for the features that will be relevant to the material in that section. The section then presents the concepts, and, finally, the interactive manifestation of these concepts.

Classes and Attributes

This section examines classes and attributes.

Requirements

The Easylearn school offers different courses to its students, such as “Cooking for Dummies,” “Computers for Everybody,” and so on. For each course in the school, the information system should keep the following information:

- Name of the course (for example, “Cooking for Dummies,” “Computers for Everybody,” and so on)
- Duration of the course
- Price of the course
- Level of the course (which can be “basic,” “intermediate,” or “advanced”)
- An indicator that tells whether the course is offered on behalf of an associate school with which Easylearn has a cooperation contract

Part II: Overview of OOIS UML

Additionally, the school maintains information about all persons who either attend courses or work for the school (teaching or administration). For each person, the system should provide the following information:

- Name
- Photo
- Postal address
- Phone number
- E-mail address
- Credit card number

Concepts

The concepts of UML that are used to model the posed requirements are classes and attributes. This discussion describes the background of these concepts and their basic meanings.

Abstractions and Properties

The concepts of *course* and *person* are *key abstractions* in the problem domain of the Easylearn information system. As described in detail in Chapter 20, an abstraction is the result of discovering the essential characteristics of one entity or a group of entities that make them distinguishable from all other kinds of entities. Obviously, the concepts of *course* and *person* are significantly different from each other, as well as from other concepts and individual entities in the system, and are, therefore, abstractions. One form of abstraction is the tendency to neglect some differences of individual entities in the real world, in favor of generalizing their commonalities. For example, you do not need to distinguish between the concrete courses that will exist in the Easylearn system according to, for example, the average number of words that are pronounced by the teachers during the courses, or the number of books that are handed out to the students of the courses, simply because they are not relevant or needed by the users of the system. On the other hand, you emphasize the commonalities that *are* important for the users of the system, such as the fact that each course has its name or price. An abstraction is thus dependent on the perspective of the viewer and the context, and is not *a priori* defined.

A *key abstraction* is an abstraction that is important enough to be incorporated into the system's conceptual model, regardless of its concrete implementation. In other words, no matter what the implementation will look like, the system should enable the user to directly manipulate courses and persons, and to access their properties as listed. That is why *course* and *person* are key abstractions in this problem domain.

An abstraction represents a set of *instances*. Abstraction is actually a classification of individual instances according to their commonalities perceived by the viewer. For example, "Cooking for Dummies" or "Computers for Everybody" are instances (or elements) of the *course* abstraction. Similarly, John Smith, Arthur Barnstorm, and Vanessa Springfield are instances (or elements) of the *person* abstraction.

An abstraction is characterized with its *properties*. The properties for the *course* abstraction are name, duration, and so on, and for the *person* abstraction are name, photo, postal address, and so on. Each instance of an abstraction will have its own value of each property. For example, “Cooking for Dummies” will have the number 20 as the value of its duration property, and John Smith will have `john.smith@easylearn.com` as the value of its e-mail address property.

Section Summary

- ❑ An *abstraction* is the result of discovering the essential characteristics of one entity or a group of entities that make them distinguishable from all other kinds of entities.
- ❑ A *key abstraction* is an abstraction that is important enough to be incorporated into the system’s conceptual model, regardless of its concrete implementation.
- ❑ An abstraction represents a set of *instances*.
- ❑ An abstraction is characterized with its *properties*.

Classes and Objects

In OO software engineering, abstractions are modeled with *classes*. A class is a description of a set of objects that share the same structure, behavior, and semantics. An *object* is an instance of a class, an element of the set represented by the class. It is a concrete manifestation of an abstraction. It is an entity with a well-defined boundary and identity, which encapsulates state and behavior. Therefore, `Course` and `Person` will be classes in this model, and “Cooking for Dummies” and John Smith will be objects of these classes in the information system. Of course, the objects being discussed here are simply computer representations of physical things or of logical concepts from the real world and the particular problem domain. They are, of course, simplifications (often very drastic) of those things or concepts, as the result of their abstraction.

Therefore, a class is an element of the conceptual model of a domain. It resides in the model at design time (see Figure 5-1a). It describes the whole set of objects that share the same structure and behavior. A class actually describes that structure and behavior, and represents a template for creating objects.

Objects are instances of classes. They reside in the system’s object space and live at run-time (see Figure 5-1b). Each object has its state that is defined by the dynamic values of the structural features defined in its class. It also has the capability to demonstrate behavior defined by the behavioral features of its class, when this is requested from it. The set of all objects of one class existing in the system’s object space at some moment in run-time is called the *extent* of that class.

Figure 5-1a illustrates a simple yet valid UML *class diagram*, which renders (part of) the system’s conceptual model. Class diagrams in UML depict classes, their structural and behavioral features, their

Part II: Overview of OOIS UML

relationships, and possibly other elements of the model. In the simplest case, classes are rendered as rectangles with their names centered in the rectangles, as in Figure 5-1a.

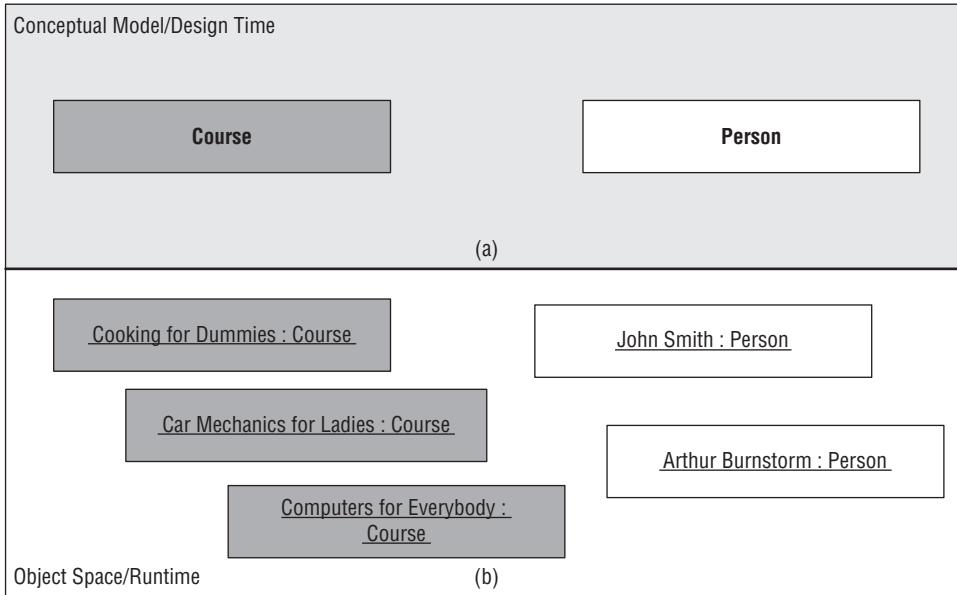


Figure 5-1: (a) Classes in the conceptual model. (b) Objects at run-time.

Figure 5-1b represents a UML *object diagram*, which may render a prototypical, sample piece of the object space. The objects in an object diagram may (but need not) represent some concrete objects — they may simply represent arbitrary, prototypical objects that may reside in the object space at some point in time. In other words, such a diagram shows what a piece of object structure may (but need not) look like. In that case, the object diagram is simply aimed at helping the reader to understand the conceptual model or some of its parts, as it is done here, and has no formal run-time semantics. In object diagrams, objects are rendered as rectangles, with the optional name of the object and its class separated by a colon, and underlined altogether. Underlining in UML diagrams generally indicates instantiation.

Consequently, a class and its objects are related by the type-instance dichotomy. Objects are instances of classes. Furthermore, a class resides in the conceptual model, at design time, and is an abstract, conceptual thing. Objects live in the application's object space, at run-time, and are concrete, physical things (in the context of the computer system). A class is only a description of a set of objects (that is, the structure and behavior that objects share). Classes live in the developers' and users' minds, or in the memory used by the modeling tool. Objects live in time and space. Classes reside on the design side, and objects on the execution side of the dichotomy.

When a class is defined in the conceptual model, it has the following semantics:

- Objects of that class can exist in the system's object space at run-time. These objects can be *created* and *destroyed*.

- ❑ These objects will demonstrate the same structural and behavioral features defined in the class.
- ❑ The objects will all have the same semantics. In particular, if a logical statement or constraint is formulated for the class, it must hold for all its objects.

An object has the following characteristics:

- ❑ An object is an instance of its class, and its class can be known at run-time. An object of a class can be distinguished from all other objects of the same or the other classes.
- ❑ An object has its lifetime, meaning that it lives since its *creation*, until its *destruction*.¹ An object can be accessed only during its lifetime. Additionally, an object takes some space in the system's object memory. Therefore, objects live in time and space.
- ❑ An object has its *identity*, meaning that it can be distinguished from all other objects of even the same class. The identity is *not* value-based, meaning that it is *not* based upon the value of any property or combination of properties of the object. Two distinct objects of the same class may have, in general, all their property values equal to each other, and they still represent two distinct entities. In addition, the identity of an object never changes regardless of the values of its properties. The identity of an object is implied by the existence of that object in space and time (that is, the object per se, and not by its internal state or behavior).

Section Summary

- ❑ *Class* is a description of a set of objects that share the same structure, behavior, and semantics.
- ❑ *Object* is an instance of a class, an element of the set represented by the class.

Attributes and Their Values

Properties of abstractions are modeled with *attributes*. An attribute is a named property of a class that describes a range of values that instances of the class can hold in *slots* that represent run-time manifestations of the property. An attribute is a structural feature of a class. Following are some elements of an attribute's definition:

- ❑ **Name** — A string of characters that uniquely identifies the attribute in the scope of its class. For example, `duration` and `outsourced` are names of the attributes of the class `Course`.

¹Note that this claim does not mention any dependence of the object's lifetime on the duration of the execution (such as of the computer program) that created it. Put simply, objects in OOS UML are implicitly persistent in the classical sense, but this is their inherent (rather than externally imposed) characteristic. Compare this to the semantics of classical OO programming languages where objects, because they are stored in the operating memory, always die with the cessation of the program that created them. This is, however, a limitation caused by the binding to the implementation (the concrete selected computer technology), completely inappropriate for the domain of information systems.

Part II: Overview of OOIS UML

- ❑ **Type** — A data type that defines the range of values that the slots of that attribute can hold, along with the operations applicable to these values. For example, the type of the attribute duration is Integer, and of outsourced is Boolean because it can have only the values true or false.
- ❑ **Default value** — The initial value that a slot of the attribute will hold when an object is created. For example, the attribute outsourced of the class Course can have the default value set to false, so that the user does not need to set its value each time a new course is created.

An attribute is a conceptual thing and lives on the design side of the type-instance dichotomy. It is a structural feature of a class. Objects, as instances of classes, which live at the execution side at run-time, have *slots* as manifestations of these attributes. Slots hold values as instances of the attributes' types. Therefore, slots of the attributes and the values they hold are actually instances of these attributes, although they do not have their own independent lifetimes. The lifetimes of attribute instances are tied to the lifetimes of their enclosing objects.² Each object of the class has its own value in the slot of an attribute, which is independent of other objects' attribute values.

The values of the attributes can be manipulated at run-time interactively (through the user interface) or programmatically (through the application's internally programmed behavior). It is said that objects change their states, which are defined by the values of their attributes, throughout their lifetimes. In other words, the attributes define a static structure of the objects of a class, which is shared by these objects, and the attribute values define the current states of the objects, which are proprietary to the objects and dynamically changed during run-time.

The values of the attributes can be accessed and modified according to the specification of their types. The types of the attributes are defined as abstract data types³ in the model and can be either built-in (that is, supported by the implementation) or user-defined (where the term *user* now denotes the modeler as the user of the modeling language). For example, built-in data types can be the following (the list is not complete):

- ❑ Boolean — Represents Boolean values, which can be either true or false.
- ❑ Integer — Represents integer numbers.
- ❑ Real — Represents floating-point numbers.
- ❑ Currency — Represents money amounts.
- ❑ Text — Represents strings of characters.
- ❑ Hyperlink — Represents strings of characters that are URLs.
- ❑ DateTime — Represents time stamps.

²This is an OOIS UML semantic specialization, not defined in standard UML. The detailed semantics will be described later.

³This is an OOIS UML semantic specialization, not defined in standard UML. The detailed semantics will be described later.

- ❑ **File** — Represents binary packages (“files”) whose contents are interpreted by an outer system (application), also known as Binary Large Objects (BLOBs) in database terminology.
- ❑ **Picture** — Represents binary files whose contents are digital pictures.
- ❑ **Sound** — Represents binary files whose contents are digitally recorded sounds.
- ❑ **Movie** — Represents binary files whose contents are digitally recorded movies (that is, video clips).
- ❑ **Email** — Represents a stored e-mail.

A concrete implementation of the profile can provide a library of built-in data types suitable for different business domains.

Section Summary

- ❑ An *attribute* is a named property of a class that describes a range of values that instances of the class can hold in *slots* that represent run-time manifestations of the property.
- ❑ Attributes define a static structure of the objects of a class (which is shared by these objects), and attribute values define the current states of the objects (which are proprietary to the objects and dynamically changed during run-time).

Notation

In UML diagrams, a class is rendered as a rectangle, optionally divided into several compartments. The only mandatory compartment is the top-most one, with the class name written in the center in bold. A separate compartment may optionally list the class attributes. For each attribute, its name, type, and default value are shown using the syntax *name : type = defaultValue*, as shown in Figure 5-2, which depicts the initial class model.

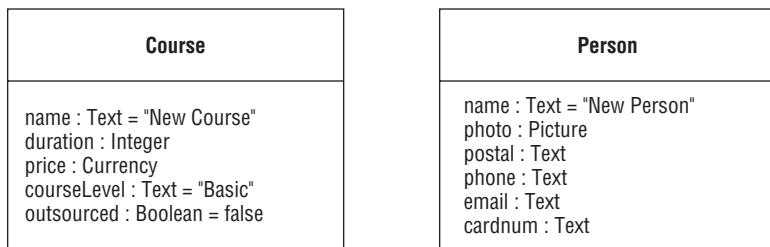


Figure 5-2: The initial Easylearn class model

When textual notation is used, slots of objects are usually referred to using the dot notation. For example, if `x` refers to an object of a class that has an attribute `m`, the slot corresponding to `x` is referred to as `x.m`.

Interactive Manifestations

The interactive manifestation of OOIS UML assumes that, regardless of the concrete appearance of the GUI, two things are important for the developers and the users:

- ❑ Developers should not make any additional or any significant effort to obtain the interactive manifestation of the developed conceptual model. In other words, the development framework should allow that the GUI could be obtained without any additional programming of the presented functionality, be it the front-end functionality (of the GUI itself) or the back-end functionality (for example, creation and destruction of objects, their persistence, and so on). This is a consequence of the fact that the described model, along with the precise semantics of OOIS UML, carries the information that is precise and sufficient to indicate the described interactive manifestation.
- ❑ The user should be able to access the information and issue the operations that imply from the semantics of OOIS UML and the conceptual model, regardless of what concrete approach the GUI uses, unless this is not desired in the target application.

The OOIS UML generic run-time environment provides different ways of presenting the object space immediately from the compiled model, without any additional manual coding or another kind of specific manual intervention, as shown in Figure 5-3. One possible approach is to render a tree whose nodes represent objects, while their sub-nodes represent the objects' slots holding attribute values. An example of a browser of the object space defined by the simple model developed so far is shown in Figure 5-3a. Note how objects of the classes Course and Person have their slots as specified in the model.

In such a GUI, the user can do several things with the system's object space:

- ❑ Create an object of a class by clicking the button for the desired class from the toolbar with the classes in the model (the part labeled "c" in Figure 5-3). The new object will appear in the "Created" object space browser (the part labeled "d" in Figure 5-3).
- ❑ Open a generic specification dialog for an object in the object space by double-clicking the object's icon. The generic specification dialog provides input-output controls corresponding to the object's attributes and their types. The part labeled "b" in Figure 5-3 shows the generic dialog opened for an object of the class Person, with, for example, a textbox for the attribute name or the picture for the attribute photo. The controls provide the necessary behavior for modifying the attributes. For example, the picture can be defined (uploaded to the system from a file in the file system), saved to another place (downloaded from the object space to the file system), viewed in a zoomed window, and so on.
- ❑ Modify attribute values of an object by double-clicking the corresponding icons for slots and modifying their values through the corresponding forms.
- ❑ Delete objects of classes by right-clicking an object in an object space browser and selecting Delete from the pop-up menu.

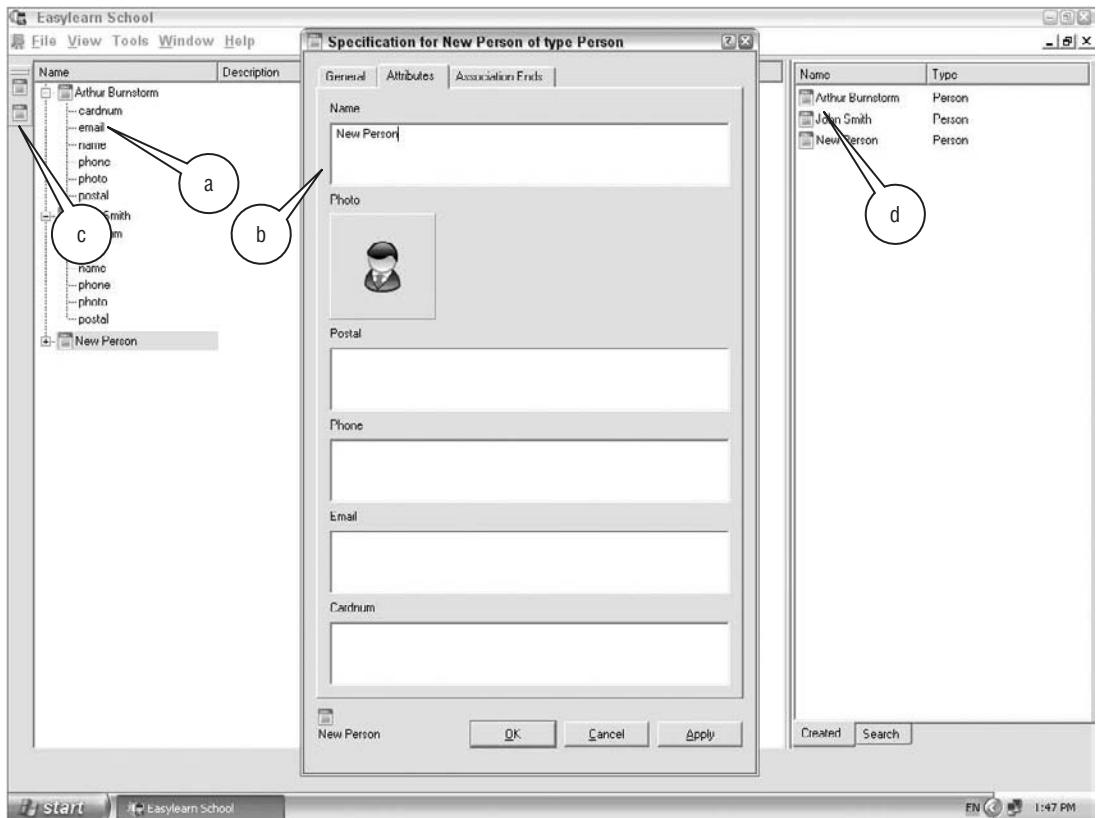


Figure 5-3: The generic run-time environment's GUI with (a) an object space browser, (b) a generic specification dialog opened for one object of the class Person, (c) the toolbar with a button for each class in the model for creating objects, and (d) the “Created” browser.

It is assumed that the system supports massive object spaces, meaning that the user can arbitrarily create many objects. Obviously, the space must be limited by the capacity of the physical storage of the computer system (that is, disks) or its address space, but it is assumed that the limits are high and are not imposed by the profile. Additionally, an object lives from its explicit creation until its explicit deletion, regardless of the activity of the execution that created it. For example, the user can create a couple of new objects, terminate the application, and then restart it later, but the objects will still be alive and accessible until they are explicitly destroyed. This feature is traditionally called *persistence* of objects, but this is an inherent property of objects in OOIS UML and is not externally imposed to the objects, nor does it have to be explicitly specified in the model or ensured by the developer.

Section Summary

- ❑ The interactive manifestations that are inferred from the semantics of OOIS UML include the following features:
 - ❑ Objects of classes can be created and deleted. An object lives in the object space during the entire interval between its explicit creation and deletion, regardless of the activity of the execution that created it.
 - ❑ Attribute values of objects can be retrieved and modified according to their types.

FAQ

Q: What is the difference, if any, between the notions of abstraction and class?

A: In a broader sense, abstraction is an ontological concept, a notion used for conceptualization regardless of the concrete language. This term refers to a mental process of simplifying, classifying, and formalizing things from the real world by revealing their commonalities and discarding irrelevant details until what remains is important or comprehensible, as well as the result of that process. Therefore, it is a bit of a vague concept without overly precise and formal semantics. Class, on the other hand, is a formal concept in OO modeling languages with precise semantics, although somewhat differently defined in various languages. In OOIS UML, class has specific and precise semantics as defined in this chapter and in the rest of this book.

Therefore, abstraction can be considered as an important aspect of the problem domain that must be modeled somehow in the software system. It is most often (but not necessarily) modeled with a class, and other mappings are possible, too. The concrete mapping of an abstraction (that is, a concept) from the problem domain to a language concept depends on the semantics of the language concept and its closeness to the requirements from the problem domain. The process of conceptualization is exactly the process of that mapping. Unfortunately, there is no formula for that process and its success, or else the process of software construction would be possible to automate and humans would not be needed. However, many useful hints exist. You will read more on this topic in Chapter 18.

Q: How can I identify classes from the problem domain?

A: As already explained, this is a matter of conceptualization — that is, of the mapping of abstractions from the problem domain (requirements) to the elements of the model. It is a mentally intensive and non-deterministic process, and there is no exact formula or prescription for it. However, many useful tips exist as described in a separate part of this book and in many other books. Essentially, because a class is a set of objects (which, in turn, represent some tangible or conceptual entities from the problem domain), revealing classes is a matter of classifying tangible or logical entities from the problem domain into sets according to their commonalities. You will read more on this topic in section “Identifying and Specifying Classes and Attributes” in Chapter 18.

Q: Given that a class is a set of objects that represent some entities from the problem domain or the real world, can an object be a direct instance of several classes at the same time because entities may be elements of several sets according to different classifications? For example, the a person may be a client of a bank and an employee of the same bank at the same time.

- A:** In standard UML, although at its higher compliance levels, an object can be a direct instance of several (not necessarily related) classes as in the given example.⁴ This feature is useful in the process of conceptual modeling in the early phases of requirements capturing. It has been introduced in the standard UML as a response to the needs as those stated in the question.

However, the feature creates many other semantic complications and undesirable side effects. Therefore, OOIS UML does not allow it, as most other OO languages with executable semantics do not support it either.

This feature can be used as an informal means for conceptual modeling in early phases of system analysis. When a formal and executable model is to be developed, such occurrences must be transformed into some other, more concrete modeling concepts available in OOIS UML. The modeler must demonstrate some creativity to find a proper transformation. As it can be concluded from the previous discussion (in this paragraph), OOIS UML is, to some extent, less expressive and less abstract than standard UML. This is a consequence of (i.e., a penalty caused by) its formal and executable nature.

- Q:** It is clear that objects of a class can be created and destroyed at run-time, but how is it actually specified in the application?

- A:** The run-time semantics of the concept of class suggest that objects of a class can be created and destroyed at run-time. There are two ways of doing that. The first is interactively, through the GUI, as described already. The second is programmatically, by the means of generic operations upon the structure (that is, the actions) specified within the behavioral part of the model. What is important are the semantics of actions that create and delete objects, which are defined in standard UML and more specifically in OOIS UML. What is less important, however, is how the actions are precisely written. This is a matter of the *surface language* used to specify details, and will be described later. The action “Create a new object of the class Person” can be written in many different ways, using either a textual or a visual notation, but its meaning remains the same. You will read more on this topic in section “Creation and Destruction of Instances” in Chapter 8.

- Q:** Can an object be reclassified to become an instance of another class during its lifetime? For example, how do you model the fact that a person changes status during his or her life and moves from child to adult, which are two distinct classes?

- A:** Standard UML at its higher compliance levels defines the action “Reclassify Object,” which does the described transformation. However, as with the feature of being a direct instance of several classes, reclassification causes some semantic complications and undesirable side effects. The problem would be simple if a class were only a simple set of objects and nothing more. But a class may specify structural and behavioral features possessed by its objects, and it is a question what happens with all of them when an object is reclassified. Moreover, the answer to this question is often problem-specific. This is why OOIS UML (as well as many other executable OO languages) does not support reclassification.

On the other hand, reclassification is very seldom, if ever, the only possible solution to a real-world requirement. There is almost always a simpler way to model such a requirement. In the most simple case, reclassification can be achieved by modifying values of attributes or changing states of a state machine. In the most complex case, reclassification can be simply equivalent to creating a new object of the target class, and copying (in a domain-specific way) its structural features from the original object, and deleting the original object afterward. This behavior can be easily implemented using other features of OOIS UML. Nevertheless, there are often simpler modeling solutions for the same requirement.

⁴This discussion is not talking about the concept of multiple inheritance that some readers might be familiar with, but about classifying an instance to several unrelated classes.

Part II: Overview of OOIS UML

- Q:** Does the separation of the two sides of the dichotomy (the modeling time and run-time) mean that the model cannot be modified while the system is running? For example, can classes and attributes be added or deleted during the exploitation of the system?
- A:** The described separation of modeling and execution is only conceptual — it does not mean that these two activities cannot happen at the same physical time. It is up to the implementation to allow, restrict, or totally prohibit the modifications of the model while the system is running. For example, an implementation can allow classes or attributes to be added or removed from the model while the system is running, or it can restrict these activities with some rules. Another implementation may, however, require that the system be shut down and the model be recompiled after modifications. What happens with the objects already existing in the object space when the model is changed is also a concern left to the implementation. Actually, the problem is very similar to the problem of modification and evolution of database schema in relational DBMSs.

However, one thing must be kept in mind. Whatever approach is taken by the implementation, standard UML and OOIS UML do *not* specify the actions of modifying the very model within the language. How the model is modified is a matter of the modeling tool and not of the language itself. In other words, there are no standard UML actions that work on the model itself. Consequently, the system (that is, “the program”) obtained from a UML model cannot modify itself during its execution. This is why modeling and execution are two distinct processes. On the other hand, OOIS UML provides a means to *read* the information about the model during execution, meaning that OOIS UML is *reflective*. This is the explanation of how the generic features described in this part are actually implemented. You will read more on this topic in section “Modeling and Execution” in Chapter 7.

- Q:** Classes seem to be somewhat similar to entity sets, whereas objects are similar to entities in ER modeling. What is the difference?
- A:** It is true that classes are similar to entity sets in ER modeling. Actually, the concept of class is a descendant of the concept of entity set. There is nothing strange about that: As I explained previously, software engineering does not experience revolutions that would discard all concepts from the past, but rather evolutions of existing successful concepts and paradigms. So far, there has been nothing particularly new that would differentiate class from entity in ER. However, there are many other features specific to the object paradigm and UML that will be presented in the rest of the book and that will differentiate these two concepts.
- Q:** Can classes and objects be implemented in a relational database?
- A:** Yes, they can. Because the concept of class is similar to the concept of entity set in ER modeling, the mapping can be the same, as long as the structural features of the class correspond to the features of an entity set. If classes have more complex features, the mapping may require more complex elements. It is worth noting once more that such a mapping is not important to the developer of an OOIS UML model, except in some very rare cases of having to cope with implementation-level and performance-tuning details. The entire executable system can be constructed using OOIS UML, and the mapping to the relational model, just as any transformation to an implementation form, can be performed by the framework (that is, model compiler and run-time environment) automatically.
- Q:** Because OOIS UML actually provides a means to implement persistent objects and supports the object paradigm, is it similar to other OO DBMSs, and what is the difference?
- A:** In essence, it is true that OOIS UML is similar to other OO DBMSs in its mission: to provide an OO modeling language that supports persistent object space, or object databases. However, there are some important differences. First, OOIS UML is based on standard UML and, thus, exploits some of the advantages of standards. It seems that the most important problem with commercial OO DBMSs is not in their quality, but in the lack of standardization. Most of them were developed long before the emergence of UML and, thus, weakly support UML (or don’t support it at all).

Additionally, OOIS UML brings a lot of other interesting concepts not available in most OO DBMSs. Finally, OOIS UML can be implemented using a commercial, off-the-shelf relational DBMS, which may be particularly important when a legacy system must be extended by an OO extension, or when several applications developed using different approaches (for example, OOIS UML and a traditional one) must co-exist working on the same relational database.

- Q:** If the identity of an object is based on its existence, and not on the value of any of its attributes, how can two objects of the same class, having all their attribute values equal, be discriminated at all? And why is this so?
- A:** The identity of an object is based on its existence in time and space, which means that it *can* be distinguished from any other object, simply because it *is* a different object. The question of *how* it can be distinguished is another issue, and can be split into two questions.

The first is how the system (that is, the execution engine) internally differentiates between the objects. This is a matter of implementation, and is not relevant to the developer, as opposed to the case when the relational paradigm is used. For example, if a relational database is used for implementing the object space, the system may internally generate a unique technical ID that serves as the primary key for the records that implement objects.

The second question is how the user of the system can distinguish the objects. This is a matter of perception, and is mostly handled by the GUI. Many approaches are possible, and examples will be given throughout this book.

Another issue is whether two different objects representing two different entities from the real world should or should not be allowed to have all their attribute values equal. This is an ontological question and is discussed in detail later in this book. You will read more on this topic in section “Identity” in Chapter 8.

Finally, why is an object’s identity defined in this way and not simply, as in the relational paradigm, based on the object’s attribute values? Of course, it could have been defined otherwise — the definition of the semantics are simply human decisions, which are made according to estimations of what is more or less useful in practice. And, this is the very answer to the question: current understanding of the OO concepts is that objects should have existence-based identity because this is more suitable in practice. Namely, objects most often represent tangible things from the real world, which really have an identity.

- Q:** Is there any difference between data type and class?
- A:** In standard UML, data types and classes have many things in common, but also have several differences. OOIS UML further specifies their differences. In brief, they are similar because they both represent classifiers (sets) of instances. One of their differences is the identity of their instances. Instances of classes have identity and instances of data types do not — they are simple values. The similarities and differences are discussed in Chapter 8.
- Q:** What types of attributes are available for modeling? Are they standard?
- A:** Standard UML does not constrain the type of an attribute — it can be any class or data type. OOIS UML constrains the type of an attribute to be a data type, for the reasons explained later. In OOIS UML, the type of an attribute can be any data type, either built-in (that is, part of the library provided with the implementation of the profile) or user-defined. However, a concrete implementation can reduce the set of allowable types even further. The set of built-in data types is a matter of a library of data types available in the implementation. In its current state of development, OOIS UML just recommends some data types to be available in the library, and does not standardize them. You will read more on this topic in section “Data Types” in Chapter 8.
- Q:** In the generic GUI, how do you reach a specific object that exists in the object space? And why are newly created objects collected in a separate browser?

Part II: Overview of OOIS UML

A: The object space of an information system may often be huge, so it is not convenient and even possible to have all objects displayed in a browser. The users usually work with small sets of objects for some time, until they replace the working sets. This is why a browser displays only a small part of the object space, the one that has been reached by the user at some time. The rest of the objects can be reached in several other ways (for example, by searching and querying the object space). You will read more on this topic in section “Querying” in Chapter 6.

For similar reasons, the newly created objects are temporarily displayed in a separate browser so that the user can manipulate them immediately after their creation. When the application is exited and restarted, they are treated just as all other objects — available in the browser if included in the working set, or reachable from the object space by other means.

Associations

This section introduces a basic relationship between classes for modeling structure — association relationship.

Requirements

In the Easylearn school information system, objects of the classes Course and Person do not exist unrelated. It is obvious that these objects must be related in a way that should conceptualize the fact that a person can *attend* some courses, and a course can be attended by some persons. The system should support this relationship. The system should allow the user to assign persons to courses, as well as to release those assignments when necessary. Of course, the user should be also able to navigate from a person to the courses the person attends, and vice versa.

Concepts

In UML, classes can be related by several kinds of relationships. One of the most important relationships is the *association* relationship.

Association is a structural relationship among classes that describes a set of *links* that may connect their objects. Association is a semantic relationship between classes that implies connectivity of their objects.

In UML, an association is rendered as a line connecting classes, as shown in Figure 5-4. Each association can have a name that describes its meaning. The name is written on the line, somewhere near its middle. Each end of an association can have a name that describes the *role* of that side in the association.

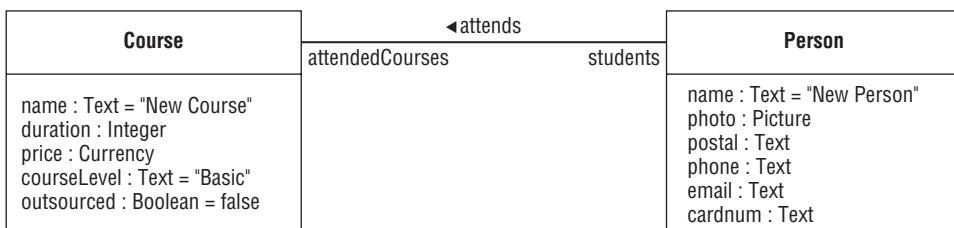


Figure 5-4: The Easylearn class model with two classes (Course and Person) and one association (attends).

Obviously, the classes `Course` and `Person` should be related with an association, as shown in Figure 5-4. The association is named `attends`, and means that an object of the class `Person` can be linked with a set of objects of the class `Course`, with the meaning that the person *attends* the courses (the small filled triangle near the name points to the direction of reading and interpreting the association name). The roles are named accordingly. The association end at the class `Course` is named `attendedCourses` because the courses linked to a person play the role of the person's attended courses, and the opposite end is named `students` because the persons linked to a course play the role of the course's students.

If an association between the classes *A* and *B* exists in the model, as illustrated in Figure 5-5a, a link of that association can be created between an object *a* of *A* and an object *b* of *B* at run-time, as illustrated in Figure 5-5b. An existing link can be deleted at run-time, too.

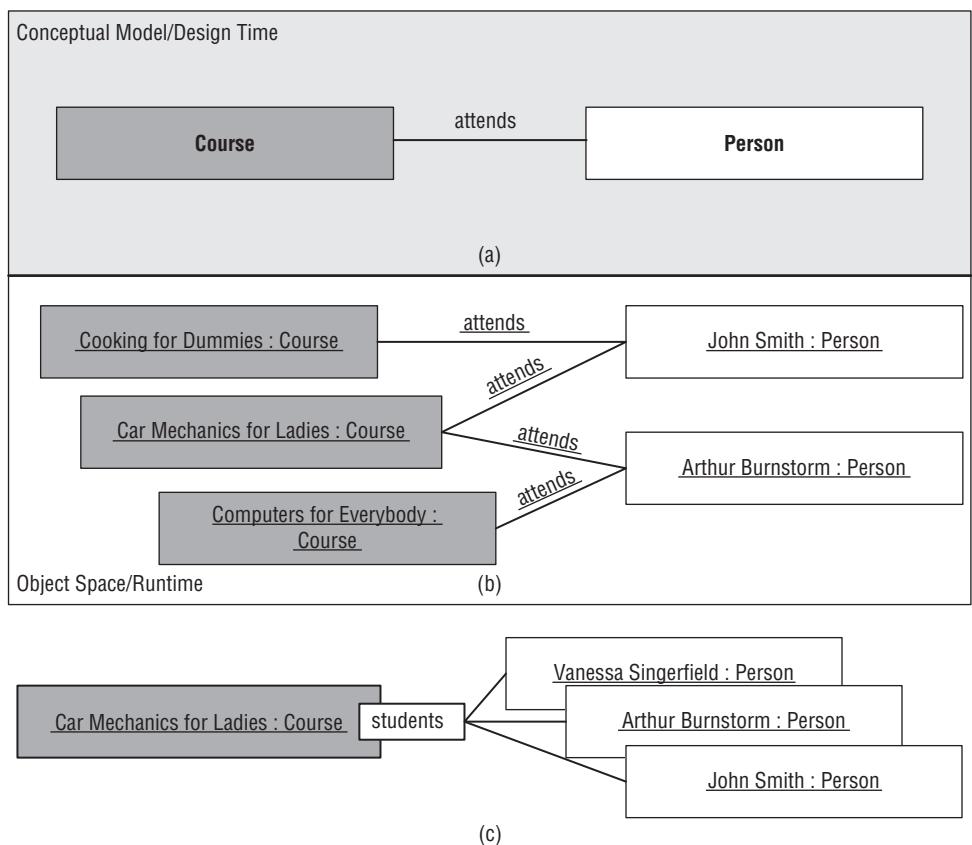


Figure 5-5: (a) Classes and association in the conceptual model. (b) Objects and links at run-time. (c) The collection of linked objects can be accessed through a slot in an object, which is a manifestation of the association end owned by the class at the opposite end of the association.

The association end at the class *B* can be specified in the model as a *property owned* by the class *A* at the opposite end of the association. Each object *a* of the class *A* will have a *slot* of this property that results in the collection of objects of *B* linked to the object *a* owning the slot, as shown in Figure 5-5c. For example, the association end `students` is treated as a property owned by the class `Course`, and if *c* is an object of

Part II: Overview of OOIS UML

Course, its slot `students` will result in a collection of linked objects of `Person` (see Figures 5-4 and 5-5c). The slot is textually referred to as `c.students`. To be a property owned by a class, an association end must be specified so in the model; if it is not (and it does not need to be), the objects will not have such a slot. In OOIS UML, an association end is owned by the opposite class by default. The alternative will be described later, and is not significant at this moment.

It should be noted that, as opposed to unidirectional references in classical OO programming languages, links are inherently bidirectional connections between objects. Namely, if a link between a course `c` and a person `p` is created, then `c` will have `p` in its collection `c.students`, and `p` will also have `c` in its collection `p.attendedCourses`. There is no need for any additional maintenance than a simple creation of a link between `c` and `p`. In other words, the semantics of associations inherently ensures that if `p` exists in `c.students`, then `c` also exists in `p.attendedCourses`, or vice versa. This can be compared to the more elementary semantics of unidirectional references in classical OO programming languages where separate actions are needed to set references in objects on both sides and keep them in sync. Instead of having a simple meaning that an object “refers to” another object, which is rather low-level and implementation-oriented (as is the case in classical OO programming languages), the concept of associations and links is more abstract and closer to the real-world interpretation that two objects are in a conceptual relationship.

These are the most important run-time manifestations of the concept of association.

Because attributes of a class are manifested with slots in the class’s objects resulting in attribute values much as owned association ends are manifested with slots that result in collections of linked objects, and denoted textually in the same way by using the dot notation, the term “properties of a class” will uniquely denote attributes and owned association ends, and the term “slots of an object” will uniquely denote their manifestations that result in attribute values and linked objects respectively, if something holds for both.⁵

Links exist at run-time, in the system’s object space (see Figure 5-5b). They are structural connections between objects. A link does not have its independent life — a link automatically dies when an object on any of its ends is destroyed. In other words, a link cannot exist without the objects on all its ends. By default, there can be no more than one link of the same association linking the same objects. (In a more general case that will be described later, there can be several links of the same association between the same objects.) Links are structural connections between objects, meaning that different parts of the running system (such as behaviors of objects, queries, or the user through the GUI) can traverse the object structure, by accessing the objects linked to one object over the links of a certain association.

As a general conclusion, links are instances of associations. An association is a relationship between classes, while links are connections between instances of classes. An association is a description of a set

⁵This is a discrepancy from standard UML terminology, but not in the semantics. Namely, in standard UML, *property* is a model element that can (but need not) be also an end of an association. Additionally, a property that is an association end may be owned by the class or by the association. In case the class owns the property, it is called an *attribute*, regardless whether it is or it is not an association end. Therefore, in standard UML terminology, attributes denote properties owned by a class and may be also association ends. However, this terminology seems to be somewhat confusing for beginners (and even for experienced users of earlier versions of UML), difficult to explain and understand. It differs from the usual and commonly accepted understanding of the terms “attribute” and “association end.” On the other hand, OOIS UML introduces some other semantic specializations for the properties that are and that are not association ends, so it needs different and clear terms for both of them. Note that, in standard UML 2.x (as opposed to earlier versions of UML), there are no separate terms for attributes (properties) that are and that are not association ends. In OOIS UML, they are commonly referred to as *properties*, and when discriminated, they are referred to as *attributes* and *association ends*.

of links. Therefore, an association is a conceptual thing that exists in the conceptual model, at design time. Links are physical things that exist in the object space, at run-time. In other words, associations exist at the design or type side, whereas links exist at the execution or instance side of the dichotomy (see Figure 5-5a, b).

According to these definitions, the object structure of the system can be regarded as a typed graph (see Figure 5-5b), whereby objects are nodes, and links are edges of the graph. The graph is typed because each node (object) has its type (that is the class of which the object is an instance), and each edge (link) has its type (that is the association of which the link is an instance). It is as if the graph is “colored”—each node (object) has its color (representing its class), and each edge (link) has its color (representing its association).

For example, imagine that the nodes representing objects of `Course` in Figure 5-5b are red and the nodes representing objects of `Person` are green, whereas the edges representing links of `attends` are blue, as opposed to links of some other associations that may exist in an enhanced model. In other words, the elements of the graph (that is, nodes and edges) are classified into different sets—these are classes and associations. This is why both classes and associations are referred to as *classifiers* in UML. Note that the links in UML object diagrams such as the one in Figure 5-5b are rendered as lines, with an optional name of the association written near the middle of the line and underlined (underlining again indicates instantiation).

At run-time, the behavioral part of the system manipulates the graph by creating and destroying objects, reading and modifying their attribute values, creating and destroying links, and accessing the objects over the links. In effect, the entire system’s behavior is almost all about manipulating the underlying graph-like structure as described. This is the paradigm used in OO information systems modeled in UML, and the image that the developers should have in mind when making the design of the system. Compare this with the image of tables, fields, and records that the developers have in mind when using the relational modeling paradigm. These two images should not be mixed up, and the developers should not think about the mapping of the graph-like structure into the relational model, simply because it is not needed—the UML semantics provide enough concepts to build the entire system based on the described structure.

Section Summary

- ❑ An *association* is a structural relationship between classes that describes a set of links among their objects.
- ❑ Each association can have a name that describes the meaning of the association. Each end of an association can have a name that describes the *role* of that side in the association.
- ❑ An association end can be a property of the class at the opposite end of the association. Each object of the class will have a *slot* of this property that results in the collection of objects of the class at that end linked to the object owning the slot.
- ❑ At run-time, the system manipulates with the object graph-like structure by creating and destroying objects, reading and modifying their attribute values, creating and destroying links, and accessing the objects over the links.

Interactive Manifestations

The introduction of an association into the conceptual model of the system has the following interactive manifestations in the generic GUI of the execution environment:

- New sub-nodes appear under the nodes for objects in the object space browser, as shown in Figure 5-6. These sub-nodes represent the slots that are manifestations of association ends, whose sub-nodes are objects linked to the slot. The user can navigate through the object space by hopping from one object over its slots to the linked objects, and so on, indefinitely. In other words, the user may circle over the same objects when further expanding the nodes of linked objects. This is because the browser actually shows a tree spanned over the objects and links in the object space graph structure. The tree can be expanded endlessly because it is spanned over the cycles in the graph.

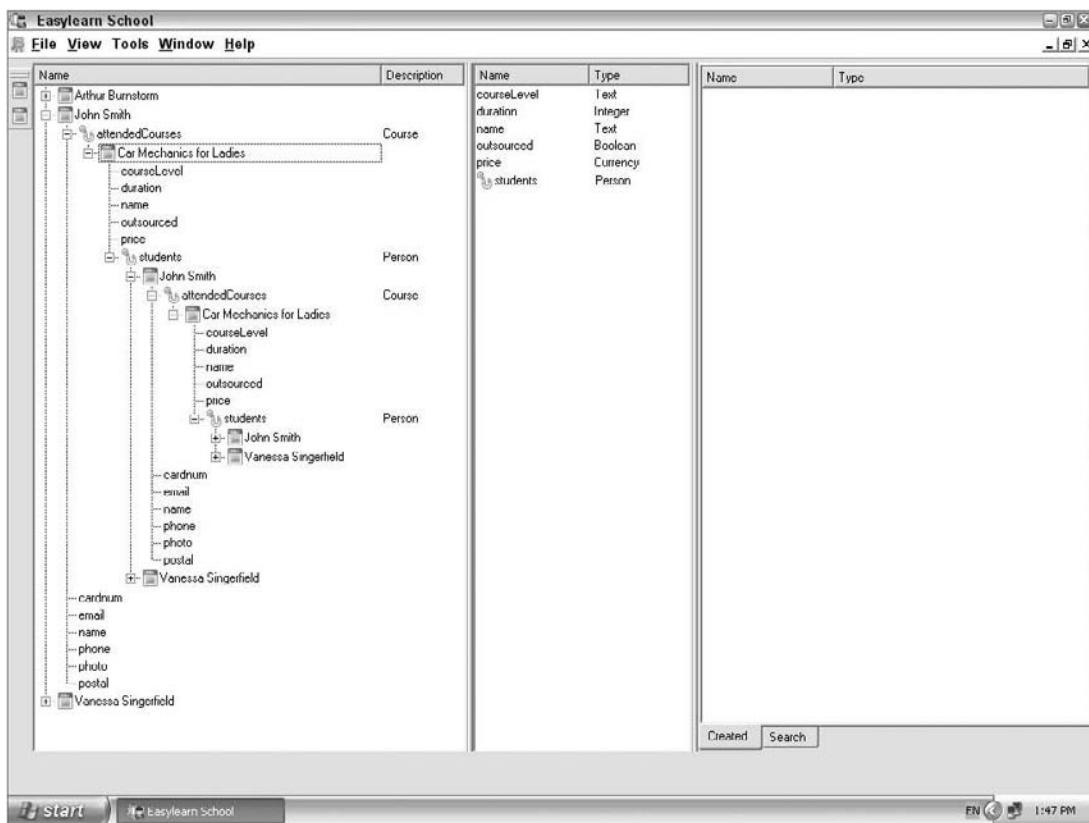


Figure 5-6: An example of an object space browser, where objects have sub-nodes for the linked objects.

- The user can create a link of the association attends by simply dragging an object of the class Person and dropping it onto the slot students of an object of the class Course. Of course, the dragged object will appear under the students slot of the target course, and the course will also appear under the attendedCourses slot of the person. The same effect can be achieved if

an object of the class Course is dragged and dropped onto the slot attendedCourses of an object of the class Person.

- ❑ The collection of linked objects can be seen in many other ways. For example, the generic specification dialog for a slot that is a manifestation of an association end lists the collection of all objects linked to that slot, as shown in Figure 5-7.

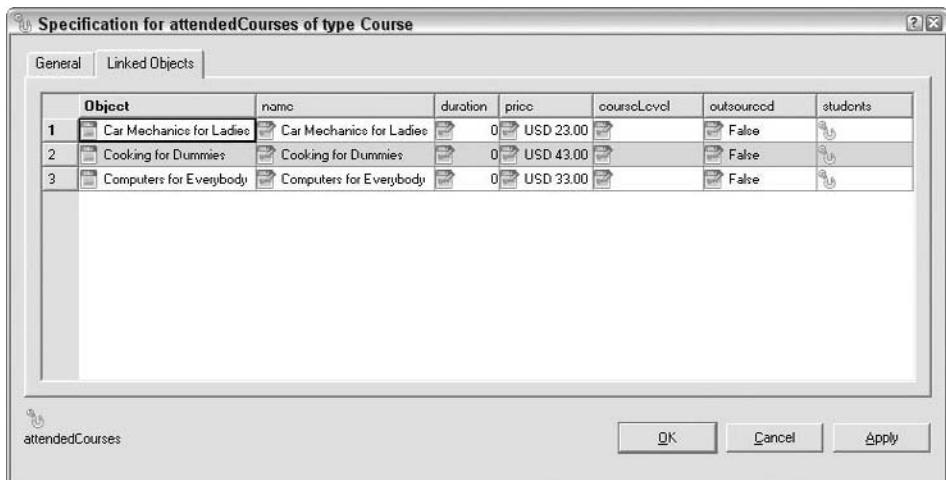


Figure 5-7: An example of a specification dialog for a slot listing all linked objects

- ❑ To delete a link, the user can select an object in the collection and then right-click the pop-up context menu command Unlink Objects. Note that only the link will be deleted, and not the linked object.
- ❑ The user can create many arbitrary links between objects, but not more than one link of the same association between the same two objects by default. As objects, the links also survive the termination of the application, until they are explicitly deleted, or until an object on any end is deleted.

Section Summary

- ❑ An association from the conceptual model has the following interactive manifestations in the running application:
 - ❑ The user can traverse the object structure by hopping from one object, via its slot that is manifestation of an association end, to the linked objects.
 - ❑ Links can be created and deleted by appropriate GUI commands. For example, a drag-and-drop of an object onto a slot creates a link.

FAQ

- Q:** Can an association relate more than two classes, or is it just a binary relationship?
- A:** In UML, an association can arbitrarily relate many classes (that is, it can be an N-ary relationship). In general, a link as an instance of an association that relates N classes is a tuple of N object identities, whereby each of these objects is an instance of the corresponding class. However, the association ends of an association with more than two ends never belong to a class, so the objects of the associated classes never have slots as manifestations of the association ends. Association ends can be owned by classes only in binary associations. You will read more on this topic in section “N-ary Associations” in Chapter 10.
- Q:** In the given sample model, nothing is said about how many persons can attend a course, and vice versa. What if I want to restrict the number of courses attended by a person?
- A:** This is a matter of cardinality of the collections of objects linked to an object by links of the given association. This cardinality is arbitrary, by default, at both association ends, and can be constrained by the means of *multiplicity* described later. You will read more on this topic in section “Consistency Rules” in this chapter.
- Q:** What about the time component of objects being related to each other? Does a link carry that component? For example, what if I want to capture the fact that a particular person used to attend a course in the past within a specified period, and that the same person now also attends the same course, but there was a pause between these two periods.
- A:** Basically, a link is a binary piece of information — it either exists or does not exist between the given objects, and, thus, does not carry any time stamp. In other words, a link indicates that the objects are related at that moment, and nothing more. For the given example, a simple association between the classes Course and Person thus cannot provide the information about what used to be in the past. If this is needed, however, as in the example given in the question, an *association class* can be used instead of a simple association. An association class is both a class and an association, and its instances are links and objects at the same time, so they can have attribute values. The attributes of the association class in this case can carry the information about the start and end of the period of course attendance. You will read more on this topic in section “Association Classes” in Chapter 10.
- Q:** The concept of association appears to be similar to the concept of relationship set in ER modeling. What are the differences, if any?
- A:** Just as class is a descendant of entity set, association is a direct descendant of relationship set in ER modeling. However, associations in UML have many other features that did not exist in ER modeling. You will read more on this topic in section “Special Characteristics of Association Ends” in Chapter 10.
- Q:** What about the implementation of associations and links in a relational database?
- A:** Because association is basically close to the concept of relationship set in ER modeling, it can be mapped to a relational database in the same way.
- Q:** Both an attribute of type X and an association end connected to the class X have the same manifestation in objects of the considered class that owns the attribute or association end: a slot that refers to a (collection of) instance(s) of type X. What is the difference, if any, between the two? How can I deduce which concept to use in the model — an attribute of type X or an association with X?
- A:** These two really have the same manifestation in the objects. Standard UML defines even more features applicable to attributes (that are not association ends) that make them semantically equivalent to association ends, such as multi-valued attributes that result in collections of instances.

The notation also supports this equivalence: an attribute of type X can be rendered in the notation shown for attributes, as well as an association toward X, or even both at the same time. However, standard UML does not provide more significant difference in the run-time semantics between the two concepts. The lack of significant semantic difference may raise questions like this one during conceptual modeling.

In order to avoid such ambiguities in conceptual modeling, OOIS UML clearly differentiates the run-time semantics of the two concepts. The differences are explained later in this book. The differences then clearly indicate which concept should be used to meet a requirement — the one that better fits the requirement.

Generalization/Specialization Relationships

This section introduces generalization/specialization relationships.

Requirements

In the Easylearn school information system, there is another key abstraction of *teachers*. Teachers are those persons who *teach* courses. Moreover, being persons, teachers also have their names, photos, e-mail addresses, and all other properties of persons, and they can be students of other courses in the school. However, teachers are a special kind of persons for which the system should also record their Web site addresses (so that the students can be informed about their teachers), titles, and reception hours (so that the students can contact them personally).

Concepts

In UML, another important kind of relationship between classes is *generalization/specialization* relationship.

In the conceptual model of the Easylearn information system, the new class Teacher will be a *specialization* of the class Person (see Figure 5-8). Or, taken from the opposite direction, the class Person will be a *generalization* of the class Teacher. This relationship will conceptualize the fact that a teacher is *a kind of* a person. As the simplest implication of this relationship, objects of Teacher will have all properties of the class Person, although they are not explicitly re-specified in the class Teacher. Obviously, this reduces redundancies in the model and simplifies modeling, but this is yet the most elementary effect of this relationship.

Generalization/specialization is a relationship between two classes, which implies that the objects of the specializing class are substitutable for objects of the generalizing class. The specializing class is also called the *subclass*, the *derived class*, the *child*, or the *descendant*. The generalizing class is also called the *superclass*, the *base class*, the *parent*, or the *ancestor*. This relationship is sometimes also called *inheritance* because it is related to the notion of inheritance in the object paradigm, but this relationship has a broader meaning in UML and, thus, will not be referred to as such here.

Generalization/specialization relationship has two significant semantic manifestations:

- **Inheritance** — The subclass inherits all structural and behavioral features, relationships, and semantics from the superclass. This means, for example, that objects of the subclass have all slots that exist in objects of the superclass (note that transitivity is assumed if the base class is a specialization of another class, and so on), without explicit specification. Similarly, the subclass

Part II: Overview of OOIS UML

inherits all relationships, meaning, for example, that objects of the subclass can be linked by the links of the associations connected to the superclass. Inheriting the semantics means that an object of the subclass is conceptually a kind of an object of the superclass. In other words, any statement that holds for objects of the superclass also implicitly holds for objects of the subclass. Consequently, everything that can be done with objects of the superclass can also be done with objects of the subclass.

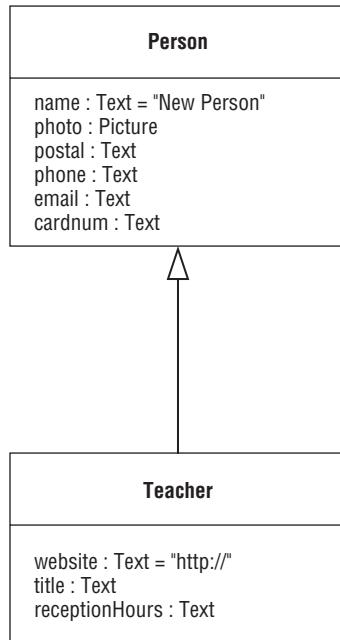


Figure 5-8:
Generalization/specialization
relationship — the class Person
is a generalization of the class
Teacher, or, taken from the
opposite direction, the class
Teacher is a specialization of the
class Person.

- **Substitution⁶** — Whenever and wherever an object of the superclass is expected, an object of the subclass can occur. This is a consequence of the fact that an object of the subclass is a kind of object of the superclass, and that everything that can be done with objects of the superclass, can also be done with objects of the subclass.

Essentially, as already described, a class specifies a set of objects in the information system. The statement that an object is an instance of a class is equivalent to the statement that the object is an element of the set represented by the class. If a derived class *D* (for example, *Teacher*) is a specialization of a base class *B* (for example, *Person*), this means that the set of objects *D* is actually a *subset* of the set *B* (the

⁶This property is often called the Liskov Substitutability Principle, after Barbara Liskov, who first formulated it.

set of teachers is a subset of the set of persons). In other words, specialization relationship specifies subsetting — D is included in B . Generalization relationship (actually, the same relationship observed in the opposite direction) specifies supersetting — B includes D . Therefore, when an object is an instance of D , it is also implicitly an instance of B because all elements of D are also elements of B . Additionally, everything that holds for the elements of B must also hold for the elements of D because they are also implicitly elements of B . For example, every teacher is implicitly also a person, and because of that, all that is said to hold for persons in general, must also hold for teachers in particular.

The inheritance manifestation and the substitution rule are, therefore, implications of the described interpretation of generalization/specialization relationship in the set calculus. The inheritance manifestation means that every instance of B has a certain set of features. Because instances of D are implicitly also instances of B , they also have the same features. Similarly, the substitution rule is implied from the fact that if, at some place, an instance of B is expected, instances of D may also appear because they are also instances of B .

Consequently, the phrase “a direct instance of a class” denotes an instance (element) of that class (set) that is not an instance (element) of any other specializing class (subset). On the other hand, the phrase “an instance of a class” denotes all direct instances (elements) of that class (set) along with all (direct or indirect) instances of its specializing classes (subsets), transitively. In the object paradigm generally, and in UML especially, the substitution rule is thus the fundamental rule that is always assumed when a statement is given for instances of a class — that statement must hold for all direct or indirect instances of that class.

Actually, it is a general tendency that one who deals with an instance of a class actually does not assume that the instance is really a direct instance of that class and nothing more special than that. If a software system is soundly based on that principle, it will gain more flexibility, simply because a specializing class can be added at any time later, without affecting the clients that deal with any instance of its generalization. For example, adding the class `Teacher` has not affected the class `Course` whose objects can still expect from persons all they were able to expect in the past (for example, accept them as students), although new, special kinds of persons (such as teachers) appear at some places.

As you can see, also, if an object is an instance of a certain class D , it is said that it is of type D , and also of type B , because generalization/specialization relationship includes subtyping semantics.

A class can be *abstract*, meaning that it cannot have direct instances at run-time. Such a class is aimed as a generalization of other, concrete classes that can have direct instances. It serves to make the clients of the class hierarchy more independent because they rely only on generalization, without being aware of specializations. For example, if you specialize the class of persons into male and female subclasses, the class of persons will be abstract because there will be no direct instances of that class (that is, there will be no objects that are persons and neither male nor female). The existence of the class of persons is probably important for the observed system because many of its features do not depend on specializations, and are thus easier to implement uniquely simply by relying on generalization.

In summary, generalization/specialization is a conceptual relationship that exists in the conceptual model, at design time. It has the described manifestations in the object space, at run-time, but does not have an explicit “instance” counterpart.

In UML notation, a generalization/specialization relationship is rendered as a line with a hollow triangle as an arrowhead, drawn between the symbols representing the related classes. The arrowhead points to the generalizing class (that is, the superclass), as shown in Figure 5-8. The graph of classes with generalization/specialization relationships in the model is, therefore, a directed, acyclic graph because circular

Part II: Overview of OOIS UML

specializations are not allowed (a class cannot be a direct or indirect specialization of itself). If a class is abstract, its name is written in italics.

Consequently, there is a new class **Teacher** in the conceptual model of the problem domain, as shown in Figure 5-9a. However, this class is not independent, but it specializes the class **Person**. This is because a teacher *is a kind of* a person, meaning that a teacher inherits all attributes and relationships from a person, and everything that is applicable to a person is also applicable to a teacher.

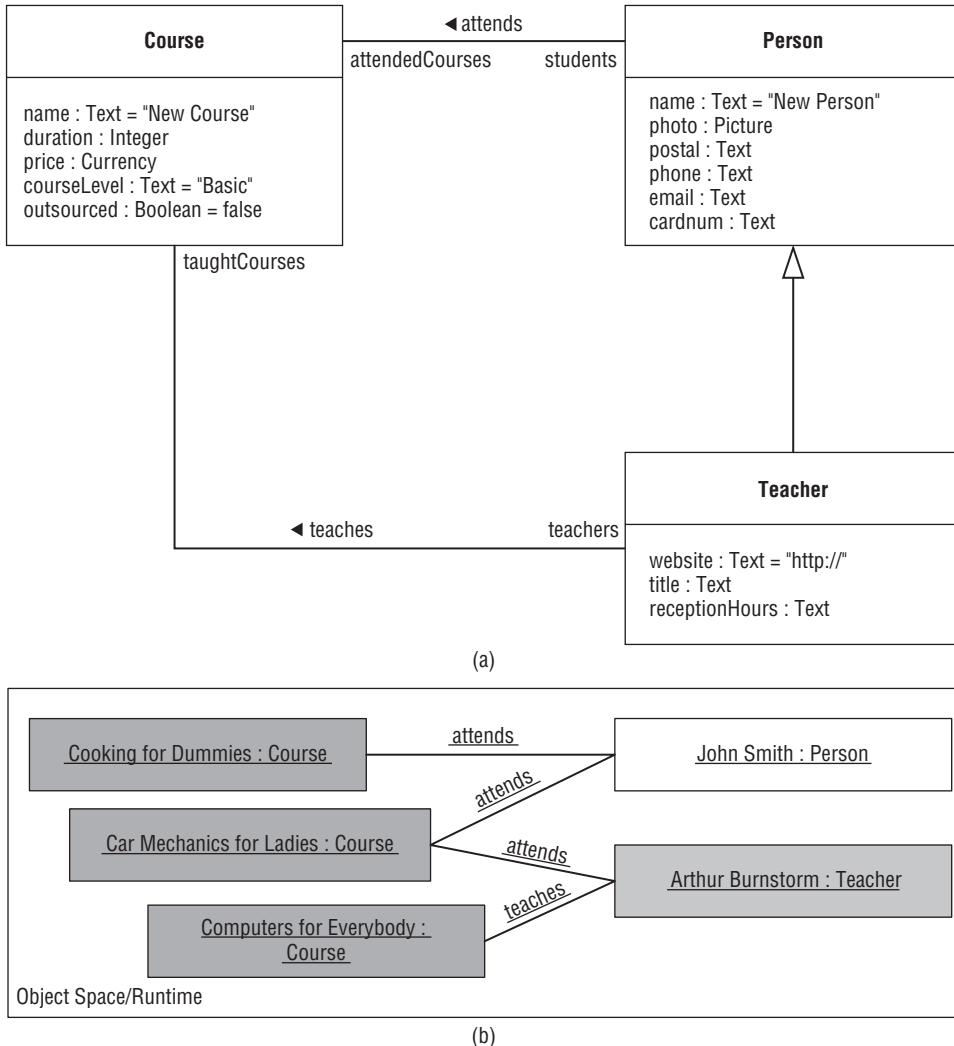


Figure 5-9: (a) The Easylearn conceptual model with three classes (Course, Person, and Teacher), two associations (attends and teaches), and one generalization/specialization relationship (Teacher is specializing Person). (b) An example of the object structure at run-time.

First, Teacher inherits the attributes from Person, so a teacher also has a name, postal address, phone, e-mail, photo, and card number. Teacher also inherits the participation in the association attends, meaning that a teacher can attend courses like any other person. In other words, an object of the class Teacher

can also be linked with objects of the class `Course` by links that are instances of the association `attends` (see Figure 5-9b).

In general, whenever and wherever a person is expected, a teacher can occur, too. All this implies from generalization/specialization relationship between these two classes and is directly manifested at run-time without any other explicit specification.

As you can see, a course can have both teachers and ‘ordinary’ (non-teaching) persons as its students. Although they are direct instances of different classes, the course looks on them as persons only, and does not see any difference between them. In other words, the course abstracts the differences between these specific kinds of students and generalizes all students. Consequently, a course can treat its students just as persons, and not as anything more specific.

For example, a course cannot rely on the fact that any of its students has a Web site address. This is the power of abstraction and generalization, whereby a client (the course in this case) abstracts the differences between the suppliers (the students in this case) and relies on their common interface (person in this case), allowing that the suppliers may be also some specific things, without jeopardizing the relationship between the client and the supplier.

This is the key prerequisite for reliable and flexible software, because interconnections between abstractions are weak and controllable. For example, a new specializing class can be added to the model, significantly enriching the application’s conceptual model and functionality, but without modifying the existing part. In fact, note that the new class `Teacher` has been added to enhance the system’s features, without a necessity to modify the existing part (the classes `Course` and `Person`, and the association `attends`).

However, teachers are not ordinary persons, but *special* ones, meaning that they also have something special that other persons do not have. First, they have their Web site addresses, titles, and office hours, modeled with the attributes of the class `Teacher`. Additionally, teachers can *teach* courses. This is modeled with the association `teaches`, as shown in Figure 5-9a. Thus, objects of the class `Teacher` can be linked with objects of the class `Course` by links that are instances of two different associations. The first kind of links has a meaning that a teacher, like any person, can *attend* courses, and the second kind has a meaning that a teacher, unlike other persons, can *teach* courses. Consequently, a teacher *is* a kind of a person, but a person *is not* a kind of a teacher.

Section Summary

- ❑ *Generalization/specialization* is a relationship between two classes, which implies that objects of the specializing class are substitutable for objects of the generalizing class.
- ❑ A generalization/specialization relationship has two significant semantic manifestations:
 - ❑ **Inheritance** — The specializing class inherits all structural and behavioral features, relationships, and semantics from the generalizing class.
 - ❑ **Substitution** — Whenever and wherever an object of the generalizing class is expected, an object of the specializing class can occur.

Interactive Manifestations

Generalization/specialization relationship has precise semantics so that the following interactive manifestations can be obtained from the conceptual model without any additional specification or programming effort:

- ❑ Objects of the specializing class will have slots for the properties inherited from the generalizing class, along with slots of the properties explicitly defined in the specializing class. For example, in an object space browser, objects of the class Teacher will have slots for all attributes from the class Person, as shown in Figure 5-10.
- ❑ The same holds for slots for the association ends inherited from the generalizing class, along with the association ends owned by the specializing class. For example, in an object space browser, objects of the class Teacher will have slots for the association ends attendedCourses, as well as taughtCourses, as shown in Figure 5-10.

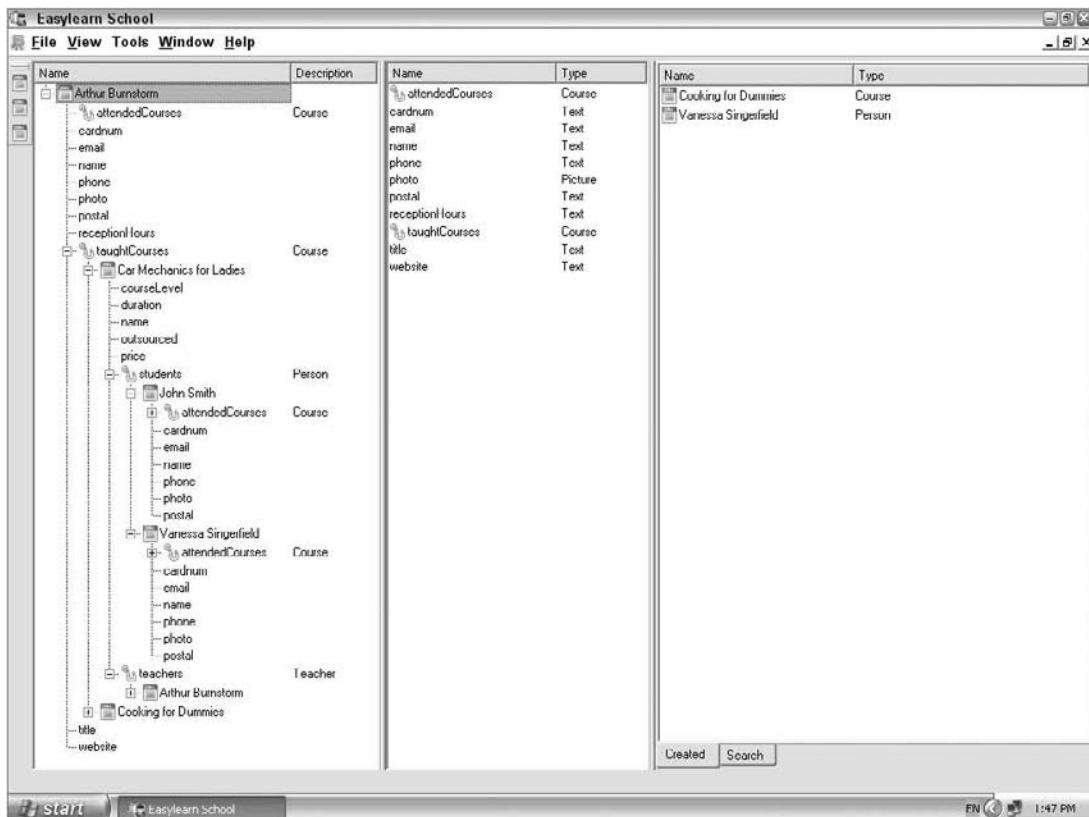


Figure 5-10: An example of an object space browser, where objects of the specializing class have slots for the properties inherited from the generalizing class.

- ❑ Objects of the specializing class are substitutable for objects of the generalizing class. For example, a link of the association attends between a course and a teacher can be created in

exactly the same way as between a course and an ordinary (non-teaching) person (for example, by dragging a teacher to the slot students of a course), without any additional programming effort. Note also that a course can have both teachers and ordinary persons as its students. Although they are direct instances of different classes, the course looks upon them as persons only, and does not see any difference between them.

- Objects of the specializing class are capable of doing something that direct instances of the generalizing class are not. For example, a link of the association `teaches` can be created between a course and a teacher, but not between a course and an ordinary person because an ordinary person is not a kind of a teacher.

Section Summary

- The application should demonstrate the following interactive manifestations of the generalization/specialization relationship, without any additional programming effort:
 - Objects of the specializing class have slots for all properties inherited from the generalizing class.
 - Objects of the specializing class are substitutable for objects of the generalizing class. Whatever can be done with an object of the generalizing class can also be done with an object of the specializing class.

FAQ

Q: Can a generalizing class have several specializations?

A: Yes, it can, and this is often the case.

Q: Can a base class have its base class, and so on, and is there any limitation to the size of generalization/specialization hierarchies?

A: A class can have its base class, which can have its base class, and so on. Note that transitivity of inheritance and of the substitution rule always holds.

There are no language limitations to the size of a class hierarchy, either in its depth (the longest chain of generalization/specialization relationships) or its width (the number of descendants of a class). However, very deep class hierarchies tend to be impractical and difficult to understand and maintain. This is why it is recommended to keep the hierarchies reasonably sized in both dimensions, where “reasonably” does not have any particular counterpart in concrete size, but is rather an ad-hoc estimation.

Q: Can a class be a specialization of several classes?

A: Both standard UML and OOIS UML allow a class to be a specialization of several base classes. This is traditionally called *multiple inheritance*. However, although it may look like a simple generalization of single inheritance, it may imply some semantic or implementation problems. This is why some OO programming languages do not support it.

Part II: Overview of OOIS UML

- Q:** How can I recognize a generalization/specialization relationship in the problem?
- A:** It is, again, a matter of conceptual modeling, which is not a deterministic process. What is worse, many real-world problems may easily lead to a misused generalization/specialization relationship. Some guidelines exist, fortunately. You will read more on this topic in section “Identifying Generalization/Specialization Relationships” in Chapter 18.
- Q:** How can a generalization/specialization relationship be implemented in a relational database?
- A:** There are several ways to map a set of classes related with generalization/specialization relationships to a relational model. One is similar to ER modeling as described in Chapter 23. The reader is encouraged to discover the others and to compare them with the given one.
- Q:** According to the given Easylearn school model, there are two associations, which imply that a teacher can attend and teach courses at the same time, but can a teacher attend and teach the same course at the same time? If the teacher can do that, why is it so because it makes no sense, and how can that be prevented in the system?
- A:** By the default semantics, there cannot be two links of the same association between the same two objects. But the default semantics do not prevent two links of different associations from connecting the same two objects. Therefore, the model developed so far really allows a teacher to attend and teach the same course. If such cases should be prevented, a *constraint* should be introduced into the model. You will read more on this topic in section “Consistency Rules” later in this chapter.

Operations

This section introduces several basic behavioral modeling concepts — operations, methods, and actions.

Requirements

The administrators of the Easylearn school may want to know the revenue that is collected from the students of each course. The revenue from a course is calculated by summing up the amount that each student of the course has to pay for attending that course. Basically, a student pays the price of the course that is recorded in the corresponding attribute of the course. However, the school offers some discount opportunities:

- If a student attends less than three courses, the student pays the full price for each attended course.
- If a student attends three to seven courses, the student pays 80 percent of the price for each attended course.
- If a student attends more than seven courses, the student pays 50 percent of the price for each attended course.

Concepts

This discussion examines operations, methods, and actions, as basic behavioral modeling concepts in OOIS UML. In OOIS UML, methods are implemented in a detail-level language.

Operations

Operation is a feature of a class representing a specification of a service that can be requested from any instance of that class in order to activate an associated behavior. Operation is the first behavior-related

UML concept presented here; so far, you have learned about structural concepts only. An operation is an element of the conceptual model, precisely, a behavioral feature of a class. It specifies that a service can be requested from any (direct or indirect) instance of that class by *invoking* that operation for that instance. An operation specifies that service, but does not specify *how* that service will be fulfilled. It has a name, and may have formal parameters and a return type.

At run-time, an operation of an object may be invoked. The actual arguments are then supplied. The arguments are references to instances of data types or to objects of classes. The invocation of the operation is manifested by the behavior specified as the implementation of the operation, and which provides the requested service.

In the UML notation, operations are specified inside the symbol for a class, in a separate compartment. They are rendered as lines of text that include the operations' names, parameters, and return types, as shown in Figure 5-11.

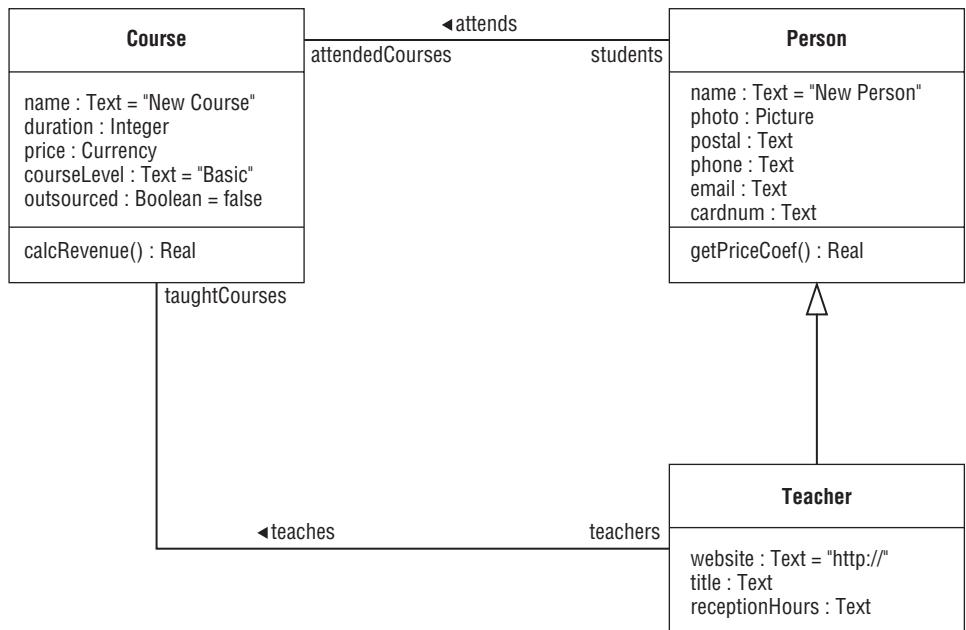


Figure 5-11: The Easylearn class model with operations specified in the classes Course and Person.

The given requirements can be fulfilled by two operations:

- **Operation getPriceCoef of the class Person** — When invoked for an object of this class, this operation should calculate the coefficient that is to be applied on the prices of the attended courses for that person. The coefficient should be calculated according to the school discounting policy, which takes into account the number of the courses attended by the person. Precisely, the implementation of the operation should retrieve the number of links of the association attends linked to the person and return a number (0.5, 0.8, or 1.0) according to the described discounting policy.

Part II: Overview of OOIS UML

- ❑ **Operation calcRevenue of the class Course** — When invoked for an object of the class `Course`, this operation should calculate the revenue for that course. The revenue should be calculated by traversing all the students of that course (that is, the objects of the type `Person` linked to the course by links of the association `attends`), obtaining the price coefficient by invoking the operation `getPriceCoef` of each such object, multiplying the coefficient by the price of the course, and summing up all such products cumulatively.

Note that an operation is just the specification of a service that can be requested from objects of a class. It does not specify *how* this service is fulfilled. Therefore, the current model has specified only the services that can be requested from the objects of classes `Course` and `Person`, but not how those services will be implemented.

Section Summary

- ❑ *Operation* is a feature of a class representing a specification of a service that can be requested from any instance of that class in order to activate an associated behavior.
- ❑ Operation is a behavioral feature of a class. It has a name, and may have formal parameters and a return type.
- ❑ At run-time, an operation of an object can be invoked. The actual arguments are then supplied. The invocation of the operation activates the behavior specified as the implementation of the operation.

Methods, Detail-Level Language, and Actions

An implementation of an operation is called a *method*. A method is the specification of behavior that is activated when the operation of an object is invoked.

There is no special notation for methods within class symbols. They are incorporated in the model and related to their operations by internal relationships maintained by the modeling tool. It is up to the modeling tool to provide a means for specifying methods of operations.

In each class, at most one method can be attached to each of its own or inherited operations. If a class does not attach a method to an operation of that class, the operation is called *abstract*. An abstract operation is shown in italics in diagrams. Such a class is then also abstract because it cannot have direct instances (in UML, it is not allowed to create objects without behavior defined for each specified service). If a class does not attach a method to an inherited operation, it inherits the method from its base class (transitivity is assumed), unless the operation is abstract in its base class, in which case the operation remains abstract.

In OOIS UML, methods can be specified using a *detail-level language*. The code of a method written in the detail-level language includes *actions* upon the underlying object space (such as, for example, creating or destroying objects and links, reading or writing attribute values, and calling operations of objects). The ordering of execution of the actions is determined by *control flow constructs*, such as conditional statements (that is, `if-then-else` constructs), loops, and so on. The method can also hold *local variables* that refer to objects or data values, and are used locally within the method to temporarily store the needed references

and values. In official UML parlance, such a detail-level language is also known as an *action language*, while both terms can be used interchangeably.⁷

In OOIS UML, one predefined detail-level language is *OOIS UML native detail-level language*. It is a Java-like language tailored for use in OOIS UML. Because it is used for specifying methods and other details of models, the syntax of that part of Java (that is, expressions and statements that can only appear within bodies of Java methods) is taken and extended for this purpose.

On the other hand, in OOIS UML, the detail-level language can also be an existing procedural or object-oriented programming language, like Java, C++, C#, or similar. In that case, the code of the method can access the object space in a restricted manner, through the application programming interface (API) of OOIS UML run-time environment that provides such services (that is, provides the “system calls” for actions upon the object space). More precisely, the code in such a detail-level language contains the control-flow constructs, local variables, expressions, and other specific elements of that programming language, but access the object space through the services of OOIS UML run-time environment.

For the purpose of the ongoing Easylearn example, the code for the methods for the two mentioned operations is given here; Java is used here as the detail-level language. (The code would look very similar in C++ or C#, and should be quite readable to those familiar with these languages.)

The method for the operation `getPriceCoef` of the class `Person` looks like this:

```
double coef = 1.0;
long numCourses = this.attendedCourses.size();
if (numCourses < 3) coef = 1.0;
if (numCourses >= 3 && numCourses<=7) coef = 0.8;
if (numCourses > 7) coef = 0.5;
return new Real(coef);
```

The method for the operation `calcRevenue` of the class `Course` looks like this:

```
double sum = 0.0;
for (Person s : this.students.read())
    sum = sum + s.getPriceCoef().toDouble();
sum = sum * this.price.read().toDouble();
return new Real(sum);
```

The code is easy to understand if some additional explanations are given:

- Java keywords are printed in bold to be distinguishable from user-defined identifiers.
- `double` is a Java built-in data type for floating-point numbers, and `long` for long integers.
- `this` is an implicit reference that always refers to the object whose method is being executed at run-time (also called the *host* object).

⁷This discussion still uses the term “detail-level language” because it indicates the purpose of the language (that is, to specify details, not only of methods, but also of some other parts of models). In addition, the term “action language” may be unwittingly associated with the exclusive use of actions, but methods may also contain control-flow statements and other elements that do not fall into the category of UML actions — that is, that map to some other parts of the language (for example, activities) or are completely external to UML.

Part II: Overview of OOIS UML

- ❑ . is the member-access operator in Java. Thus, `this.attendedCourses` refers to the slot of the host object for the association end `attendedCourses`. Similarly, `this.price` refers to the slot of the host object for the attribute `price`.
- ❑ `size` is the operation of OOIS UML API that returns the number of elements in a collection designated by a slot. `read` is the operation of OOIS UML API that reads and returns a value of a slot from the object space. `toDouble` is an operation that provides a conversion from OOIS UML type `Real` into the detail-level language built-in type (“native type”) `double`.
- ❑ `for` in this example is a construct for iterating through a collection. `for (Type elem : inCollection)` iterates through the collection `inCollection`. The current element of the iteration is referred to from within the body of the loop by the identifier `elem` and is of type `Type`.
- ❑ `s.getPriceCoef()` invokes the operation `getPriceCoef` of the object referred to by `s`.

As you can see from the given examples, a method specified in the detail-level language can have several types of contents:

- ❑ **Actions** — These are fundamental units of behavior that read or modify the underlying object structure (the typed graph) or affect behavior by, for example, invoking operations of objects.
- For example, there are actions such as “Create a new object of the given class,” “Delete the given object,” “Read and return an attribute value of the given object,” “Read and return the collection of objects linked to a slot of the given object,” “Invoke the given operation of the given object with the given arguments and return its result,” and so on.

Standard UML specifies the set of available actions and their semantics, although not completely. OOIS UML reduces the set of available actions, but defines the semantics completely and in a slightly different way, so that it can be mapped to standard UML’s action semantics. The entire set of available actions will be given throughout this book.

In the given examples of methods, actions are, for example, the following:

- ❑ `this.attendedCourses.read()` — This is the action “Read a slot of the given object and return the collection of linked objects,” which reads the slot `attendedCourses` of the host object referred to by `this` (the object whose method is being executed).
- ❑ `this.attendedCourses.size()` — This action returns the number of elements in the given slot (`attendedCourses`, in this case).
- ❑ `this.price.read()` — This is the action “Read and return the value of the slot of the given object,” which reads the attribute value `price` of the host object referred to by `this` (the object whose method is being executed).
- ❑ `s.getPriceCoef()` — This is the action “Invoke an operation of the given object and return its result,” which invokes the operation `getPriceCoef` of the object referred to by `s` (the current object in the iteration through the collection).

As you can see from the given examples, actions specify some processing upon some parameters that specify the input values of the actions. For example, the action “Read an attribute value” works upon the specified slot of the specified object. Similarly, the action “Delete object” works on a specified object as its parameter. These parameters of actions are called *input pins*.

On the other hand, some actions produce results. For example, the action “Create a new object of the given class” produces as its result a reference to the created object, so that the actions that follow can work with that object. Similarly, the action “Read an attribute value” results in the

value of the attribute. These results of actions are called *output pins* of actions. It is specific to each particular action which input and output pins it has. This is defined by the language and cannot be modified by the developer.

- ❑ **Control structures** — They define the control flow of the method, that is, determine the ordering of execution of actions and other parts of the method. If a classical programming language is used, these are often specified by the statements in the language. For the given example, these are the constructs `if`, `return`, and `for`.
- ❑ **Other parts specific to the detail-level language** — This includes declarations (for example, `double coef = 1.0;`), expressions (for example, `numOfCourses < 3`), and others.

In summary, the detail-level language is the host language for the actions because it embodies actions within control structures and other computation, as well as the *surface language*, because it provides a concrete notation for the actions. Note, however, that the detail-level language does not need to be an existing textual programming language. It can be another ad-hoc, specific language, or even a language with a visual notation.

The purpose of using OOIS UML native detail-level language in some examples throughout this book is to focus the reader's attention on the core meaning of a presented feature of the language, without distractions caused by syntactic or minor semantic peculiarities of other programming languages that also can be used for the same purpose. The syntax and semantics of OOIS UML native detail-level language are fine-tuned for the semantics of OOIS UML profile. Usually, although the code shown in OOIS UML native detail-level language can be easily mapped to the code in another traditional OO programming language that can also be used for the same purpose, the former is usually more concise than the latter. This also keeps the material presented in this book less dependent on a selected detail-level language.

Of course, many users of OOIS UML profile would prefer using their favorite OO programming language as the detail-level language. Actually, many examples in the book are also written in one of the traditional OO programming languages. This is most often the case when those code snippets do not contain peculiarities of these languages that would distract the reader's attention from the main point, and when they would look more or less the same in all these languages. In addition, an implementation of the profile can also opt for one or the other detail-level programming language and completely ignore OOIS UML native detail-level language. Whichever language is used, however, the set of available actions is the same and their coupling to the rest of the model is formal and complete.

Section Summary

- ❑ An implementation of an operation is called a *method*. A method is the specification of behavior that is activated when the operation of an object is invoked.
- ❑ Methods specify behavior in terms of *actions* and *control structures*. An action is an elementary, fundamental unit of behavior that affects the underlying object space. Control structures define the ordering of action execution.
- ❑ In OOIS UML, methods are specified using a detail-level language (also known as the *action language*).

Continued

- ❑ One predefined detail-level language is *OOIS UML native detail-level language*. It is a Java-like language for specifying methods, with several OOIS UML-specific extensions.
- ❑ If an existing programming language is used as the host detail-level language, the actions are specified using OOIS UML API, and control structures are taken from the host language.

Interactive Manifestations

When the new model with two added operations and their methods is executed, the following is obtained in the generic execution environment:

- ❑ The operations as services of objects are available in the right-click pop-up context menu of objects (see Figure 5-12). The pop-up context menu for each object of the class Course will have the service (operation) calcRevenue, and the menu for each object of the class Person will have the service (operation) getPriceCoef. Moreover, the menu for each object of the class Teacher will also have the operation getPriceCoef because of inheritance.

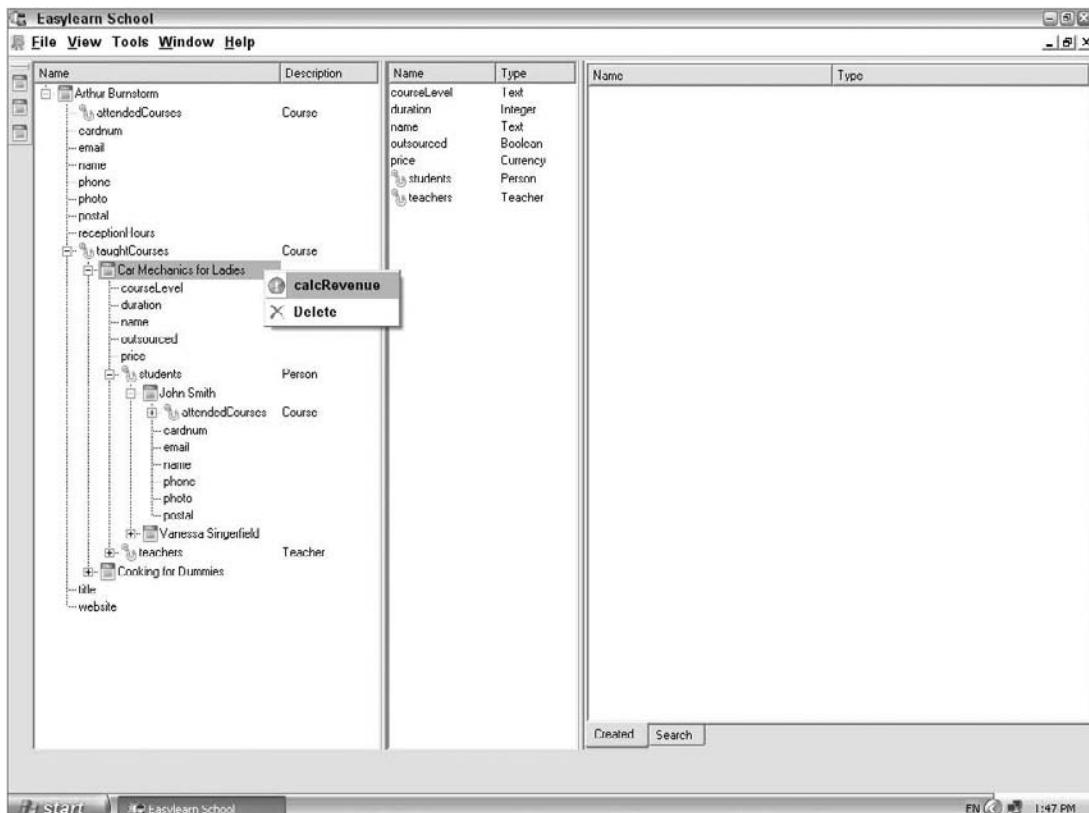


Figure 5-12: The object browser with objects with operations. When an operation is invoked, its result is displayed in a message box.

- ❑ The operations can be invoked by selecting an operation from the pop-up context menu. When the operation is invoked, the method attached to it is executed and the returned result is displayed in a message box. For example, if the operation `getPriceCoef` of a teacher is executed, it will return the price coefficient according to the number of the courses attended by the teacher and the general discounting policy (that is, according to the behavior specified in the inherited method). Similarly, if the operation `calcRevenue` of a course is invoked, it will give the calculated revenue for the course. It should be noticed that the calculation will take into account all students of the course in the same way, be they teachers or not. Teachers behave just like other persons because of inheritance of methods.

Section Summary

- ❑ The interactive manifestations of operations and methods are:
 - ❑ Operations of objects can be invoked, when the corresponding methods are executed.
 - ❑ Inheritance of operations and methods is assumed.

FAQ

- Q:** Why is the separation of operation and method so strongly emphasized?
- A:** These two concepts are so strictly separated because it leads to better flexibility. As explained already, a derived class always inherits an operation from the base class, but may also attach a new method to the same operation. If it does not attach a new method, the method from the base class is inherited. If it attaches a new method, however, that method will be activated if the operation is invoked for an instance of that class, even if the caller has accessed the object as an instance of the base class. This mechanism is called *polymorphism* and will be explained in the next section in detail because it is one of the key features of the object paradigm.
- However, the separation of operation and implementation is much older than the concept of polymorphism. Even traditional procedural languages supported it, although not in such an emphasized way. Namely, what was usually called the *declaration* or *signature* of a sub-program consisted of the sub-program's name and types of parameters only. The implementation carried the body of the sub-program. However, the motivation was somewhat different — reducing compilation dependencies among software modules (that is, the modules of the caller and of the sub-program). Although the same motivation still remains, the concept of polymorphism is a much stronger reason for such clear separation of concepts in the object paradigm.
- Q:** What types of parameters and return values are available in OOIS UML? How are they passed, by value or by reference?
- A:** In OOIS UML, parameters and return values of operations can be objects of classes, instances of data types, and values of native types available in the host detail-level language.

Part II: Overview of OOIS UML

Instances of classes and data types are always accessed indirectly, over references, in order to support the substitution rule at all places. Therefore, they are always passed by reference. For the types specific to the used host detail-level language, the passing mechanism is defined by that language and is not a matter of OOIS UML.

Polymorphism

This section introduces one of the most effective mechanisms in the object paradigm — polymorphism.

Requirements

The administrators of the Easylearn school now want to introduce a new element to the discounting policy, offering special discounts to teachers when they attend courses. Teachers are “special guests” of the school and should have special treatment when paying for courses. Therefore, the school offers a special policy for its teachers:

- If a teacher attends less than four courses, the courses are completely free for the teacher.
- If a teacher attends four or more courses, the teacher pays 50 percent of the price for each attended course.

Concepts

You have already learned that a derived class inherits a method for an inherited operation, as in the case of the class Teacher and the operation `getPriceCoef`. However, a derived class can provide its own method for an inherited operation, offering different behavior for the same service. In other words, a derived class can *replace* an inherited method for an inherited operation. This is called *redefining* or *overriding* the operation’s method.

There is no standard notation for specifying redefinition of methods in UML class diagrams. It is the responsibility of the UML modeling tool to provide a way to specify the new redefining method for the operation in the derived class.

According to the requirements, the method of the operation `getPriceCoef` should be redefined in the class Teacher, with the method coded to support the special discounting policy for teachers:

```
double coef = 1.0;
long numCourses = this.attendedCourses.size();
if (numCourses < 4) coef = 0.0;
if (numCourses >= 4) coef = 0.5;
return new Real(coef);
```

At run-time, an operation of an object may be invoked. The invocation of the operation activates execution of the method most relevant for the class of which the object is a direct instance, regardless of how the object was accessed (that is, possibly as an instance of a base class). In other words, a client that invokes an operation of a supplier object can access the supplier object as a generalized entity (that is, as an object of the base class). In that case, the method of the derived class will be executed, if it exists. This mechanism is called *polymorphism*.

Consequently, the method of the operation `calcRevenue` of the class Course does not need any modification. This method navigates through the linked persons and invokes the polymorphic operation

`getPriceCoef` of the objects at the opposite side of the links. These opposite objects are regarded as generalized persons. However, some of them can be teachers and the others are not (but are other persons). Each of them will respond to the invocation in the way that is specific for its class (that is, with its own method), and will provide the price coefficient according to the extended requirements:

```
double sum = 0.0;
for (Person s : this.students.read())
    sum = sum + s.getPriceCoef().toDouble(); // Underlined is
                                                // where polymorphism occurs!
sum = sum * this.price.read().toDouble();
return new Real(sum);
```

This way, the behavior of the system has been modified by simply adding a new method to the model (that is, by redefining an operation's method at the supplier side), without modifications of any part, especially not the client side (the class `Course` and the method for its operation `calcRevenue`).

In general, the purpose of polymorphism is to make the clients that invoke an operation independent of the variation of the operation's implementation. In other words, the client is spared from knowing the specialties about the supplier of the service. Instead, the client tends to regard the supplier as a generalized thing, and accesses it through its generalized interface. The specialties of different kinds of suppliers are incorporated in the polymorphic operations and their possibly redefined methods. This way, the interfaces between clients and suppliers become looser, and, therefore, more controllable.

This is a key point to constructing flexible software. Modification of the behavior of the supplier side can be achieved by *adding* parts of software (that is, redefining methods in derived classes), and not *modifying* parts of software, which is often error-prone and risky. Put another way, the client does not experience any modification if a new class at the supplier side is added in the hierarchy, or a polymorphic operation's method is redefined in a derived class in the supplier-side hierarchy, and yet the software behaves differently. This mechanism is one of the main features of the object paradigm.

Section Summary

- ❑ A derived class can provide its own method for an inherited operation, offering different behavior for the same service. This is called *redefining* or *overriding* the operation's method.
- ❑ The invocation of the operation is manifested by execution of the corresponding redefining method of the derived class of which the object is a direct instance, regardless of how the object was accessed (that is, possibly as an instance of a base class). This mechanism is called *polymorphism*.

Interactive Manifestations

The modified model from this section does not affect the presentation of the object space browser because the objects offer the same services (operations) as before. Only the implementation of an operation has been changed. However, the implementation is not part of the objects' interfaces, and is thus not visible in the browser.

Part II: Overview of OOIS UML

On the other hand, the new application provides different behavior than before. For example, if the operation `calcRevenue` is invoked for the same object as in the previous section, and the object has both teachers and other persons as its students, the result will be different because teachers now behave differently when asked for the price reduction coefficient.

Section Summary

- ❑ Redefined methods and polymorphism affect the behavior of the system because different behavior is activated upon invocation of polymorphic operations.

FAQ

- Q:** Does a polymorphic operation need to be specially tagged in the model, like, for example, virtual functions in C++?
- A:** In some OO programming languages, operations have to be specially tagged to be polymorphic. For example, in C++, an operation must be specified as *virtual*, or otherwise cannot be overridden. However, the modern understanding of the object paradigm assumes that polymorphism is a crucial concept and that operations should be polymorphic by default. Therefore, in some newer languages (such as Java and UML), operations are polymorphic by default.
- Q:** What if some operation should not be polymorphic?
- A:** If an operation's method should be forbidden for redefinition (which is a much rarer situation than the opposite), the operation should be tagged as *leaf* in the model.

Consistency Rules

This section explains how consistency rules can be imposed by specifying constraints.

Requirements

So far, the Easylearn school information system allowed a teacher to teach an arbitrary number of courses. Let's assume that the administrators of the school want to constrain the number of courses taught by a teacher to at most three, for whatever reason.

Moreover, the administrators want to pose two other constraints. First, for the obvious reason, a teacher cannot attend the courses he or she teaches at the same time. Second, a person cannot attend a course that is outsourced if the person has not provided his or her credit card number because the Easylearn school must regulate the payment to the provider of the course.

Concepts

This discussion examines the basic UML concepts of multiplicity and constraints.

Multiplicity

A special adornment of each association end is *multiplicity*. It is a specification of the range of valid cardinalities of the collection of objects in a slot of that association end.

For example, the first stated requirement could be met using the concept of multiplicities of association ends. As specified in the model shown in Figure 5-13, an object of the class Teacher can be linked with from zero to at most three objects of the class Course by links of the association teaches. Note that such a constraint does not affect cardinalities of other associations' links. A teacher can attend (over the inherited association attends) an arbitrary number of courses, even those that he or she teaches.

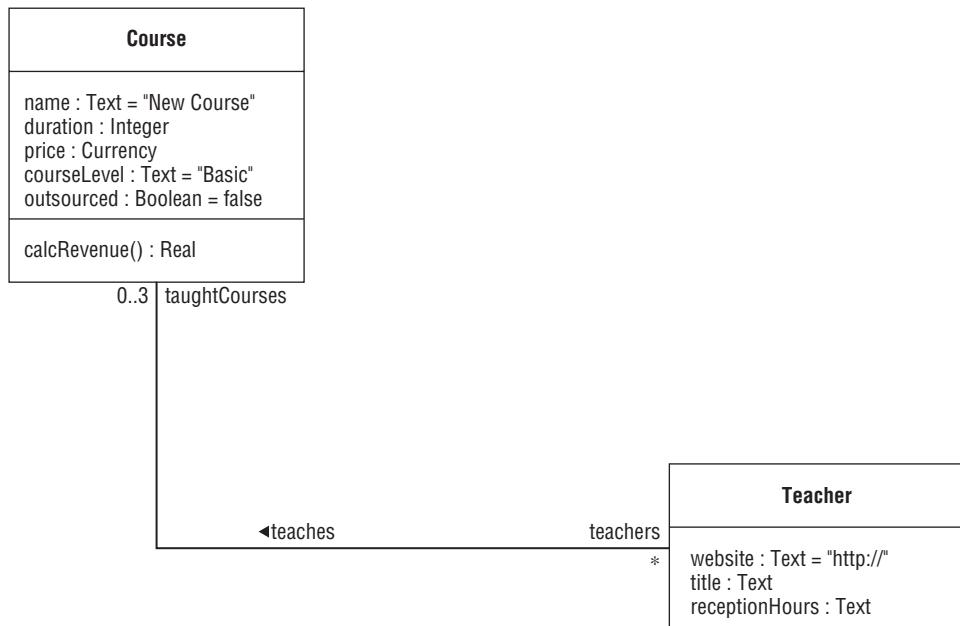


Figure 5-13: Structural constraints specified by the multiplicities of association ends

In general, multiplicity specifications can be as follows:

- 0..1, meaning zero or one.
- 1..1 (or 1, for short), meaning exactly one.
- 0..* (or *, for short), meaning from zero to arbitrarily many (* means unlimited).
- 1..*, meaning from at least one to arbitrarily many.

Part II: Overview of OOIS UML

- ❑ $m..n$, meaning from m to n (inclusive), where m can be a non-negative integer and n can be a positive integer greater than or equal to m or * (unlimited). If m and n are equal, this can be abbreviated with m only.

The default multiplicity of each association end in OOIS UML is * (unlimited).

Section Summary

- ❑ A special adornment of an association end is *multiplicity* — a specification of the range of valid cardinalities of the collection of objects in a slot of that association end.
- ❑ Multiplicity specification has a notation $m..n$, meaning that the valid cardinality is from m to n (inclusive), where m can be a non-negative integer and n can be a positive integer greater than or equal to m , or * (unlimited). If m and n are equal, this can be abbreviated with m .

Constraints and OCL

The remaining two constraints posed in the “Requirements” section are too specific for the described problem domain to be incorporated in UML as a construct of general applicability. Therefore, UML introduces the concept of *constraint*, which is an arbitrary consistency rule that can be attached to one or more model elements of any kind. A constraint represents additional semantic information attached to the constrained elements. It is a restriction, given as a Boolean expression, that must be satisfied by the correctly functioning system. Constraints are used to specify additional restrictions to the constrained elements beyond what other language rules for those elements impose.

Constraints in standard UML can be specified in any formal or informal language, including a natural language, a traditional textual programming language, a visual language, or a standard predefined language that is part of the UML. The interpretation of the constraint is the responsibility of the modeling tool or the human reader. Usually, constraints are textual expressions written between braces {} and attached to model elements.

In order to be formal and executable in OOIS UML, constraints must be expressed using a machine-interpretable language. It can be, for example, the same detail-level language used for specifying methods, or the predefined *Object Constraint Language* (OCL), which is a standardized part of UML. OCL is a formal language like the classical mathematical quantifier calculus, adapted for UML and navigation through the object structure. It can be used to specify Boolean expressions, such as “for each X holds ...” or “there is at least one X for which holds ...” and so on, which evaluate to `true` at certain moments during execution of the system.

A more complete specification of the syntax and semantics of OCL and OOIS UML constraints will be given later in the book. This section simply illustrates the usage of these concepts for the stated requirements.

Figure 5-14 shows a constraint attached to the class Teacher:

```
self.taughtCourses->forAll(c | self.attendedCourses->excludes(c))
```

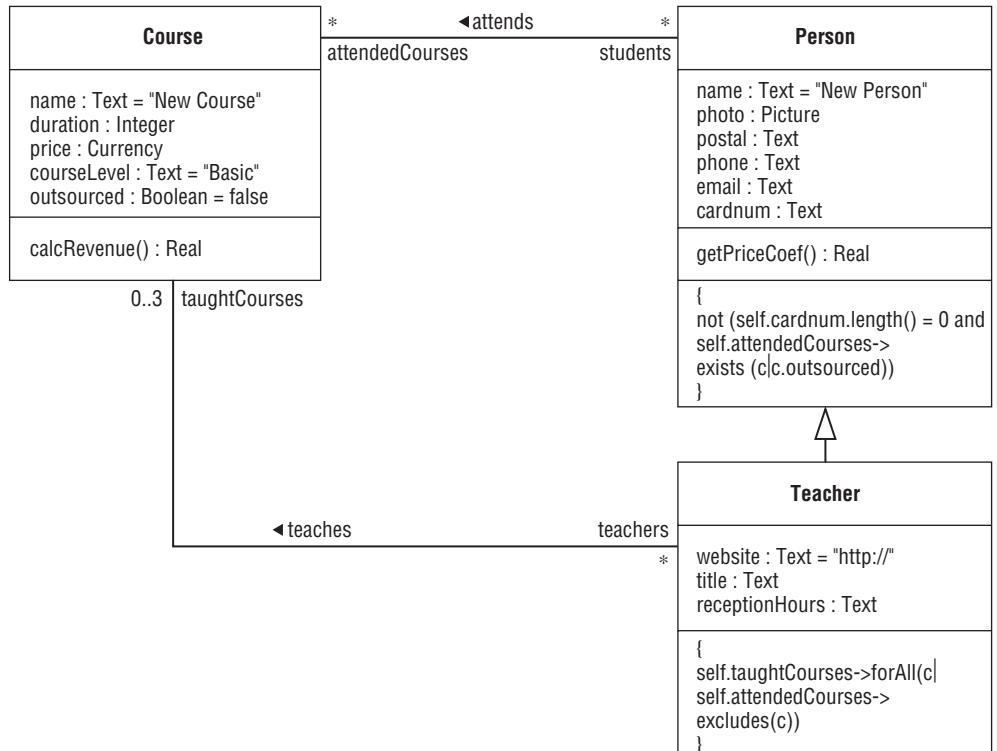


Figure 5-14: Structural constraints specified by OCL expressions

This expression should evaluate to true for each object of the class Teacher, referred to as `self` in the expression.⁸ It is interpreted in the following way. For each object, referred to as `c`, in the collection `self.taughtCourses`, it must hold that the collection `self.attendedCourses` excludes `c`. This is a formal statement of the requirement that a teacher cannot attend the courses he or she teaches, and vice versa.

Similarly, Figure 5-14 also shows a constraint attached to the class Person, which specifies that a person cannot attend an outsourced course, if the person has not provided his or her credit card number:

```
not (self.cardnum.length()=0 and
     self.attendedCourses->exists(c | c.outsourced))
```

⁸OCL uses `self` to refer to the host object, but some popular OO programming languages use `this` for the same purpose. This is why OOA UML native detail-level language opted for the latter.

Part II: Overview of OOIS UML

In fact, multiplicity constraints are just specific predefined constraints that have all the semantics of usual constraints. For example, the multiplicity constraint 0..3 at the association end `taughtCourses` can be expressed in OCL within the context of the class `Teacher` as:

```
self.taughtCourses->size()<=3
```

Section Summary

- ❑ A *constraint* is a Boolean expression attached to a set of constrained elements that represents a restriction that must be satisfied by the correctly functioning system.
- ❑ One way to specify constraints is through *Object Constraint Language* (OCL).
- ❑ OCL is a formal language like the classical mathematical quantifier calculus, adapted for UML and navigation through the object structure.

Constraints as Objects

The constraints described so far are incorporated in the model. They are checked implicitly by the execution environment at certain moments in run-time. Therefore, they ensure a permanent consistency of the object space, as specified by the modeler. The details of this mechanism will be given later in this book.

However, sometimes there is a need to simply check and report whether the object space satisfies a certain criterion at run-time. For example, the user might want to know whether there is a course attended by teachers only, or whether all persons have unique names. Such criteria are not obligatory rules for the system's consistency, but simple queries about some logical properties of the object space.

For that purpose, the user can create and define OCL constraints at run-time. Such constraints are actually created as ordinary objects of a special built-in class `OCLConstraint` that is part of OOIS UML model library. One of the attributes of this class provides the OCL expression that specifies the constraint to be checked when explicitly requested from the user. At run-time, it represents a loose constraint because it is not implicitly ensured by the execution environment. Instead, it is explicitly evaluated when the user wants to check some additional properties of the object space. When such a constraint is violated, it does not necessarily mean that the system is in an inconsistent state, but it may be simply a warning to an unusual situation in the system.

If the OCL constraint specified within such an object uses the reference `self`, it must be applied to a single object as its parameter, when the reference `self` refers to that object. Otherwise, the constraint may be simply evaluated without parameters when it checks a general condition of the system. When a constraint is evaluated, its result may be just `true` (if the constraint is satisfied) or `false` (if the constraint

is violated). Optionally, in the case of a false answer, the execution environment may provide more details about the parts of the structure that violate the constraint.

For example, the following OCL constraint checks whether all persons have unique names:

```
Person.allInstances()->forAll(p1,p2| p1<>p2 implies p1.name<>p2.name)
```

Similarly, the following constraint should be applied to a course to see whether it has any student that is not a teacher:

```
self.students->exists(s| not s.oclIsKindOf(Teacher))
```

A constraint as an object has its name, which helps the user to identify it at run-time, and its description, which helps the user to understand its purpose. A constraint is specified in one of its attributes, written in OCL.

The generic GUI of the execution environment allows all generic operations on constraints as objects (for example, create and delete). Additionally, a constraint can be edited and evaluated, when its result is displayed in a special message box, optionally with an additional dialog providing more details about the violation of the constraint. Such usage will be described in the next section.

Section Summary

- ❑ The constraints incorporated in the model are checked implicitly by the execution environment at certain moments in run-time. They are restrictions that ensure a permanent consistency of the object space, as specified by the modeler.
- ❑ OCL constraints can also be defined as objects of the built-in class `OCLConstraint` at run-time. Such objects represent requirements to check certain properties of the object space at the moment the user wants to do that. They just report possible consistency violations, but do not ensure consistency.
- ❑ Constraints, as objects, have names and descriptions, which help the user to identify and understand them. The constraints can be created, deleted, edited, and evaluated.

Interactive Manifestations

When the new model is executed, the constraints introduced in the model are manifested in the following way.

Part II: Overview of OOIS UML

First, an object of the class Teacher cannot be linked to more than three courses by links of the association teaches. For example, if the user tries to link a teacher to the fourth taught course, the execution environment will display a warning message and prevent the action (see Figure 5-15). This is because the execution environment checks the constraint specified by the multiplicity 0..3 at the end taughtCourses of the association teaches on link creation.

This also holds true for the other two constraints incorporated in the model. If the user wants to perform an action that violates a constraint, the execution environment issues a message and prevents the action. For example, if the user wants to attach a teacher to attend a course that he or she already teaches, the system will issue a warning and prevent the action. Similarly, the system will prevent linking a person (be it a teacher or an ordinary person) to an outsourced course, if the person does not have a credit card number.

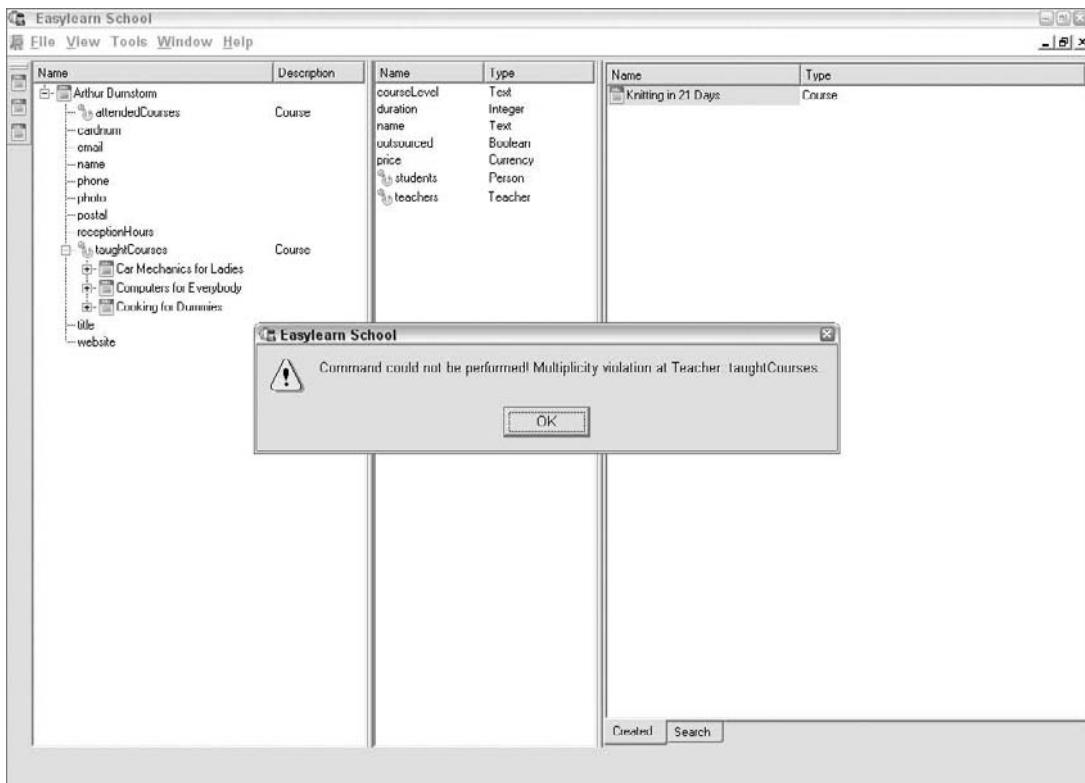


Figure 5-15: The execution environment prevents all user actions that violate the constraints incorporated in the model.

Finally, the execution environment supports manipulation with OCL constraints as all other objects at run-time (see Figure 5-16). The user can create a new OCL constraint as an object in the usual way, and then set the values for its name and description attributes. The user may also specify the OCL source of the constraint in a special editor. Finally, the user can evaluate the constraint and see its result.

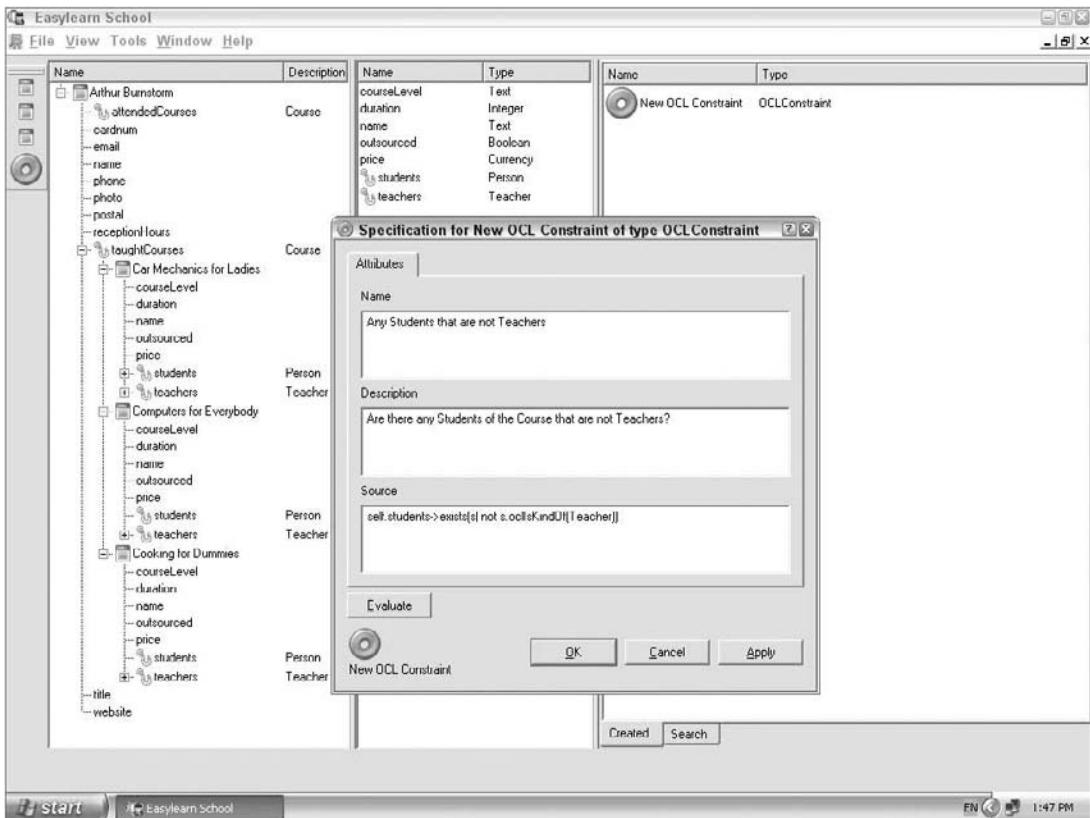


Figure 5-16: The execution environment supports creation and evaluation of OCL constraints as objects.

Section Summary

- The execution environment prevents all user actions that violate the constraints incorporated in the model and displays warning messages.
- The execution environment allows manipulations with OCL constraints as objects that can be created and evaluated at run-time.

FAQ

- Q:** When attempting to specify OCL constraints for the given requirements, somewhat different expressions come to mind. Can they be correct, too?
- A:** Of course. Obviously, there may be many expressions with equivalent meaning. First, the usual logic transformations (for example, De Morgan's laws) can be applied to an OCL expression

Part II: Overview of OOIS UML

(because it is a logical expression) to get equivalent expressions. For example, consider the following expression:

```
self.taughtCourses->forAll(c| self.attendedCourses->excludes(c))
```

This can be rewritten in an equivalent form, with negating predicates and complementing the quantifier `forAll` into `exists`:

```
not (self.taughtCourses->exists(c| self.attendedCourses->includes(c)))
```

Or, it can be rewritten using the `intersection` operation on sets of objects:

```
self.taughtCourses->intersection(self.attendedCourses)->isEmpty()
```

This means that the intersection of the set of the attended courses and the set of the taught courses of the teacher must be empty.

Similarly, the same requirement can be specified using different navigation over the object space. For the given example, the constraint can be expressed within the context of an object of the class `Course` (`self` now refers to a course), being:

```
self.students->intersection(self.teachers)->isEmpty()
```

This means that the intersection of the set of students and the set of teachers of the course must be empty.

Q: Can OCL expressions invoke operations of classes?

A: Yes, they can. Of course, the invocation of an operation within an OCL expression is polymorphic. However, the operation must not have side effects, meaning that it must not modify the object space. In other words, only the operations that are pure functions (do not have side effects) can be invoked within OCL expressions.

In general, OCL expressions have no side effects, because OCL does not define actions that modify the object space, UML actions that modify object space cannot be used within OCL constraints, and operations invoked within OCL expressions cannot have side effects (that is, their methods cannot incorporate actions that modify the object space).

The same holds for constraints expressed in any other formal language, too.

Q: When exactly are the constraints defined in the model evaluated? Is there a notion of an explicit trigger in OOIS UML that activates execution of a constraint, as in traditional RDBMSs?

A: The precise run-time semantics of constraints specified in the model will be defined later in this book. Basically, the evaluation of a constraint is triggered implicitly, whenever an action being executed modifies (an instance of) a model element to which the constraint is attached. Therefore, there are no explicit triggers defined by the designers, but they are rather implicit. After an action has been executed, all constraints that are attached to the elements affected by that action are checked. This way, the designer of the system does not need to specify explicit triggers, but simply to attach constraints to certain elements of the model. Any action that modifies any of these elements triggers the execution of the constraints. There are some variations of this rule of evaluating constraints on any action, however, and they will be explained later in this book. You will read more on this topic in Chapter 11.

- Q:** Does it mean that a constraint specified in the model must hold always (that is, at any time during execution of the system), or not?
- A:** No, this means that a constraint holds at *certain points in time*, or better to say during certain *intervals*. In other words, there are some intervals during which a constraint may not hold. One example is the very interval of the action execution. Because the execution of an action takes some time in reality, at some point in that time, a constraint may not hold because the object space is in transition from one consistent state to another. There are some other cases, too. However, the default semantics of actions and constraints guarantee that the constraints hold *before* and *after* the action execution. Moreover, no other concurrent executions of actions may affect the considered action execution. Therefore, an action execution can be considered *atomic* in these aspects. You will read more on this topic in section “Run-time Semantics” in Chapter 11.
- Q:** What exactly happens if a constraint specified in the model fails during the execution of the system?
- A:** The action execution that activated the execution of the constraint raises an exception and fails.
- Q:** Let’s say I need a “one-to-many” association between Department and Employee, where an Employee must be assigned to exactly one Department. The multiplicity constraint 1 at the Department end ensures that. However, the described semantics of constraints checked on each action execution prevent the reassigning of an Employee to another Department. Namely, to do that, two actions are required: the first (Destroy Link) will remove the existing assignment of the Employee, and the second (Create Link) will assign the Employee to a new Department. However, at the end of execution of the first action, the cardinality of the end will be 0, and the constraint will fail. On the other hand, if the multiplicity at the same end is defined to be 0..1, this problem will be solved, but another one appears. Such a multiplicity will allow someone to accidentally or deliberately unlink an Employee from any Department for a longer period, leading to an inconsistent system state, which is not allowed by the requirements. How should this be handled?
- A:** Indeed, the described semantics of constraints and actions lead to the described problem. As the given problem is very often in practice, OOIS UML offers solutions — for example, the action Relink Object. It performs the deletion of one link and the creation of another in one atomic action. However, a similar problem exists during object construction (initialization). When an object is being created, some actions must be performed to transform the object from an “empty” state into the consistent state satisfying all constraints. Before these actions, none of the constraints may hold, because, for example, no links exist. In general, some actions may lead to the consistent state at the end of object construction, but after some of them, the constraints are still not satisfied. Consider, for example, the multiplicity with the lower bound greater than 1 at one end. Several actions are needed to create links and achieve that cardinality, so all but the last action will not satisfy the multiplicity constraint. In order to cope with that problem (a frequent occurrence), OOIS UML allows a set of actions to be declared as a *constrained action group*, meaning that the constraints will not be checked at the end of each action execution as by default, but once all of the actions are completed.

6

Interaction and Querying

This chapter describes how the presentation and behavior of the application can be customized based on the information taken from the conceptual model, without any further coding, and even by demonstration. Additionally, it examines the querying feature of OOIS UML and a query language that exploits the object paradigm. The discussion continues with the same running example of the Easylearn school system that has been partly constructed in Chapter 5.

Customizing Presentation

This section explains how the appearance of the application can be customized at run-time, without any coding.

Requirements

So far, the presented examples have used the generic user interface of the system provided by the OOIS UML run-time environment by default. This is acceptable as long as you want to make quick prototypes of the system without any special effort made for customization of the user interface. On the other hand, this approach leads to a rather unattractive and less usable system, because the icons for objects are all equal, the browsers are rather generic and not tailored for specific users' needs, and so on. Now, let's customize the user interface in the following manner.

First, let's use different icons to represent objects of different classes, so that the objects of different classes are easily distinguished. Second, apart from the name of a person, let's make the e-mail address of the person appear as its description next to it, and the course level appear next to courses.

Finally, the generic tree-view browser is not clear enough because it provides a rather cluttered view over all properties of each object. Instead, let's shape the hierarchical tree view in several ways, depending on the perspective of the user. Namely, apart from the default browser used so

Part II: Overview of OOIS UML

far, users may want to have two other types of browsers, which provide different perspectives over the object space:

- ❑ **Courses and Students** — Within this type of browser, each course should have its students as its direct sub-nodes (instead of being sub-nodes of the students slot node). Each person (as a node in such a hierarchy) should have all its attended courses as its direct sub-nodes (instead of being sub-nodes of the attendedCourses slot node).
- ❑ **Courses and Teachers** — Within this type of browser, each course should have its teachers only as its direct sub-nodes (instead of being sub-nodes of the teachers slot node). A teacher (as a node in such a hierarchy) should have the following sub-nodes: e-mail address, Web site address, and reception hours, as well as the node for its taughtCourses slot. The other slots should not appear in order to improve readability. Additionally, instead of the e-mail address, the title of the teacher should appear as the description next to the teacher's name.

Using these two kinds of browsers, users can have a better overview of the object space. Depending on which information they are interested in, they will have different viewpoints.

Concepts

This section examines the basic OOIS UML concepts related to the customization of presentation.

GUI Item Configuration Settings

It is now clear that different elements of the overall system's representation within the GUI (such as objects, slots, classes, and so on) have something in common: they may be represented by icons, text may appear next to the icon, they may have their sub-nodes in tree views, they may have their specification dialogs that are open on a double-click, they may have their right-click pop-up context menus, they may react on drag-and-drop and other mouse events, and so on. Therefore, there is an inherent generalization of all these elements of the GUI — a *GUI item*. A GUI item is an element of the GUI that has all these properties and behavior, and that represents an element of the underlying object space or model.

In order to be easily customizable, the appearance and behavior of each GUI item should be parameterized, and these parameters should be modifiable by the modeler and/or the user, depending on who is interested in such parameterization. For this purpose, a *GUI item configuration setting* represents a set of presentational and behavioral parameters defined for one particular kind of element in the system. It specifies how the elements of that kind appear and behave in the application's GUI. For example, a configuration setting for the objects of class *Course* defines how these objects are rendered and react on mouse events in the application.

A GUI item configuration setting includes all relevant parameters that are used in the GUI for rendering elements, such as the following:

- ❑ Icons (small, large, for drag-and-drop, and so on)
- ❑ Texts that are displayed as the name, type, description, label, and tip of the element
- ❑ The way in which sub-nodes in a tree view are obtained from the element

- The specification dialog for the element
- The set of commands available in the right-click pop-up context menu for the element
- The behavior on double mouse click and drag-and-drop of another element onto the element

Some others are possible, too, if appropriate within the given GUI.

GUI item configuration settings are modeled with the built-in class `GUIItemSetting` from the OOIS UML model library. During the execution of the system, GUI item configuration settings exist as ordinary objects of this class in the system's object space. These objects can be accessed and manipulated as all other objects in the system. The GUI component of the system interprets the contents of these objects and provides the desired appearance and behavior of GUI items.

On the other hand, the customization of that appearance and behavior can be done by simple modifications of these objects. In other words, the configuration of the GUI is provided as an ordinary structure of objects of the classes from the OOIS UML library, interpreted by the GUI run-time component. The OOIS UML semantics of this object structure, as well as the capability to manage that object structure as any other, is one of the key features of OOIS UML that provides proper and complete coupling between the GUI and the "back-end" object space.

One GUI item configuration setting is associated with one type of element (for example, with a class or a property in the conceptual model). It then specifies the listed presentational parameters for all instances of that type (class), including its subtypes (specializations), unless there is another setting defined especially for the subtype. For example, a configuration setting for the class `Person` also is valid for objects of its specializations (for example, of the class `Teacher`).

GUI item configuration settings can take part in a relationship that is interpreted as their specialization or generalization. A configuration setting can be a *sub-setting* of another setting, meaning that it *inherits* all the parameters from the latter, but it can also *redefine* (that is, override) any of the parameters, specifying different appearance or behavior. For example, one setting defined for the class `Person` may specify all common presentational parameters for all persons, including teachers, but a specializing sub-setting for the derived class `Teacher` can redefine only the icon. Note, however, that this kind of relationship is materialized as an ordinary link between objects of `GUIItemSetting`, while the GUI run-time component simply interprets these links as just described.

For the stated requirements, within the context of the default browser, two GUI item configuration settings are needed, as shown in Figure 6-1:

- For objects of the class Person** — This setting should inherit all parameters from the default built-in setting for objects of any class (this default setting is an object that always exists in the object space by default), but redefine the icon and the description text so that it is taken from the `email` attribute of the object.
- For objects of the class Teacher** — This setting should inherit all parameters from the setting for objects of `Person`, thus inheriting the description text taken from `email` attribute, but redefine the icon for teachers.

Figure 6-1 shows the described configuration diagrammatically. The GUI item configuration settings are rendered as rectangles with upper corners cut off and divided into two sections. The upper

Part II: Overview of OOIS UML

section first reads the name of the setting. The name carries no semantics and is just for readability purposes, so that modelers can identify the settings within the entire GUI configuration. Following a colon is the name of the class whose objects this setting is defined for. The bottom section contains the specifications of the presentational parameters that are redefined. It is assumed that all other parameters of the setting are inherited from the super-setting. In the given example, the setting for Person redefines the icon and the description text to be taken from the email attribute.

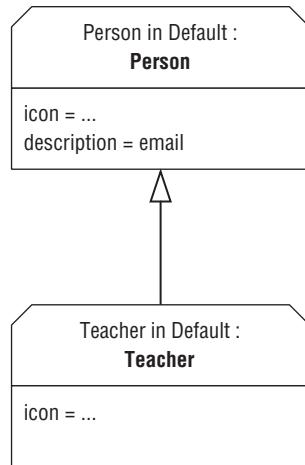


Figure 6-1: Two GUI item configuration settings related by the specialization relationship. The setting for the class Teacher specializes the setting for Person, thus inheriting all parameters (including the description text), but redefining the icon. Therefore, teachers will appear and behave the same as all other persons, except that they will be represented with different icons.

The generalization/specialization relationship between GUI item configuration settings is rendered as for classes. In Figure 6-1, the setting for Teacher specializes the setting for Person, thus inheriting all parameters (including the description text), but redefining the icon. Therefore, teachers will appear and behave the same as all other persons, except that they will be represented with different icons.

If a GUI configuration setting does not specialize any other setting explicitly, it will implicitly specialize a built-in default setting for objects of all classes, which defines the default appearance and behavior of all objects. This was the case with the GUI developed so far. For example, the default setting specifies that the “name” text of the object is taken from the attribute name of the object, if such exists. This is why the system was able to display the names of the persons and courses in the generic GUI, although this was nowhere specified in the model. Therefore, the setting for Person inherits all other default parameters except for those that it redefines.

Section Summary

- ❑ A *GUI item configuration setting* collects presentational and behavioral parameters defined for one particular kind of element in the system. It specifies how the elements of that kind appear and behave in the application's GUI.
- ❑ A GUI item configuration setting can be a *sub-setting* of another setting, meaning that it *inherits* all the parameters from the latter, but it can also *redefine* (that is, override) any of the parameters, specifying different appearance or behavior.

GUI Context

A GUI item configuration setting defines the presentational parameters for a single kind of element (for example, a class of objects). However, we need a whole set of item configuration settings that define the presentational aspects within one context of the GUI. The requirements of this section state that two other kinds of browsers are needed, and that the objects of the same class should appear differently in those browsers. For example, in the first kind of browser, courses should have their students as sub-nodes, whereas in the other, they will give their teachers. But in both, they have all other presentational aspects the same.

GUI context is a collection of GUI item configuration settings that define the presentational parameters for one part of the system's GUI. The part can be a browser window, a dialog, or a GUI widget within a dialog. In general, different parts of the system's GUI can represent contexts — a browser, a dialog, a set of Web pages, and so on. It is up to the GUI development framework, method, or execution environment to define what kind of things may represent contexts, and what kinds of contexts exist. In the generic OOIS UML run-time environment, a context may define the contents and appearance of one kind of browser. This means that each browser can be opened in a certain context, and many browser windows can be opened in the same context.

GUI contexts are again ordinary objects of the built-in class `GUIContext` from the model library, and can be accessed and manipulated as all other objects in the system.

GUI contexts can also be specialized and generalized. A GUI context can be a *sub-context* of another context, meaning that it *inherits* all the configuration settings from the latter, but it can also *redefine* (that is, override) any of the settings. In other words, you can define the settings within the topmost, default configuration context, and then develop other contexts with just slight redefinitions. For the given requirements, the context "Courses and Students" renders the courses with their students as sub-nodes in the tree view, whereas the context "Courses and Teachers" shows their teachers as sub-nodes; both reuse all other parameters from the default context.

Figure 6-2a shows part of the GUI configuration model. The context "Default" contains three GUI item configuration settings: "Course in Default" for the class `Course`, "Person in Default" for the class `Person`, and "Teacher in Default" for the class `Teacher`. The ownership between the context and its GUI item settings is denoted with a full line path with a filled circle at the context's end. The item setting for `Teacher` specifies the setting for `Person`, as already described.

Part II: Overview of OOIS UML

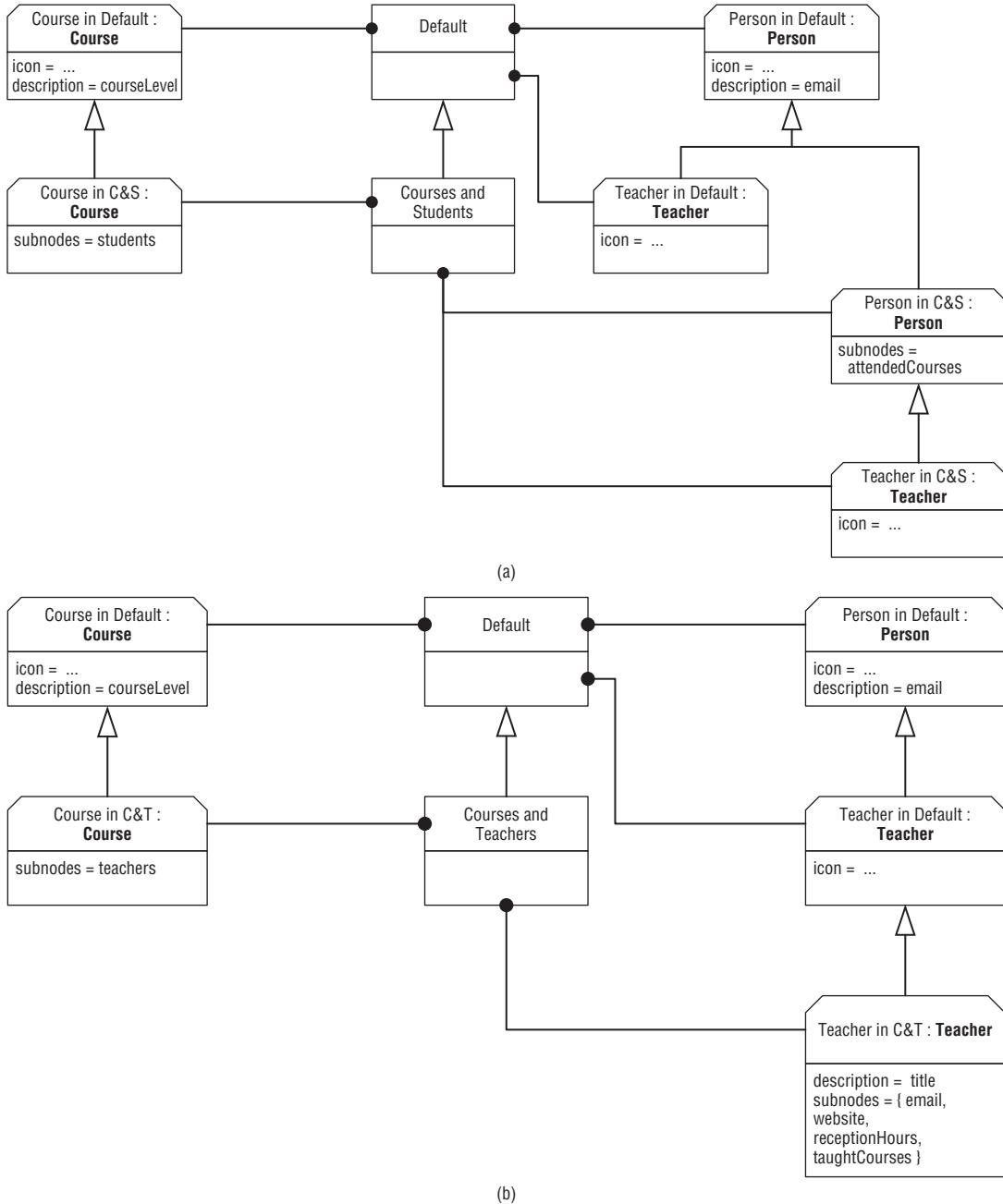


Figure 6-2: (a) GUI configuration for the default and the “Course and Students” contexts. The latter specifies the former, redefining the settings it owns. (b) GUI configuration for the default and the “Course and Teachers” contexts. The latter specifies the former, redefining the settings it owns and inheriting all others.

The model in Figure 6-2a has the following effect within the particular context named “Default.” Objects of the class Teacher appear and behave according to the setting “Teacher in Default,” because this setting fully matches the type of the object (the object is a direct instance of the class for which this setting is defined). If a presentational parameter is found in that setting, it is applied. For example, the icons for these objects are taken from this setting. If a presentational parameter is not found in this setting, its super-setting is searched. In this case, this is the setting “Person in Default.” Therefore, teachers appear and behave in the GUI as all other persons except for the icons.

The context “Courses and Students” in Figure 6-2a specifies the context “Default,” inheriting its entire GUI item configuration settings. This means that if a setting that exactly matches the type of an object exists in this context, that setting is applied. Otherwise, the super-context is searched for the matching setting, and so on. In this example, the context “Courses and Students” has the settings for all classes in the model, so this retrieval of the super-context will never take place. However, in general, this may happen if a sub-context does not need to redefine all settings. Within the sub-context “Courses and Students,” objects of the class Person will appear and behave according to the setting “Person in C&S” owned by this context. This setting redefines only how sub-nodes of the item are obtained — they are taken from the collection of objects in the property attendedCourses of the person. All other presentational parameters for objects of this class will be inherited from the super-setting “Person in Default.” The precise type matching and resolution rules, in general, will be defined later in Chapter 16.

Figure 6-2b shows the other part of the configuration model, for the context “Courses and Teachers.” Note that this context must redefine only the settings for Course and Teacher because only for these objects is a customization needed. If direct instances of Person occur in this context anyhow (for example, in some specification dialogs, within the collections of objects linked to a course), they will appear and behave as defined in the context “Default” because the sub-context “Courses and Teachers” inherits the corresponding setting for Person.

GUI context is just an abstract generalization of different specific kinds of concepts provided by the concrete GUI framework. In other words, the concrete GUI framework defines for which of its constitutional elements a context can be defined. For example, the generic OOIS UML execution environment that has been used in the examples so far provides a browser-oriented GUI as described. In it, one kind of browser represents a context. As a specialization, a browser may have other contextual features that can customize its appearance, such as:

- ❑ How the root nodes of the browser are obtained. For example, one type of browser can have one particular object as its root, or a set of objects specified in a certain way.
- ❑ What commands constitute its main toolbar.
- ❑ What classes from the conceptual model are available in its toolbar for creating objects.

Other features may be present, if appropriate.

Similarly, other GUI frameworks may define their specializations of context. As already stated, a context can represent a set of Web pages in a Web-oriented GUI framework, or a set of forms in a form-oriented GUI framework, and so on. All these specializations may define their own presentational parameters suitable for their appearance. What is the same, however, is that all of these kinds of contexts can inherit or redefine these parameters from their super-contexts.

Part II: Overview of OOIS UML

As already stated, the ultimate configuration of the GUI consists of objects of the predefined built-in classes `GUIItemSetting` and `GUIContext`, with their attributes set to define the appropriate appearance and behavior of the system. Their sub-setting relationships, of course, exist as simple, ordinary links between these objects in the object space. This allows significant flexibility of customization of the presentational parameters. However, if more complex logic is desired for defining the presentation (for example, dynamically or differently for each particular object), it can be achieved by developing subclasses of these built-in classes and overriding the corresponding methods.

Such an approach with implicit inheritance of properties of settings and contexts, but with the possibility of their redefinition, allows you to create compact configuration structures with lots of diversity in their presentation, yet with fewer redundancies and less development effort.

Section Summary

- ❑ A *GUI context* is a collection of GUI item configuration settings that define the presentational parameters for one part of the system's GUI.
- ❑ A GUI context can be a *sub-context* of another context, meaning that it *inherits* all the configuration settings from the latter, but it can also *redefine* (that is, *override*) any of the settings.

Interactive Manifestations

When the new system, customized with the described GUI configuration model, is executed, it demonstrates the following features (see Figure 6-3):

- ❑ The user can open a new browser over the main menu command File ⇔ Open. The contexts defined in the system are available in the sub-menu of that command.
- ❑ If the browser is opened in the context “Default,” the browser appears as before, except for the following:
 - ❑ The objects are represented with their class-specific icons.
 - ❑ Next to the objects of the class `Person`, each person’s e-mail address (if specified) appears as the description text in the list view.
 - ❑ Next to the objects of the class `Course`, the level of each course (if specified) appears as the description text in the list view.
- ❑ If the browser is opened in the context “Courses and Students,” the browser appears as in the context “Default,” except for the following:
 - ❑ The objects of the class `Course` have students as their sub-nodes in the tree view.

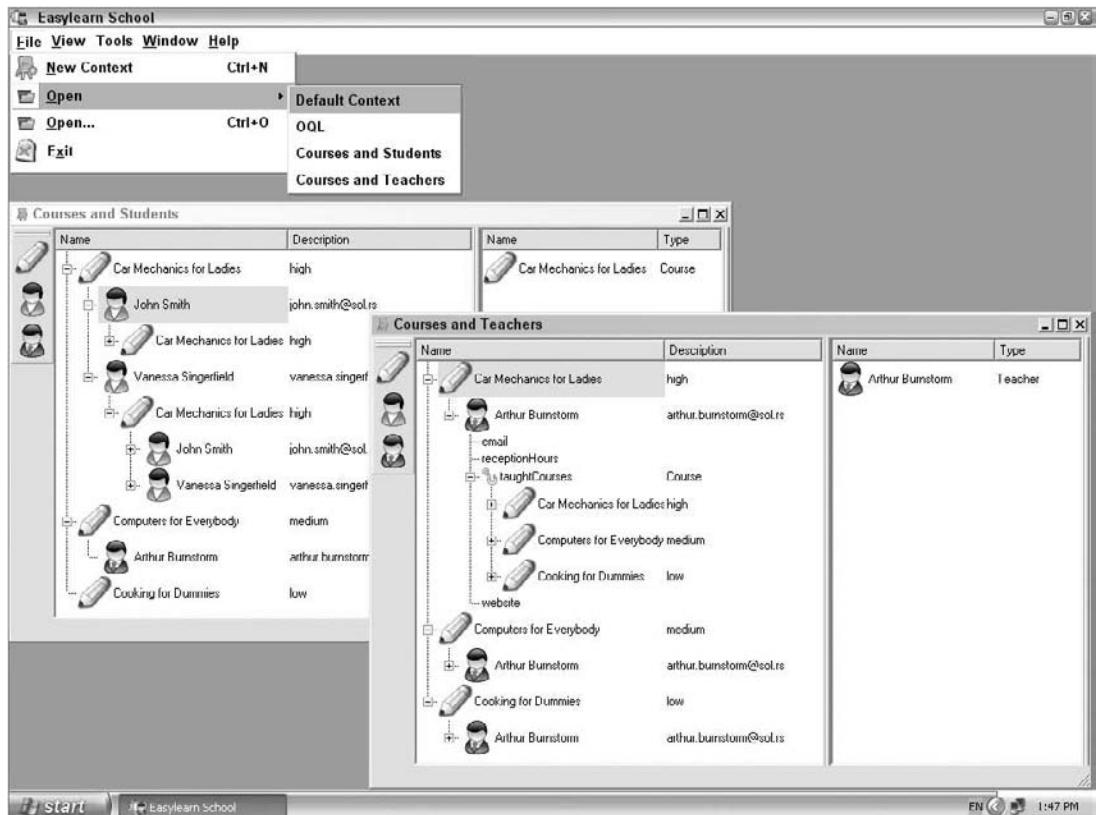


Figure 6-3: The newly customized contexts define different appearances and behaviors of browsers and their GUI items.

- The objects of the class `Person` have attended courses as their sub-nodes in the tree view.
- If the browser is opened in the context “Courses and Teachers,” the browser appears as in the context “Default,” except for the following:
 - The objects of the class `Course` have teachers as their sub-nodes in the tree view.
 - The objects of the class `Teacher` have these sub-nodes: email address, Web site address, reception hours, and taught courses.
 - Next to the objects of the class `Teacher`, the title (if specified) appears as the description text in the list view.

Note that the manifested customized appearance does not require any GUI-specific coding. This is again because the described GUI configuration model has formal semantics and carries enough information that can be interpreted by the GUI environment to have this manifestation.

Section Summary

- ❑ The GUI configuration has immediate effect in the GUI, according to the defined semantics of GUI item configuration settings, contexts, and their relationships.
- ❑ The manifested customized appearance and behavior does not require any additional GUI-specific coding.

FAQ

Q: Why doesn't the GUI configuration diagram use the standard UML class notation? For example, why isn't the ownership between a GUI context and a GUI item setting represented by the UML aggregation relationship?

A: Because GUI contexts and GUI item configuration settings are *not* classes, but *objects*. GUI contexts are objects of the built-in class `GUIContext`, and item settings are objects of the built-in class `GUIItemSetting`. Therefore, the relationships between these objects are *links* as instances of pre-defined built-in associations, not class relationships. For example, two associations are named `specialization`, one enabling links between objects of `GUIContext`, and the other enabling links between objects of `GUIItemSetting`. Similarly, the presentational parameters of the contexts and item settings (for example, the icons, the settings of how a description text is obtained) are mostly *attribute values* of these objects. The GUI configuration structure is thus an object structure, interpreted by the GUI environment in the defined way.

Consequently, the diagrams of GUI configuration structures are UML object diagrams by their nature. However, in order to be more comprehensible, instead of the standard UML notation for objects and links in object diagrams, GUI configuration diagrams use the specific notation. This is completely legal in UML. A standard modeling concept with a profiled meaning may have a specific notation.

Q: When can a GUI configuration be defined — at modeling or at run-time?

A: Being an object structure, a GUI configuration can be defined both at modeling and at run-time.

If it is defined at modeling time, it is defined in terms of object structures as parts of the model, as described in this chapter. Such object structures in the model have formal, *creational* semantics, instead of informal, *illustrative* purpose that UML object specifications have in general. A GUI configuration specifies what object structure should be created within the object space of the running system. These structures are just represented by the GUI configuration diagrams, but are defined by the corresponding model elements (that is, object and link specifications) depicted in those diagrams.

On the other hand, the same structure can be also defined at run-time. The GUI environment can provide a means for defining GUI configuration structures at run-time, interactively. The same structure can even be created and modified programmatically, during the execution of the system. This may allow the system to change its appearance dynamically, on demand, without any modeling efforts or need for recompilation. The generic OOIS UML run-time environment provides such a feature.

- Q:** Because the GUI configuration structure can be defined and even modified interactively, who exactly is supposed to do that — the developer or the user? If the user can do it, isn't it too dangerous that the user can modify the application?
- A:** Basically, the features of interactive definitions and modifications of the application's GUI is primarily dedicated to developers. It is unlikely to expect ordinary users to do that. This feature is primarily aimed at improving the system's flexibility and reducing the effort for developing and maintaining GUIs.

However, the same feature can be used by the end users if needed. For example, the underlying GUI configuration class model (the model that defines the built-in classes of GUI configuration elements) can be extended or merged with a conceptual model that deals with users of the system and their privileges. This way, the feature may be used (probably in a restricted form) for *personalizing* the GUI by users. For example, one group of users of the Easylearn school information system may be authorized to use the "Courses and Students" context and another group of users may be more interested in using the "Courses and Teachers" context. By managing a usual object structure as an instantiation of the underlying conceptual model for customizing the GUI, such an approach provides virtually unlimited flexibility.

It is true, however, that such a feature can be dangerous if available to inexperienced, unauthorized, or malicious users of the system. In that case, this feature should be prohibited. But it is simply a matter of disabling this feature, or restricting the rights of its use if necessary, which is much simpler than providing an advanced, easy, and flexible customization of the application. As already described, this feature is aimed at making the developers' life easier and providing better flexibility and maintenance of the application. But if it is not needed, it can be simply turned off. This is certainly less complicated than developing customized GUIs in the traditional way.

- Q:** What are the presentational features that can be specified in the GUI configuration?
- A:** The concrete set of presentational features supported by the GUI configuration structure is not defined by the OOIS UML profile. It is up to the implementation to define it. The described GUI configuration approach is simply a framework for defining coherent, simple, non-redundant, and flexible structures that specify the appearance and behavior of the application's GUI at one central place, instead of scattering such information across the application's code or model. However, in order to allow diversity of GUIs, the approach does not define the concrete set of features, but only some basic concepts. Every GUI development framework, method, or run-time environment can define its own set of concepts and their features, specifying the defined general concepts of GUI item setting and GUI context. These specializations thus form a model library supported by the given environment. The OOIS UML generic GUI environment provides one such support, as demonstrated in this chapter. You will read more on this topic in section Presentation in Chapter 16.
- Q:** What is the exact resolution policy for finding a GUI item setting for a certain type of element? Namely, if a certain context does not contain a GUI item setting for a certain class of objects, but for its superclass, and the super-context contains the setting for that very class, which one is taken?
- A:** The resolution policy will be discussed later in this book. Basically, if a certain context contains an exactly matching item setting (that is, the setting defined for the class the considered object is a direct instance of), this setting is taken. However, if the context does not contain such a setting, but contains a setting for its superclass, and the super-context contains an exactly matching setting, the priority is not defined. In other words, it is up to the implementation to select a priority. Therefore, there is no precise answer to the example stated in the question. It will depend on the implementation. Consequently, it is not wise to make such ambiguous configuration structures. Modelers should do their best to avoid such cases and to make clear and unambiguous

configuration structures. If such an example occurs in practice, it is wise to specialize the setting in the sub-context for the considered class to remove the ambiguity. Note that such an ambiguous case would exist in the Easylearn example, in the context “Course and Students,” if the setting for Teacher were removed (see Figure 6-2a). This is, actually, the reason why it has been redefined in this context. You will read more on this topic in section Presentation in Chapter 16.

- Q:** What about other kinds of user interfaces, such as Web? Namely, if I am not interested in a desktop-oriented GUI like the one described, but develop only Web applications, how can I use this?
- A:** As already discussed, the described concepts define the core technology for defining robust, coherent, and flexible GUI configurations. It is applicable to all kinds of GUIs, including Web. The given desktop-oriented GUI is just one illustrative example, but the approach is by no means constrained to it. It is perfectly applicable to Web GUIs, too. However, it is up to the Web GUI development framework, run-time environment, or method to specialize the given concepts and provide specific features suitable for Web GUIs.

Customizing Behavior

This section describes how the interactive (that is, behavioral) part of the GUI can be customized, again without any coding, and even with simple run-time configuration.

Requirements

The OOIS UML run-time environment with the GUI presented so far deals with the object space in a rather generic way by default. For example, in order to create a new student of a course, the user must perform two generic activities: the first to create a new object of the class Person or Teacher, and the second to link the new object by dropping it onto the slot students of the given course. Similarly, in order to assign a person to a course, the user must expand the node for the course object in order to see its students sub-node, and then drag and drop the person object onto that node.

However, this is not what the user intuitively expects. Moreover, the property students is not displayed in the customized context “Courses and Students” at all. What would be more convenient for the users is to have a sort of shortcut to the manipulations they do frequently, such as:

- 1.** Within the context “Courses and Students,” a person or a teacher object can be created and immediately linked to the selected course object as its student. Such a command may be (and is expected to be) offered in a right-click pop-up context menu of a course object, so that the user intuitively creates a student object of the selected course, as the user creates a subfolder of a folder in a GUI of a file system.
- 2.** In a similar way, within the context “Courses and Teachers,” a teacher object can be created and linked to a course object as its teacher (that is, immediately assigned to the given course on creation). Similarly, this command may also be offered in the right-click pop-up context menu for a course.
- 3.** Within the context “Courses and Students,” there should be a possibility to simply drag a person object and drop it onto a course object (not to its students slot because it is not displayed in this context’s tree view at all), which should result in assigning the person as a new student of the course. This is an intuitive user’s request to assign a student to a course within this context.

4. Quite similarly, within the context “Courses and Teachers,” there should be a possibility to simply drag a teacher object and drop it onto a course object (not to its teachers slot because it is not displayed in this context’s tree view at all), which should result in assigning the teacher to the course. This is an intuitive user’s request to assign a teacher to a course within this context.

Concepts

This section introduces the notion of a command that is the basic underpinning for the mechanism of customizing the application’s behavior, also described in this section.

Command

A *command* is a class whose objects represent requests for a service from the information system, issued interactively from the GUI. When a command is issued from the GUI, an instance of the command class is created and its `execute` operation is ultimately invoked. The behavior of the command is thus specified within the method of this operation. It can invoke an operation of an object, for example, or it can incorporate arbitrary code in the detail-level language.

A command can have its *input pins*. Input pins of a command are its parameters whose values are provided by the GUI dynamically, at the command’s execution. For example, a command that has two input pins can be activated by a drag-and-drop, when the GUI supplies the dropped and the target element to the adequate input pins of the command configured for that drag-and-drop.

Although a command is an ordinary class whose instances can be created programmatically from the application’s back-end behavior (for example, from the methods of domain-specific operations), instances of commands are generally targeted for direct activation from the GUI. In that case, when a command instance does not have a value on its input pin, the user may be asked to provide the value through the GUI interactively. In addition, commands have some attributes specially treated by the GUI in order to help the user identify the command and understand its meaning and purpose. These are the attributes `name` and `description`, which are of type `Text`, and which store a brief name of the command and a longer description of the purpose of the command, respectively. The GUI may display the values of these attributes in a generic and uniform way, regardless of the specific command.

A command is modeled as a class specifying (directly or indirectly) the built-in class `Command` available in the OOIS UML library. Input pins are modeled as properties (attributes or association ends) specially tagged with the symbol `<inputPin>`. The attributes `name` and `description` are features of the class `Command`, so they are inherited in the specializing class. In order to further emphasize that a class represents a command, especially when the specialization relationship is not shown in the diagram (but it must exist in the model), the class may be adorned with `<command>`, as shown in Figure 6-4.

For the second requirement specified at the beginning of this section (to provide a direct way to create a teacher of a course), a command could be defined in the model as shown in Figure 6-4. The command should have its features specified as follows:

- ❑ The name of the command class is `CmdCreateTeacherOfCourse`. This is the name of the class itself, as an element of the model.
- ❑ The name of the command that is used within the GUI of the application is “Create Teacher of Course.” The class `CmdCreateTeacherOfCourse` actually *redefines* the default value of the

Part II: Overview of OOIS UML

attribute `name` from the class `Command`, so that each object of the specializing class will have this particular value of the attribute on its creation.

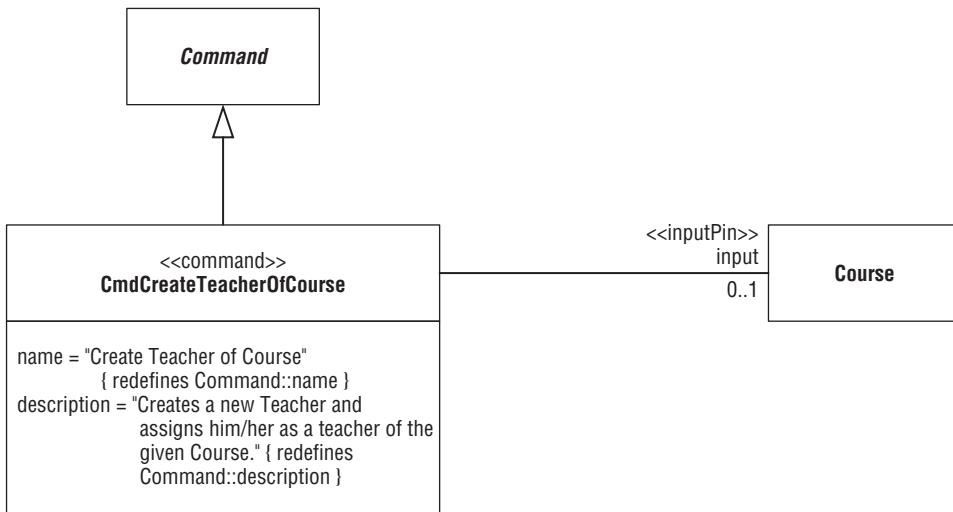


Figure 6-4: The command `CmdCreateTeacherOfCourse` that creates a new teacher object and assigns it to the given course. The course object is provided through the input pin named `input`.

- ❑ The description that is used within the GUI as a hint that helps the user to understand the meaning of the command is “Creates a new Teacher and assigns him/her as a teacher of the given Course.” The class `CmdCreateTeacherOfCourse` also redefines the default value of the attribute `description` from the class `Command`, so that each object of the specializing class will have this particular value of the attribute on its creation.
- ❑ The command has one input pin modeled as the association end `input` of the association with the class `Course`.

The redefined method for the inherited operation `execute` specifies the behavior when a command instance is activated. For the considered command, the method coded in Java or C# could be as follows:

```
Teacher t = new Teacher();
Course course = (Course)input.val();
course.teachers.add(t);
```

The first line of the code creates a new object of the class `Teacher`. A link of the association `teachers` is created in the third line, using the action `add`. The object of the class `Course` is already attached to the input pin `input` at that time because the run-time environment ensures that all input pins have their proper values when this method is activated. If a command is issued without values on all its input pins, the run-time environment may provide a way to ask the user for the values.

At run-time, a command can be applied to an object or to a collection of objects. If a command is applied to one object, and the command has any input pin that can accept that object, an instance of that command

is created, the reference to the object is supplied to the input pin, and the command instance is executed (by invoking its `execute` operation). If a collection of objects is provided instead, a separate command instance will be created for each object in the collection, and it will be applied to the object. This is a convenience for GUI, which can easily and generically support advanced features such as drag-and-drop of one object onto another object or a collection of objects, such as a result from a query or a multiple selection from a list.

Section Summary

- ❑ A *command* is a class whose objects represent requests for a service from the system, issued interactively from the GUI.
- ❑ When a command is issued from the GUI, an instance of the command class is created and its operation `execute` is ultimately invoked. The behavior of the command is specified within the method of this operation.
- ❑ A command can have its *input pins*, which are parameters whose values are provided by the GUI dynamically, at the time of command's activation.
- ❑ The elements used in the GUI to help the user understand the command are specified in its `name` and `description` attributes.
- ❑ A command can be applied to an object or to a collection of objects.

Categorization of Commands

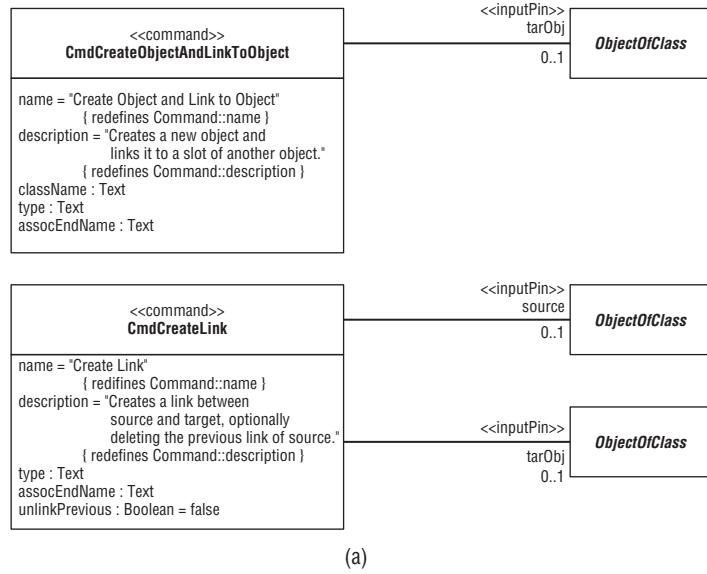
There are several *generic* (also referred to as *built-in*) commands supported by OOIS UML. They embody actions upon the object space that are applicable according to the semantics of OOIS UML, such as Create Object, Destroy Object, Create Link, and so on. For example, some generic commands are available by default in the generic OOIS UML GUI, either in the main toolbar, the toolbars of the browsers, or in the right-click pop-up context menus. Generic commands (as classes) are always implicitly defined in the model, because they are part of the OOIS UML model library. Therefore, they can be also instantiated programmatically or within a customized GUI.

You can fulfill the requirements posed at the beginning of this section by introducing specialized commands for each particular need, as shown previously in Figure 6-4. However, such manipulations with the object structure are pretty common in usual business applications and, thus, represent a kind of pattern. This is why the OOIS UML library of built-in commands includes the following two commands shown in Figure 6-5a:

- ❑ “**Create Object and Link to Object**” command — This command creates a new object of a certain class and links it to a certain slot of the given object. This command has the following properties:
 - ❑ The class of which a new object will be created. This is encoded as the name of the class in the attribute `className` of this command. (This is an ordinary textual attribute of this class, not an input pin.)

Part II: Overview of OOIS UML

- The association end of the slot to which the new object will be linked. This is encoded as the name of the association end in the attribute assocEndName of this command. (This is an ordinary textual attribute of this class, not an input pin.)



(a)

(b)

Figure 6-5: (a) Two generic (built-in) commands: “Create Object and Link To Object” and “Create Link.” (b) The configured prototypes of generic commands that support requirements given in this section.

- The object to whose slot the new object will be linked. The object is given in the input pin named tarObj.
- The class of the object at the input pin. This is encoded as the name of the class in the attribute type of this command. (This is an ordinary textual attribute of this class, not an input pin.)

The class `ObjectOfClass` is an abstract built-in OOIS UML class that is a direct or indirect generalization of all classes in the user's model.¹

- “**Create Link**” command — This command creates a new link between the two given objects, optionally deleting an already existing link. This command has the following properties:
 - The first (“source”) object that is going to be (re)linked. It is given in the input pin named `source`.
 - The second (“target”) object to which the first one is going to be linked. It is given in the input pin named `tarObj`. The names “source” and “target” of these two input pins reflect the usual roles of these two objects when this command is issued on a drag-and-drop.
 - The class of the object at the `tarObj` pin. This is encoded as the name of the class in the attribute `type` of this command. (This is an ordinary textual attribute of this class, not an input pin.)
 - The association end of the slot of the target object to which the `source` object will be linked. This is encoded as the name of the association end in the attribute `assocEndName` of this command. (This is an ordinary textual attribute of this class, not an input pin.)
 - The flag that indicates whether a possibly existing link of the same association in the `source` object is going to be deleted prior to linking the source to the target. There is a need to delete such a link if the multiplicity constraint at the association end given in `assocEndName` prohibits creating a new link between the source and the target. For example, this is the case when the upper bound of the multiplicity is 1. This flag is given in the Boolean attribute `unlinkPrevious` of the command (it is not an input pin, but an ordinary attribute).

These command classes allow for creating pre-configured objects that have their attributes set to the necessary values. The run-time environment can simply assign one object of a command class to a tool button or another control in the GUI, which serves as the *prototype* of the command that should be issued when the button is clicked. At that moment, the prototype object is *cloned*, by the means of invoking the polymorphic operation `clone` of `Command`. This operation creates and returns a clone (that is, a copy) of the prototype object. The clone has all attributes set to the same values as the prototype. The clone is then executed by the environment (by means of invoking its `execute` operation). This way, the run-time environment can issue the commands that are even dynamically configured and assigned to the tool buttons and menu items, without knowing their concrete types.

This is actually a direct application of the Prototype design pattern [Gamma, 1995]. As a result, instead of extending the model with a new command class for each of the requirements, the developer simply configures the prototype objects of the ready-to-use generic commands, which can be done even at run-time, without any modeling effort, as you will soon see.

¹A tool supporting OOIS UML may simply introduce a specialization relationship from any class in the model to the class `ObjectOfClass` if the former is not already specialized from any other class. That way, any class in the model will be always directly or indirectly specialized from `ObjectOfClass`.

Part II: Overview of OOIS UML

Figure 6-5b shows how five prototype command objects should be configured in order to support the requirements posed at the beginning of this section. The objects in the first row support the requirements 1 and 2, which deal with creating new objects and linking them to a course (as a student, the first two objects — or as a teacher, the third one). The objects in the second row support requirements 3 and 4, and deal with linking an object to a course (as a student or a teacher).

Apart from the generic built-in commands, the system can have *domain-specific commands*. Such commands are designed by the modeler and can incorporate arbitrary programming code in their methods for `execute`, possibly issuing OOIS UML actions. Such commands are services to users accessible through the user interface in the same way as built-in commands. They represent entry points to the implementation of the specific functionality of the system, not directly supported by the execution environment and generic commands. By the mechanism of prototype objects, these commands can be configured to be available in the toolbars, right-click pop-up context menus, or in other places in the GUI of the system. The previous discussion gave an example of such a command in Figure 6-4.

Section Summary

- ❑ *Generic* (or *built-in*) commands are commands that embody certain UML actions upon the object space. They are part of the OOIS UML model library.
- ❑ *Domain-specific* commands represent entry points to the implementation of the specific functionality of the system, not directly supported by the execution environment and generic commands. They are introduced into the model by the modeler.

Customizing Behavior by Modeling

The customized behavior described in the “Requirements” section can be obtained by extending the GUI configuration model shown previously in Figure 6-2. Figure 6-6 shows a piece of that model extension, which customizes the behavior for the third requirement in the “Requirements” section (dragging and dropping a person object onto a course object to assign the person as a student of the course).

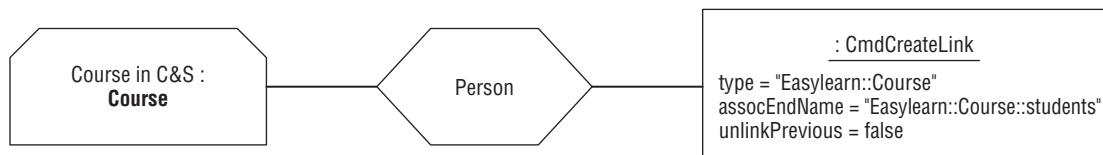


Figure 6-6: A part of the model for customized behavior. A two-way command mapping, rendered as a hexagon, is assigned to the GUI item configuration setting for the class `Course` in the context “Courses and Students.” The mapping is defined for the objects of type `Person`. When such an object is dropped onto an object of the class `Course`, the associated command prototype is cloned and executed.

Within the context “Courses and Students,” a command that is issued when an object of the class `Person` is dropped onto an object of the class `Course` should be defined. For that purpose, a special GUI configuration element, *two-way command mapping*, is assigned to the GUI item configuration setting for the

class `Course`. The two-way command mapping defines which command should be executed when an object of another type is dropped onto an object of the type for which the GUI item setting is defined. The command is defined over a prototype command object, which was already specified in Figure 6-5b. The command that is specified must have at least two input pins that accept objects of the corresponding types. (The substitution rule is always assumed.)

In Figure 6-6, the two-way command mapping is rendered as a hexagon. It is defined for a certain type of the dropped object (`Person`, in this case). When a source object of `Person` is dragged and dropped onto the target object of `Course`, within the context “Courses and Students,” the corresponding GUI item configuration setting is searched for the two-way command mapping that matches with the type of the dropped object. The type matching and resolution rules are similar to those for the GUI item settings. When such a two-way mapping is found (either in this GUI item setting, or its super-setting), the associated command prototype object is cloned. The dragged object is supplied to the clone’s input pin named `source`, and the object onto which that one is dropped is supplied to the clone’s input pin named `tarObj`. The operation `execute` of the clone command object is ultimately invoked.

It should be noted that the diagram in Figure 6-6 is again an object diagram with formal creational semantics. It defines which objects should be created at run-time to provide the desired customization of behavior. The two-way mapping hexagon also represents an object of the corresponding built-in class from the OOIS UML model library for GUI configuration.

The commands available in the right-click pop-up context menu for objects of a certain class are defined in a similar way — through a collection of command prototype objects linked to a GUI item configuration setting.

Section Summary

- ❑ The behavior of the GUI can be customized by extending the GUI configuration model. The model is an object structure specification with creational semantics. It defines which objects and links in the GUI configuration structure should be created at run-time, in order to provide the desired behavior of the system. The objects in this structure are mostly instances of built-in OOIS UML library classes.
- ❑ *Two-way command mapping* is a class whose object can be assigned to a GUI item configuration setting in order to define the command that is issued when an object of a certain type is dropped onto an object of the type for which the setting is defined.

Customizing Behavior by Demonstration

The presented approach of customizing behavior by modeling (that is, by developing specifications at design time) has some drawbacks. Namely, in order to fulfill a requirement such as those at the beginning of this section, developers must go through the same old, well-known development procedure.

Part II: Overview of OOIS UML

Generally, when designing a software system, developers must first understand the requirements and try to imagine the proper behavior of the system that would meet a particular requirement. To do this, developers usually simulate or envision the system's structure and behavior in their minds, often using some concrete examples from the problem domain to make the visions more clear. Then, developers use their mental skills to abstract the foreseen structure or behavior in order to produce more general and formal specifications of the system (for this particular example, of GUI configuration, like that one presented in the previous section). From such specifications, the target system can be obtained either by transformations into lower-level implementation forms, or directly interpreted by a run-time environment. Developers must then verify the resulting system by executing test cases, whereby the system exhibits its behavior on concrete testing examples. Developers compare the exhibited behavior with the one initially envisioned or sketched. If the former behavior does not correspond to the latter one, developers go back to the specification to correct it, and the process is repeated.

The first part of this process (mapping of the envisioned samples of structure or behavior into the formal generalized specifications) is the crucial one in the entire process because the rest of the process can be automated or strongly supported by tools. It is mentally intensive and tough. Additionally, it is error-prone, because developers usually resort to poorly structured and non-deterministic mental processes. Thus, developers may easily overlook a specific detail since the chosen examples are not manifested in a concrete, visible, and tangible way. Finally, it is often time-consuming and may need recompilation of the system.

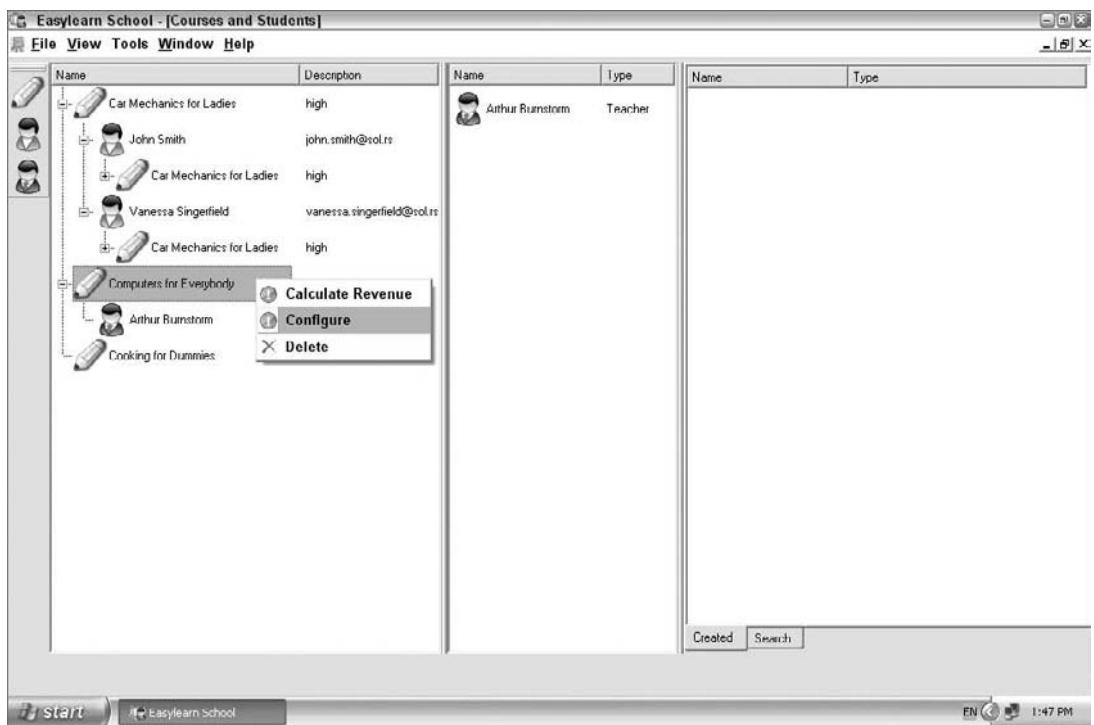
Instead of such an approach, where developers first *map* their imagination or sketched particular demonstrations into formal specifications (models) that *instruct* the computer how to handle general cases, it may sometimes be more convenient to do it more directly. Developers may *demonstrate* the desired behavior *directly* to the computer, as a teacher does to a pupil, using *concrete examples*, and the computer may *reproduce* the demonstrated behavior on other cases during the exploitation of the constructed system. This metaphor has been known for a long time as *programming by demonstration (PBD)* or *programming by example (PBE)*.

The requirements given at the beginning of this section can be fulfilled completely by demonstration, at run-time, without any modeling efforts and needs for recompilation. This is because the GUI configuration structure described in the previous subsection, which is needed for the desired behavior, is an ordinary object structure that can be constructed at run-time. The execution environment can handle the user's actions upon the concrete objects in the system, which serve as examples, and suggest the commands that should be configured on those actions, or even allow developers to demonstrate the behavior of new commands. For the given requirements, this may look as follows.

The first requirement in the "Requirements" section (adding a command that creates a new person object as a student of the selected course) can be fulfilled through the following process whose steps are shown in Figure 6-7:

- ❑ The developer right-clicks on an object of the class `Course` in a "Courses and Students" browser. By default, the right-click pop-up context menu does not contain a command that can create a new student of the course. Therefore, the developer selects the command `Configure`, which is available by default in the menu (see Figure 6-7a).
- ❑ The run-time environment searches for the GUI item configuration setting that determines the appearance and behavior for the clicked object. If there is no exactly matching setting in the cur-

rent context, the developer is offered to create one, to configure the setting from a super-setting, or to configure a setting for a super-type.

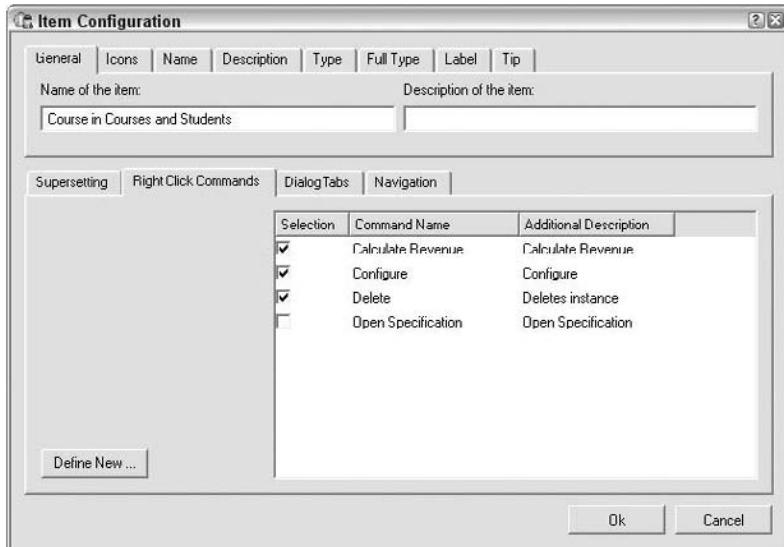


(a)

Figure 6-7: The configuration process for a one-pin creational command in the right-click pop-up context menu. (a) The default right-click pop-up menu of a course does not contain the required command. The user selects “Configure.”

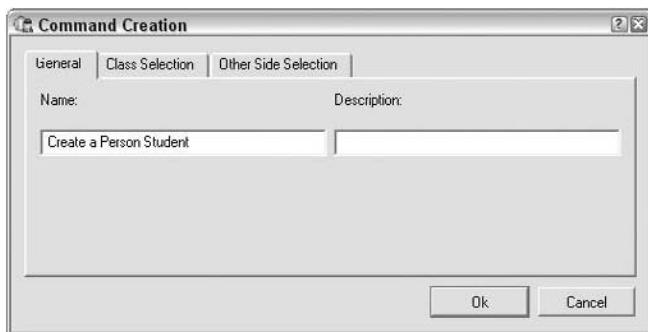
- ❑ The configuration dialog for the selected GUI item configuration setting is opened. The dialog's Right-click Commands tab contains the list of the commands available in the pop-up menu. This list can be inherited from the super-setting, or redefined (see Figure 6-7b). The list offers all commands (actually, all prototype command objects in the system's object space) that can be applied to an object of the current type (Course, in this case). These are commands that have at least one pin that accepts that type (substitution is always assumed). This list does not contain the required command for creating a student of the course.
- ❑ Therefore, the user clicks the button Define New. This button leads to a dialog for configuring a new command prototype of the built-in one-pin creational command CmdCreateObjectAndLinkToObject. The dialog first allows the developer to enter the name and description of the command prototype (that is, the values of its name and description attributes, Figure 6-7c). The developer can enter the name “Create a Person as a Student” and an appropriate description.

Part II: Overview of OOIS UML



(b)

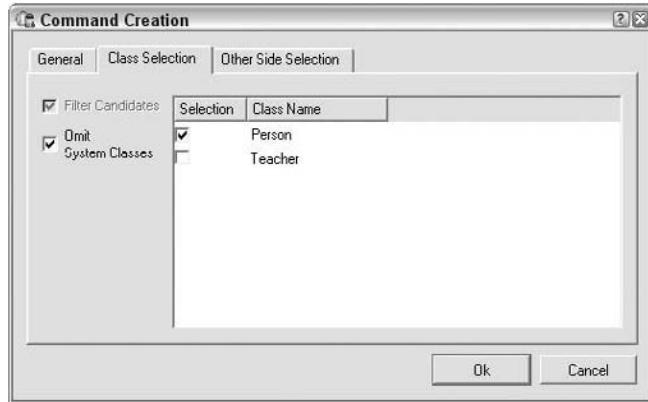
Figure 6-7: (b) The GUI item setting dialog's tab that defines the list of commands available in the right-click pop-up menu. The list can be inherited from the super-setting or redefined.



(c)

Figure 6-7: (c) The dialog for configuring a new prototype of one-pin creational command first offers to enter the name and description of the new command prototype.

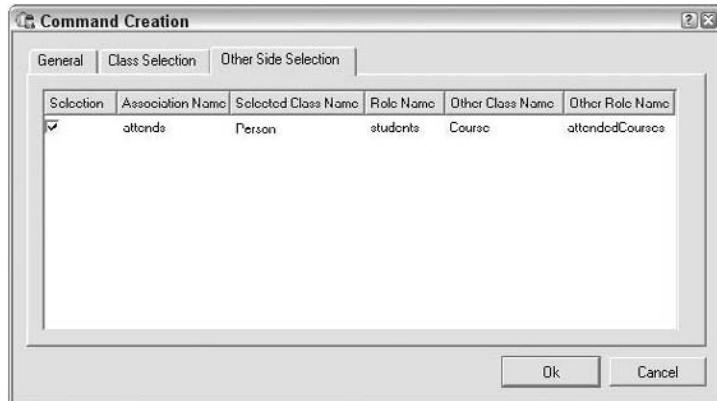
- The dialog also allows the developer to select the class of which the object will be created by the command (see Figure 6-7d). Note that the list of offered classes contains only those non-abstract classes that have at least one association toward the class of the object being configured (Course, in this case), including the inherited associations. The candidates are the classes Person and Teacher, in this case. (If the class Person were abstract, for example, it would not be a candidate here.) The user selects one of them, Person in this case.



(d)

Figure 6-7: (d) The next option is to select one of the classes that have an association towards the class Course.

- ❑ The dialog then offers the slot of the object being configured to which the new object will be linked (see Figure 6-7e). The list of candidate slots includes all the slots that can accept a link with the new object (whose class is selected in the previous step).

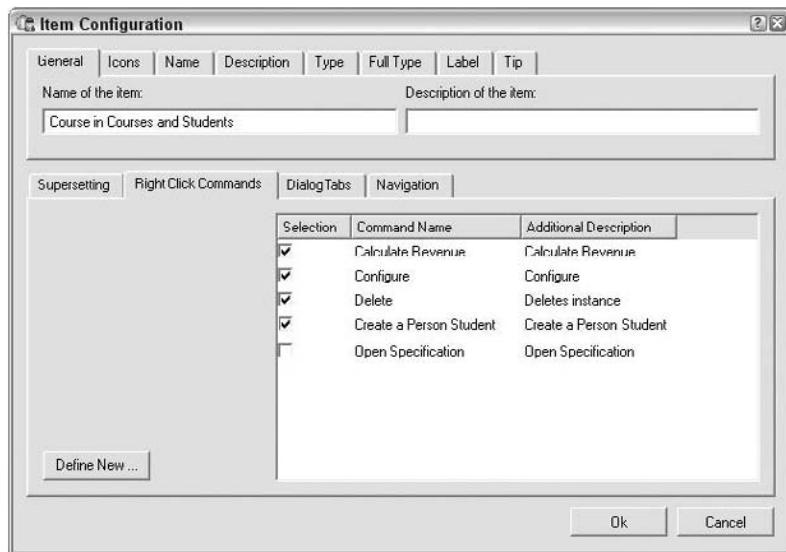


(e)

Figure 6-7: (e) Finally, the association end to which the new object will be linked is selected.

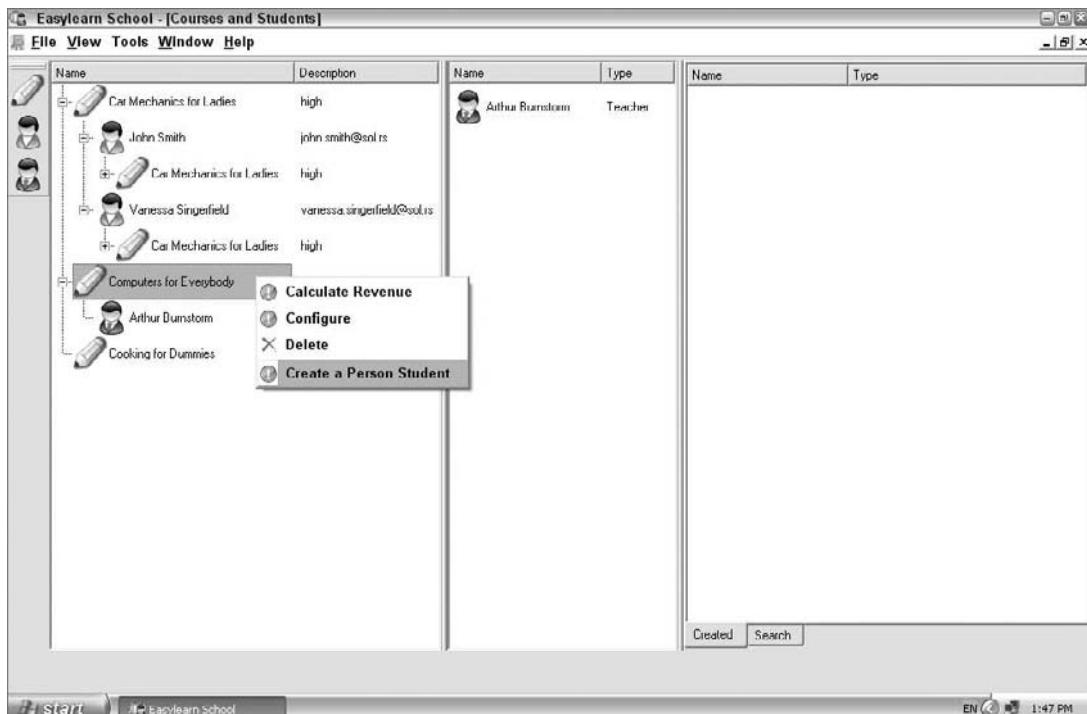
- ❑ Now, the configuration of the new command prototype is finished and the developer closes the dialog and returns to the GUI item configuration dialog (see Figure 6-7f). The new command prototype appears in the list of candidates for the right-click pop-up menu because it satisfies the condition to have at least one pin that accepts a course. The developer selects it, among the others.
- ❑ After the configuration dialog is closed, the new command “Create a Person as a Student” appears in the right-click pop-up context menu for objects of the class Course (see Figure 6-7g). When this command is issued for a course, a new object of the class Person is created and immediately linked as a student of the selected course.

Part II: Overview of OOIS UML



(f)

Figure 6-7: (f) Now, the GUI item configuration dialog offers the new command prototype.



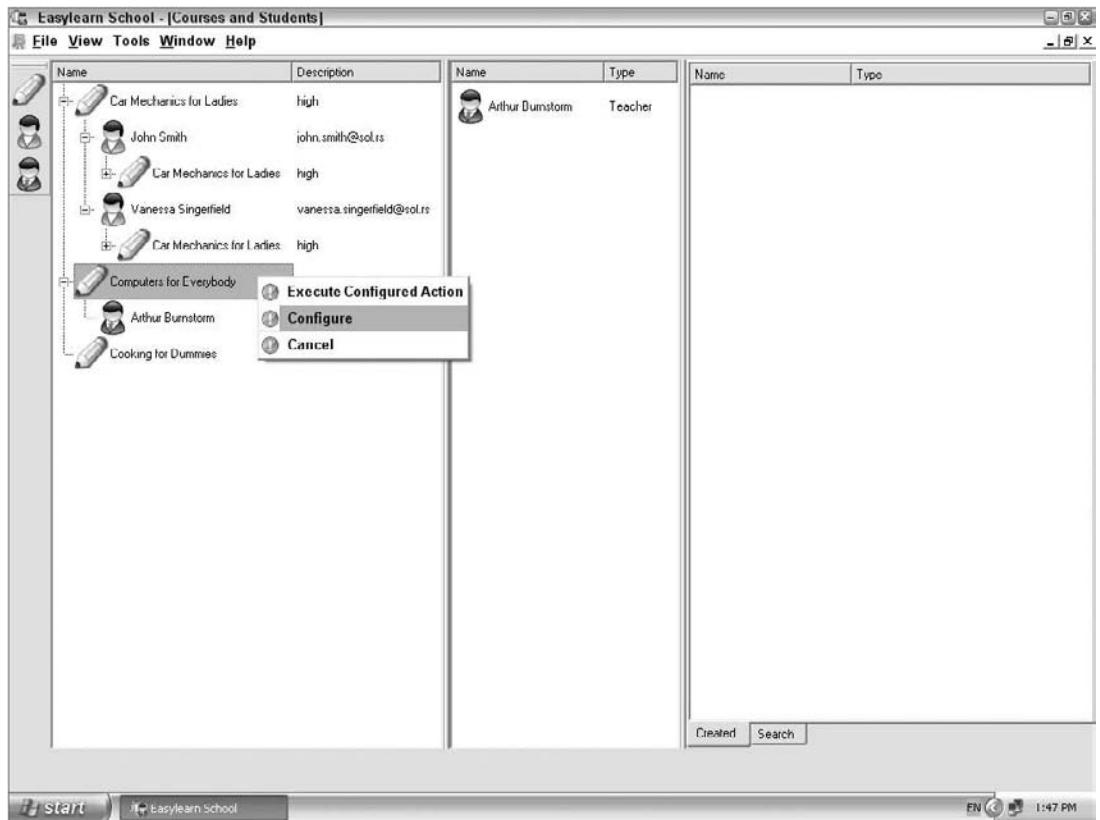
(g)

Figure 6-7: (g) After selecting it and closing the configuration dialog, the new command appears in the right-click pop-up menu of a course.

The same procedure can be accomplished to configure a command prototype “Create a Teacher as a Student” given in the requirement 1, and a command prototype “Create a New Teacher” for the requirement 2.

The rest of the requirements (3 and 4) require configuration of two-pin (re)linking commands for drag-and-drop user actions. The configuration is again accomplished by a simple procedure as follows (see Figure 6-8):

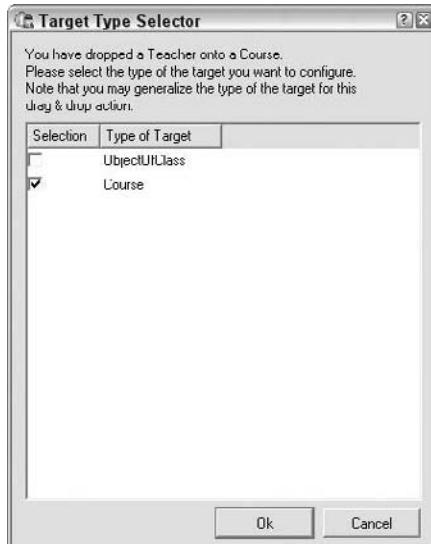
- ❑ By default, in the context “Courses and Students,” a left-mouse-button drag-and-drop of a person object onto a course object does not have a configured command and is, thus, not applicable. This is why the developer drags a person object using the right mouse button, drops it onto a course object, and selects Configure (refer to Figure 6-8a).



(a)

Figure 6-8: The configuration process for a two-pin (re)linking command. (a) The user drags a person object using the right mouse button, drops it onto a course object, and selects Configure.

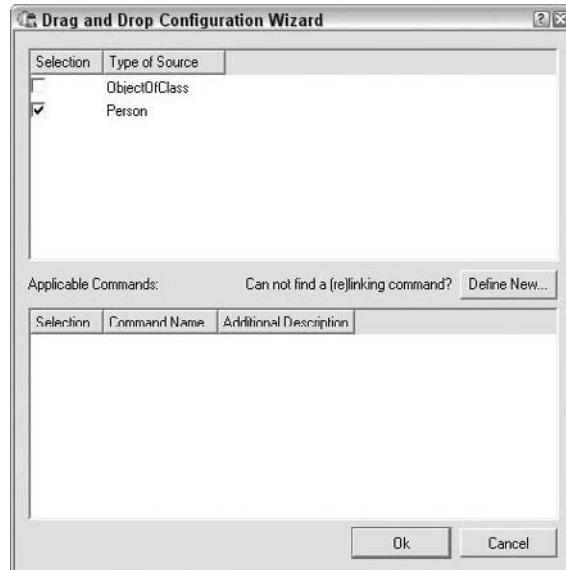
- ❑ The dialog that pops up first allows the developer to generalize the type of the target object (refer to Figure 6-8b). This may be necessary if the demonstrational example includes a target object of a concrete specialization, and the configuration should be generalized for a (possibly abstract) base class. In this case, this is not necessary, because the target object is a course.



(b)

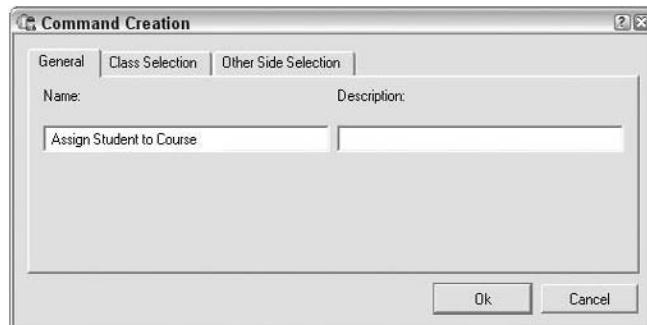
Figure 6-8: (b) The dialog that pops up allows the user to generalize the type of the target object.

- Then, the two-way command mapping configuration dialog is opened (see Figure 6-8c). It lists the types for the source object. This again is an opportunity to generalize the type for the source (dragged) object, if the demonstration were done using an object of a specializing class. In this example, if the demonstration were done using a teacher as the dragged object, the class Person should be selected here because the same behavior is needed for all persons, not only teachers.
- When the type of the source object is selected, the list of candidate command prototypes is shown in the list underneath (see Figure 6-8c). The candidates are all prototype objects of commands that have at least two pins that can accept the selected source and target objects (substitution is always assumed). The list does not have the needed command “Assign Student to Course.” For this reason, the developer clicks the Define New button.
- The dialog for configuring a new two-pin (re)linking command prototype of the class CmdCreateLink looks similar to the one for one-pin creational commands (see Figure 6-8d). First, the name and description of the command prototype can be specified. In this case, the name “Assign Student to Course” can be entered.
- Because the classes of the source and target objects that will be supplied on the input pins have been determined already at this point, there is only the possibility to select the slot (as a manifestation of an association end) of the target object to which the source object will be linked (see Figure 6-8e). The list offers all associations between those classes, including the inherited ones.
- Now the new command prototype is configured, and the dialog is closed. The new prototype “Assign Student to Course” appears in the list of candidates in the two-way command mapping configuration dialog. It should be selected. When the dialog is closed, the configuration is finished.
- If a person object is now dropped onto a course object, it is linked as a student of that course.



(c)

Figure 6-8: (c) The two-way command mapping configuration dialog allows the user to generalize the type of the source object. The list of candidate prototype command objects offers all those that can accept the source and target objects on their input pins.

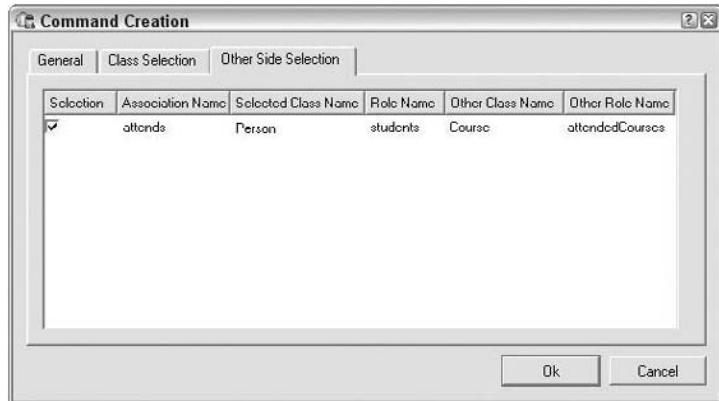


(d)

Figure 6-8: (d) The dialog for configuring a new prototype of the two-pin (re)linking command first asks the user to enter the name and description of the new command prototype.

This procedure has fulfilled the third requirement in the “Requirements” section. The same procedure should be accomplished for the requirement 4 — linking a teacher to a course in the context “Courses and Teachers.”

Note that if a behavior configuration (either of a one-pin or a two-pin command) is specified in a super-context, it is implicitly inherited in its sub-contexts, unless redefined there.



(e)

Figure 6-8: (e) Now, the two-way command mapping configuration dialog offers the new command prototype. After selecting it and closing the configuration dialog, the new command will be executed on a drag-and-drop.

The presented scenarios are very rudimentary examples of the idea of programming by demonstration. These scenarios basically do some simple configuration of the underlying object structure at run-time. Some more advanced examples of the same approach will be described later in this book, where it will be possible to specify the methods of operations by demonstrating the scenarios of actions that must be performed by the methods. These scenarios can be demonstrated interactively, on the actual (or demonstrational) object space, at run-time. For example, the described configuration procedures can allow the user to define a new command at the specific point, and define its behavior by demonstrating the scenario of managing actual objects.

Section Summary

- The object structure that customizes behavior can be created at run-time, dynamically and interactively, by *demonstration* of the user actions on concrete *examples* in the real object space, and configuring the command prototypes. The system helps in this process by offering possible candidates from the conceptual model.

Interactive Manifestations

Regardless of how a command prototype has been configured for a specific action, in the model or by demonstration, it can be accessible in the GUI in one of several ways:

- **Commands with or without input pins can appear in toolbars of the browsers.** The toolbars of the browsers can be configured as all other properties of this kind of context. If a command

has no input pins, it is applicable always. If a command has at least one input pin, it is applicable only if there is a currently selected object of the type acceptable by the command's pin. The command is disabled otherwise.

- ❑ **Commands with one input pin can appear in the right-click pop-up context menus of objects.** This is the case for the commands "Create a Person as a Student," "Create a Teacher as a Student," and "Create a New Teacher," as described (the first and second requirements in the "Requirements" section).
- ❑ **Commands with two input pins can be issued on drag-and-drop user actions.** This is the case for the commands that assign a person as a student of a course or a teacher of a course (the third and fourth requirements in the "Requirements" section).

All commands, built-in or domain-specific, are treated in the same way in the GUI. They differ only in the origin, not in the manifestation. Commands are classes, and have the manifestation within the GUI as all other classes. Command prototypes are objects as all other objects and can be manipulated as usual. Consequently, the presentational aspects of command prototypes are obtained as for all other objects — through the GUI configuration structure. The built-in GUI configuration structure provides some default icons for built-in commands, and through the mechanism of inheritance of configuration, they are used for all other domain-specific commands until they are redefined. Similarly, the names of the commands are taken from their `name` attribute by default. This is how the generic OOIS UML GUI renders the commands in the toolbars and menus.

Section Summary

- ❑ The command prototypes configured either in the model or by demonstration, may have the following manifestation in the GUI:
 - ❑ Commands with or without input pins can appear in toolbars of the browsers.
 - ❑ Commands with one input pin can appear in right-click pop-up context menus of objects.
 - ❑ Commands with two input pins can be issued on drag-and-drop user actions.

FAQ

Q: Actions and commands seem to be somewhat similar. What are the similarities and the differences between the two concepts?

A: Actions and commands have some similarities, because they both deal with behavior. More precisely, their similarities are as follows:

- ❑ Both represent requests for some service and thus deal with behavior.
- ❑ Both can be executed.

Part II: Overview of OOIS UML

- Both may have input pins that hold their input parameters.

However, actions and commands have lots of differences:

- Actions represent elementary manipulations upon pieces of the object space. Commands represent requests for services from the entire system issued by users.
- Actions are elements of the UML as a language. Commands are just ordinary classes.
- They exist at completely different levels of detail. Commands exist in the system's structural model. Actions exist within behavioral features of classes (that is, in their methods).
- Actions are used within methods. Commands execute methods.
- Actions can be connected in data or control flow structures within methods. Commands are intended for individual activation from the GUI.
- When an executed action does not have a value on one of its input pins, this is an error caused by an incorrectly defined method, and an exception can be raised. When an executed command does not have the value on one of its input pins, the value can be provided interactively through the GUI, or the command can simply be ignored.
- The identity of an action is not relevant. Commands have their names and descriptions that can be displayed to the user for convenience.
- Commands can be applied to an object or to a collection of objects. Actions cannot; they rely on their input pins alone.

Q: Why was the concept of command needed at all? Why wasn't the concept of operation sufficient? Especially for one-pin commands, like the one that creates a student of a course, couldn't the same be done using an operation of Course that creates a new student of the course?

A: It is true that the given example could have been supported by an operation of the class Course that creates a new person or a teacher object, and links it to the course. However, the concept of commands as defined here was necessary for conceptual and pragmatic reasons. Following are the conceptual ones:

- In general, a command represents a user's request for a service from the entire system, not a request for a service of a particular object issued from another object. It is an entry point to a required functionality of the system. A request for a service from the entire system and from a particular object significantly different viewpoints.
- A command can operate on one object (meaning that it has one pin), but also on several objects or no objects at all. If an operation is to manipulate multiple objects of different classes, it may be difficult sometimes to determine which of the classes should own the operation.
- A command can encapsulate a more complex business-level processing or interaction that involves many participating objects, and for which one operation is too fine-grained.

The pragmatic reasons are many and probably more important:

- By the virtue of being classes, commands can be instantiated. Objects of classes can survive their execution, so they can be stored as a log of a user's actions.

- The mechanism of authorization (access control) can be easily constructed as a model that relies on commands as classes.
- The manipulation with commands in the GUI can be implemented generically because commands are usual classes with defined prototypes (the Prototype design pattern).
- The customization of behavior by demonstration is possible because of commands and their prototypes.
- Instead of implementing a new operation and its method for each and every case like the one in the given example, it is easier just to configure command prototypes as ordinary objects of classes. And such cases of simple structural manipulations that can be supported by built-in commands are very common in business applications.

Actually, the operations of objects that are available in the generic OOIS UML GUI by default are commands that wrap the operation invocations. This is because the GUI can deal only with commands in a generic way, not directly with operations.

- Q:** What is the relationship between the concept of command and other similar concepts from the literature, such as control objects from OOSE [Jacobson, 1992], business objects in three-tier architectures, and the Command design pattern [Gamma, 1995]?
- A:** The relationship is very close. The concept of command in OOIS UML is a direct application of the Command design pattern. It also corresponds directly to the concept of control object in OOSE, and to the middle tier in three-tier architectures, which embodies the business logic. However, the concept of commands specializes these concepts, makes them more concrete, and extends them with the described features.
- Q:** Why isn't the behavior described in the "Requirements" section generically supported by the OOIS UML run-time environment, given that it is already possible?
- A:** Because it could be a too sophisticated behavior, very particular for a specific problem domain. Moreover, it would not be wise to do so for the following reasons:
- There might be many ways to interpret a drag-and-drop of objects of a single class, and the framework cannot always select the desired one. For example, there cannot be a default interpretation of a drag-and-drop of a teacher onto a course because a teacher can be both a student and a teacher of the course.
 - Such behavior is not always needed in the application and varies from case to case.
 - The behavior like this might depend on the context (that is, on the perspective offered by a particular part of the GUI). Note how the same drag-and-drop action of a teacher onto a course has different meaning in two different contexts.
- Q:** The behavior can be customized interactively, but who exactly is supposed to do that — the developer or the user? If the user can do it, isn't it too dangerous that the user can modify the application?
- A:** The answer to this question is exactly the same as for customization of presentation.
- Q:** Why do both methods of customizing presentation and behavior, by model and by demonstration, exist at all? The latter seems to be more efficient and cool, so why is the former necessary?
- A:** It is true that, in many cases, customization by demonstration, interactively and at run-time, is easier and more efficient than by modeling. However, modeling has some advantages in

Part II: Overview of OOIS UML

the case of more complex configuration structures, with lots of generalization/specialization relationships:

- The diagrammatic representation of the configuration model is much more clear and comprehensible. Models and diagrams in those cases can be irreplaceable for understanding and maintenance of the entire configuration.
- The development of complex configurations can require an iterative process of development and corrections, so the modeling approach is more appropriate because it is done “off line.”
- In some implementations, in case of modifications of the conceptual model of the system (for example, of some classes for which the GUI configuration is associated), the automatic upgrade of the configuration object structure may be problematic if the configuration is developed dynamically. Instead, an easier approach can be to regenerate the entire configuration structure from the up-to-date model.

Querying

This section describes the querying feature in OOIS UML.

Requirements

Users of the Easylearn information system want to pose some non-trivial queries upon the system’s object space. For example, they want to retrieve all the teachers who are students of the course “Computers for Everybody,” or to retrieve all outsourced courses attended by the teacher John Smith.

Moreover, users want to have queries stored in the system’s persistent storage, so that they can be re-executed many times, and even modified, if necessary, during the exploitation of the system. In other words, users want to manipulate queries just like all other objects in the system.

Concepts

This discussion examines the Object Query Language (OQL), as well as several ways to specify queries in OOIS UML.

The Object Query Language

For the purpose of querying the system’s object space, OOIS UML uses the *Object Query Language (OQL)*. OQL is an object-oriented descendant of SQL, originally defined and managed by the Object Data Management Group (ODMG).² The OOIS UML profile uses a dialect of OQL. The dialect follows the idea and basic syntax of ODMG’s OQL, but also defines some extensions and variations. OOIS UML does not support queries that modify the object space, because modification of the object space is done through actions compliant with the OOIS UML semantics.

²ODMG (www.odmg.org) disbanded in 2001. In 2005, the OMG’s (Object Management Group, www.omg.org) Object Database Technology Working Group (OMG ODTWG) was formed as the successor to the ODMG.

SQL is a standard query language that has a very broad popularity in the database community. (Interested readers unfamiliar with SQL are encouraged to read Chapter 22 available in the Supplement.) It is supported by all major commercial RDBMSs. Additionally, its syntax and semantics are rich enough to support complex and powerful queries on the database. However, SQL is not suited for the object paradigm at all. Although the object space can be mapped to a relational database schema, the relational schema of the database should not be directly visible to users and developers. Instead, users and developers should work with the pure object conceptual basis, as it has been described so far. In other words, they should deal with objects of classes, attribute values, and links of associations, instead of tables, records, and fields. Therefore, the queries that could be written in SQL to query the relational persistent storage of an object information system would be too complicated and clumsy, as will be demonstrated here. This is why OOIS UML uses OQL.

Generally, OQL supports most features of SQL, especially operators in expressions. In fact, OQL queries can be translated into SQL queries prior to their execution by the RDBMS if the object space is stored in a relational database. This is why switching from SQL to OQL is very easy.

Section Summary

- ❑ OQL is an object-oriented descendant of SQL, initially defined and managed by the Object Data Management Group (ODMG).
- ❑ OOIS UML uses a dialect of OQL restricted exclusively to the retrieval queries, but with some extensions.

OQL in OOIS UML

OQL queries that retrieve information from the object space are SELECT queries, just as in SQL. However, the main difference between SQL and OQL SELECT queries is that the former navigate over tables, records, and fields, while the latter navigate directly over objects and their slots. Namely, in order to fetch the records that are conceptually related with other records, a SQL query must explicitly specify their join using the equality of some fields (primary and foreign keys). This way, the developer or the user must make some effort to retrieve conceptually related entities. OQL queries overcome this drawback.

For example, in the Easylearn information system, whose final conceptual model is shown again in Figure 6-9, the following OQL query retrieves the names and e-mail addresses of all persons that attend the course entitled “Computers for Everybody”:

```
SELECT p.name, p.email
  FROM Course c, c.students p
 WHERE c.name = 'Computers for Everybody'
```

Part II: Overview of OOIS UML

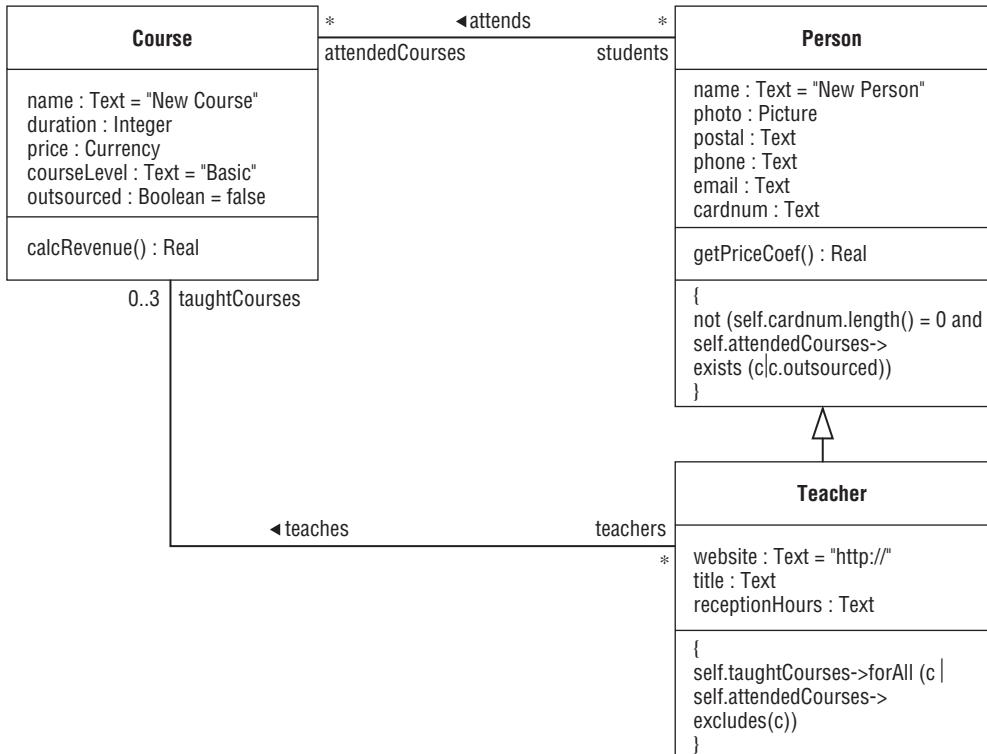


Figure 6-9: The Easylearn final conceptual model

For comparison, the equivalent SQL query must navigate over more tables and join their records explicitly:

```

SELECT p.name, p.email
  FROM Course c, Attends a, Person p
 WHERE c.ID=a.IDCourse AND a.IDPerson=p.ID
       AND c.name = 'Computers for Everybody'
  
```

Even though this is a simple query, the difference is more than obvious. For more complex queries, the equivalent SQL queries are even more cluttered and less readable than OQL queries.

Furthermore, OQL supports generalization/specialization relationships directly. For example, if only teachers who attend the mentioned course need to be selected, the query would be as follows:

```

SELECT p.name, p.email
  FROM Course c, c.students:Teacher p
 WHERE c.name = 'Computers for Everybody'
  
```

This query specializes (by the specifier :Teacher) that only those students of the course that are of type Teacher are wanted, and not all persons.

The support of generalization/specialization relationship includes, of course, inheritance. For example, the following OQL query retrieves the names of outsourced courses attended by the teacher John Smith:

```
SELECT c.name
  FROM Teacher t, t.attendedCourses c
 WHERE t.name = 'John Smith' AND c.outsourced = true
```

Note that the inherited property can be fetched from an object of the derived class, as is usual in the object paradigm. Therefore, `t.attendedCourses` is accessible in the object `t` of the class `Teacher`.

Generally, as in SQL, the result of an OQL query may be represented with a table (record set) having some columns (fields) and rows (tuples, records), as shown in Figure 6-10. Specifically, the result of an OQL query in OOIS UML is a collection of tuples, whereby each value of a tuple can be one of the following:

- An object specified with its alias defined in the `FROM` clause
- An object's slot (that is, an instance of an attribute or an association end)

Each of these elements appears and behaves according to its GUI item configuration setting in the surrounding context. For example, objects are displayed as icons that react on mouse events and drag-and-drop of other objects. Slots of association ends are displayed as icons that can be expanded into collections of linked objects. Slots of attributes are displayed using the corresponding GUI widgets, and so on.

The queries mentioned so far return attribute values (that is, simple tabular data). For example, the following query retrieves the names (attribute values) of the courses attended by the teacher John Smith that are provided by an outside source:

```
SELECT c.name
  FROM Teacher t, t.attendedCourses c
 WHERE t.name = 'John Smith' AND c.outsourced = true
```

Sometimes, the intention of the user may be to obtain just a collection of objects by a query. In that case, the resulting collection of objects allows commands to be applied to some or all of them. For example, the following query returns the courses (as objects) attended by the teacher John Smith that are provided by an outside source:

```
SELECT c
  FROM Teacher t, t.attendedCourses c
 WHERE t.name = 'John Smith' AND c.outsourced = true
```

Part II: Overview of OOIS UML

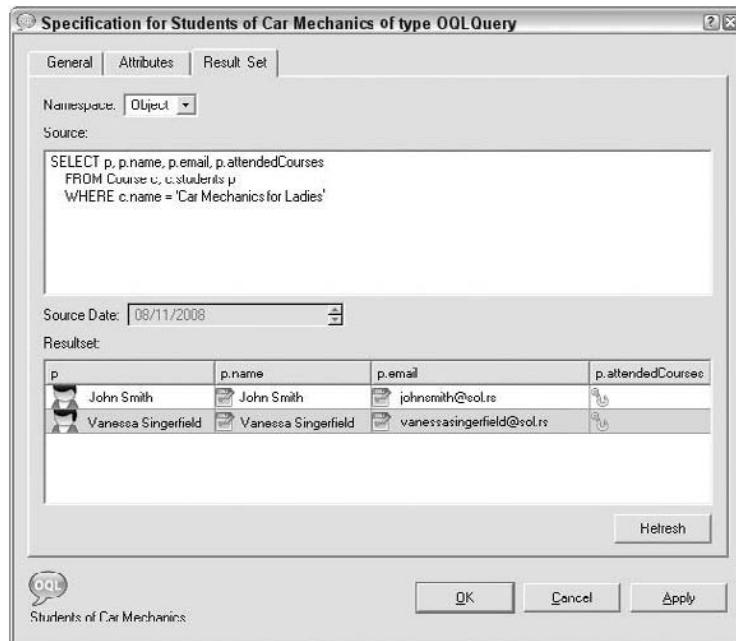


Figure 6-10: The result of a query with combined columns: SELECT p, p.name, p.email, p.attendedCourses FROM Course c, c.students p WHERE . . .

If a query returns objects, the returned objects are instances of the corresponding class, and may actually be objects of any derived class (according to the substitution rule). For example, the following query returns the persons that attend the course “Cooking for Dummies,” but the persons can also be teachers. Each object will behave as it is specified in its class.

```
SELECT p
FROM Course c, c.students p
WHERE c.name = 'Cooking for Dummies'
```

Finally, this next query returns the person (as object), its name and email attribute values, and the slot attendedCourses (as a node that could be expanded into a collection of linked objects):

```
SELECT p, p.name, p.email, p.attendedCourses
FROM Course c, c.students p
WHERE c.name = 'Cooking for Dummies'
```

Section Summary

- ❑ OQL queries have a similar syntax as SQL queries, but they navigate over the object space (that is, objects and properties), and not over the relational record space.

- OQL supports generalization/specialization relationships directly, so that inherited properties can be referred to in navigation, too.
- OQL supports type specialization.
- The result of a query is a collection of tuples, whereby each tuple may contain objects themselves, along with their slots.

Queries as Pattern Object Structures

So far, queries have been specified using the OQL textual notation. However, a typical approach a developer or a user takes when designing a query (especially its navigational `FROM` clause) is to consult the UML class diagram of the conceptual model, navigate through the classes and relationships, envision the desired object structure that is searched for, and form the `FROM` clause on the fly. In other words, the designer of the query envisions a *pattern* of the object structure that should be searched for in the entire object space. The designed OQL query (in particular, its `FROM` clause) is just a formal textual notation for such a pattern. Consequently, it is often more convenient to use a diagrammatic notation instead of the textual one to specify a query.

For that purpose, OOIS UML allows the use of models of object structures that have formal *pattern matching* semantics and represent the navigational part of the query. Such an object structure can be rendered in UML object diagrams that show which objects and links form that pattern structure. For example, the following OQL query, which selects the teachers that attend the course “Car Mechanics for Ladies” and all courses other than that attended by those teachers, is visually specified in Figure 6-11:

```
SELECT t.name AS name, t.email AS email, c2.name AS attnCourse
  FROM Course c1, c1.students:Teacher t, t.attendedCourses c2
 WHERE c1.name = 'Car Mechanics for Ladies' AND c1<>c2
```

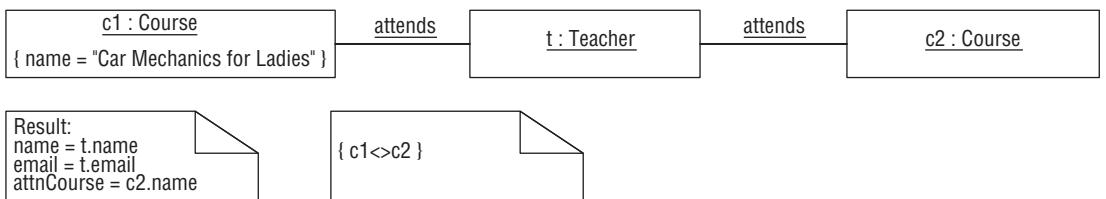


Figure 6-11: An example of a query specified by a pattern object structure. The query selects the teachers that attend the specified course and all courses other than that attended by those teachers.

As you can see in Figure 6-11, several notational styles are used to specify the three parts of the query.

The navigational part of the query (that is, the one that corresponds to the `FROM` clause of OQL) is defined in terms of the object structure. The object structure defines a pattern that should be matched in the object space when the query is executed. The pattern-matching algorithm returns all combinations of objects in the object space that are linked as defined by the pattern. In other words, it returns all sub-graphs of the object space graph that are isomorphic with the given pattern. Each such matching sub-graph will form one candidate tuple of the query result.

Part II: Overview of OOIS UML

The conditional part of the query (that is, the one that corresponds to the WHERE clause of OQL) is specified using logical condition specified in the same way as constraints. A condition may be defined within the context of one object, thus referring to its properties directly, or in the context of the entire query, where it can relate properties of all participating objects. The matching algorithm returns only those candidate tuples that satisfy (the conjunction of) all these conditions.

Finally, the selection part of the query (that is, the one that corresponds to the SELECT clause of OQL) is specified through the *output parameters* of the query. Each output parameter has its name, with the same purpose as the alias in OQL, and is assigned the value of an object's slot or an object itself. This is specified in the "Result" note in the diagram.

Section Summary

- A query can be specified using an object structure, possibly depicted in object diagrams. Such an object structure defines the *pattern* that should be searched for in the entire object space.
- The object structure has formal *pattern matching* semantics and represents the navigational part of the query, corresponding to the FROM clause in OQL.
- The conditional part of the query, corresponding to the WHERE clause in OQL, can be specified by the conditions either in the context of each object, or in the context of the entire pattern object structure specification.
- The selection part of the query, corresponding to the SELECT clause in OQL, is specified by the *output parameters* of the query.

Queries as Objects

Queries can be created as objects of the built-in class `Query` from the OOIS UML model library. This means that queries can be created (defined) and deleted by the user, at run-time, dynamically and interactively. They are stored in the system's object space as all other objects. In other words, *query* is an abstraction in OOIS UML that specifies some retrieval of information from the underlying object space. It has some attributes and some specific operations that can be applied to it. A query has a name, which helps the user to identify it, and a description, which helps the user to understand what it does. A query is specified by its *textual source*, written in OQL, or its *pattern object structure*. Note that designing pattern object structures for queries can be easily done by demonstration (that is, by creating the sample pattern structure as any other structure of objects).

Queries can be manipulated (that is, created, deleted, edited, and executed) as all other objects, from the GUI of the application (for example, from the generic OOIS UML GUI), or from within methods.

The generic OOIS UML GUI allows all generic operations on queries (for example, create and delete). Additionally, a query can be edited and executed, and its results can be displayed. The textual or diagrammatic source of a query is edited using a text or diagram editor. When a query is executed, its result is displayed in a table. Such use will be described in the next section.

When accessed from a method, the OOIS UML API can be used to create and execute the query. (The same holds for commands and constraints as objects, because they also can be manipulated from methods.) The result of the query can then be iterated or a command can be applied to it. For example, the following code snippet in Java applies the generic command Destroy Object to the result of the query, and thus deletes all objects returned by the specified OQL query:

```
OQLQuery query = new OQLQuery();
query.setOQLSource("SELECT t FROM Course c, c.students:Teacher t
    WHERE c.name = 'Computers for Everybody'");
QueryResult rs = query.execute();
Command cmd = new CmdDestroyObject();
cmd.applyTo(rs.getColumn("t"));
```

Section Summary

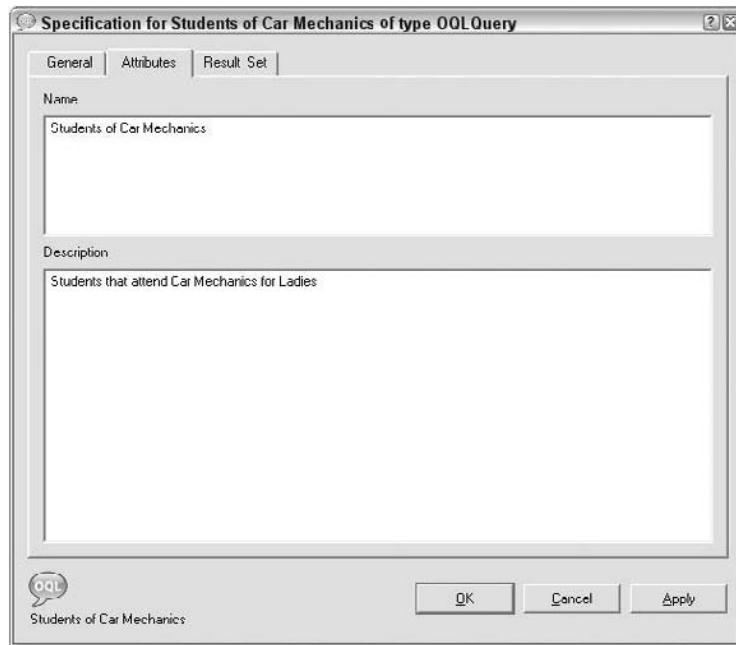
- ❑ A query is an abstraction in OOIS UML that specifies some retrieval of information from the object space. Queries are objects of the built-in class `Query`.
- ❑ A query has a name, which helps the user to identify it, and a description, which helps the user to understand what it does.
- ❑ A query can be specified by its *textual source*, written in OQL, or by a *pattern object structure*.
- ❑ Queries can be manipulated (that is, created, deleted, edited, and executed) as all other objects, from the GUI of the application, or from within methods.

Interactive Manifestations

As already explained, the run-time environment allows you to create a query just like any other object. The values of the attributes `name` and `description` can be set afterward.

The source of the query can be edited (see Figure 6-12a). Once the source is defined, the query can be executed. The result is then available in the corresponding table (see Figure 6-12b). The elements of the result appear and behave as anywhere else, and may be manipulated by the usual operations in the execution environment, such as drag-and-drop or double-click.

Part II: Overview of OOIS UML



(a)

Figure 6-12: (a) The specification dialog for a query allows editing of the textual OQL source of the query.

A screenshot of the same "Specification for Students of Car Mechanics of type OQLQuery" dialog, but now showing the results of the executed query. The "Source" field contains the OQL query:

```
SELECT p, p.name, p.email, p.attendedCourses
FROM Course c, c.students p
WHERE c.name = 'Car Mechanics for Ladies'
```

The "Resultset" section displays a table with the following data:

P	p.name	p.email	p.attendedCourses
	John Smith	johnsmith@solrs	
	Vanessa Singerfield	vanessasingerfield@solrs	

(b)

Figure 6-12: (b) The result of the executed query

Section Summary

- Queries can be manipulated as ordinary objects (created, modified, and deleted).
- The source of the query can be edited.
- When a query is executed, its result is presented in a table.

FAQ

- Q:** Who is supposed to design queries as objects — the user or the developer? Is it reasonable to expect an ordinary user to learn OQL?
- A:** It is unlikely that most ordinary users of an information system will be ready to learn OQL and write OQL queries. But this feature is not primarily dedicated to users. It is aimed at making developers' lives easier and the system more flexible. Namely, if users need a new query, developers can simply add the query as an ordinary object to the object space, without the need for re-compilation or re-installation of the system. The query object can even be sent to users by e-mail or downloaded from the Web and imported to the system. Users do not need to see the source of the query at all — they are simply interested in what the query does, not how it does it. This is why they may treat the query as any other object — they can simply double-click on it to execute it and see the result, and nothing more. However, there still might be more advanced users who are familiar with OQL, such as the administrators of the system or those who maintain it. This feature may be valuable for them. Anyhow, if the option of creating queries is not needed by users, it can be simply removed from the system or restricted to some categories of users.
- Q:** Is there any other way of specifying queries, or are OQL and pattern object structures the only available ones?
- A:** OOIS UML does not constrain the means by which a query is specified. OQL and pattern object structures are two predefined ways. The concrete implementation can provide any other method that fits in the entire framework.
- Q:** Are these two means of query specification equivalent? Why do they both exist? Which one is preferable?
- A:** Both means, OQL and pattern object structures, have the same *expressive power*, meaning that if a query can be specified using one of them, it can be specified in the other, and vice versa. However, they do not have equal *expressiveness*, meaning that the equivalent specifications in the two methods may differ in their complexity. In general, pattern object structures are more expressive, because for some queries, the equivalent specifications in OQL may require more effort to design and may be less readable. However, for simple queries, OQL may be more convenient because of its compact textual notation. Moreover, OQL may be more convenient for those familiar with SQL. This is why the choice of the method is up to the designer, and may depend on a concrete query or on the designer's personal preferences. You will read more on this topic in section Pattern Object Structures in Chapter 12.
- Q:** Do these methods support some more advanced features, such as parameterization, sub-querying (nesting of queries), and recursion?
- A:** Yes, they both do. These features will be described later in Chapter 12. However, a concrete implementation may opt not to support some of these advanced features, such as sub-querying or recursion.

Part III

Concepts

Chapter 7: General Concepts

Chapter 8: Classes and Data Types

Chapter 9: Attributes

Chapter 10: Associations

Chapter 11: Constraints

Chapter 12: Querying

Chapter 13: Operations and Methods

Chapter 14: State Machines

Chapter 15: Collaborations and Interactions

Chapter 16: Commands, Presentation, and Architecture

7

General Concepts

This chapter describes some general principles and concepts that form the foundation of UML and of the OOIS UML profile. They include a few concepts that gather the common characteristics of many other concepts in the language, such as named elements, namespaces, and multiplicity elements, as well as some concrete concepts, such as comments, packages, dependency relationships, and variables.

The Dichotomies of OOIS UML

There are several important dichotomies underpinning the OOIS UML profile. They divide the profile into somewhat opposite, but often complementary, parts.

Specification/Realizations and Classifier/Instances Dichotomies

Many modeling concepts of UML represent *classifiers of instances* — that is, specifications of groups of instances that share some common characteristics and features. For example, classes are classifiers of objects, associations are classifiers of links, and so on.

A *classifier* is one of the fundamental concepts of UML. A classifier in the model defines a classification of *instances* into sets, according to their *features*. Features are defined within a classifier. For example, attributes and operations are features of classes and represent the shared characteristics of all instances of the class.

The classifier-instance relationship is just one form of a more general relationship between a model element that represents a *specification* (or “descriptor”) and the things that exist or happen at run-time as its *realizations* (that is, its instances or activations). For example, a class is a specification of its objects, while a method is a specification of its particular executions (activations, invocations) at run-time. Just as a blueprint specifies how different houses are to be constructed, a class specifies its objects, or a method its executions.

Part III: Concepts

In summary, the specification-realizations and classifier-instances dichotomies exist in the foundations of UML. This approach is actually inherited from earlier programming paradigms that had also identified these dichotomies. For example, procedural languages clearly distinguish between data types and their instances, or procedure definitions and their invocations.

Section Summary

- Several UML modeling concepts represent sets of *instances* and specify their *features*. These are called *classifiers*.

Modeling and Execution

The model of the system is the specification of the system's structure and behavior that is exhibited at run-time, when the system is being executed. Therefore, the elements of the model form a conceptual space and are created and manipulated at design time. They specify templates and rules for the manipulation of the object space at run-time. The elements of the object space represent instances of some elements of the model. In other words, the elements of the object space are concrete, tangible incarnations of some model elements. Consequently, the conceptual space of the model and the design time of the system form one side of another important dichotomy. At the other side is the object space that exists at execution time.

In the parlance of OMG, the object, run-time space is referred as to the *M0 level*, whereas the conceptual space of the model is referred to as the *M1 level*. These are just two particular levels of a more general hierarchy of modeling levels in which each higher level represents the specification of its adjacent lower "instance" level — that is, two adjacent levels are related with model-instance relationship. That way, the M2 level is the meta-modeling level that defines the concepts of the modeling language (that is, represents the "model of the modeling language").

It should be noted that this separation is only conceptual. First, as shown already, the OOIS UML execution environment is aware of the conceptual model of the system (for example, its classes, attributes, associations, and so on). Moreover, the profile does not explicitly prevent modifications of the model during the execution of the system. The possibility to modify the model during the execution of the system is up to the implementation of the profile. The notions of the "conceptual space/design time" and "object space/run-time" are introduced to logically separate two levels of abstraction, often related by the classifier-instance relationship. Physically, modeling (that is, modifications of some elements of the conceptual space) and execution (for example, modifications of some elements of the object space) may take place at the same time. For example, an implementation may allow the modeler to work on one part of the model, while users operate upon the object space of another part of the model. Eventually, both logical spaces actually reside in the same physical space — the computer memory.

Section Summary

- ❑ At the “modeling” side of the dichotomy is the conceptual space, which comprises model elements that exist at design time. Many of model elements are classifiers.
- ❑ At the “execution” side of the dichotomy is the object space, which exists at run-time. Its elements represent instances of some model elements.

Compilation and Interpretation

Execution frameworks of programming languages generally take two approaches to the execution of a program. The first approach assumes *compilation* of the program. Compilation is a transformation of the program or model, written in a more abstract textual or visual language, into a less abstract, implementation form. The implementation form uses lower-level concepts, and is often not aware of the initial high-level representation of the program. For example, during execution of a compiled C++ program, objects are represented as simple records of data members in memory, and operations are implemented as ordinary low-level functions, coded in the machine language, which simply accept their arguments over the machine control stack. The run-time environment of a C++ program is not aware of the classes, inheritance, and other concepts that exist in the source program¹.

The advantage of this approach is efficiency because the execution of the low-level implementation form is generally more efficient than interpretation of a higher-level representation. The disadvantage of this approach is that it is less flexible because it does not allow dynamic interpretation of the actions that are issued for the source-level concepts at run-time. For example, a C++ object of a class cannot be created unless its class is statically specified in the source code of the creation action.

The other approach is *interpretation* based on *reflection*. It is said that a modeling or programming language is *reflective* if its execution environment is aware of (at least some) elements from the source representation. Consequently, the program can get the information about its own structure at run-time. This approach also allows dynamic interpretations of some actions in the program. For example, Java, as a reflective language, allows creation of an object by specifying the class dynamically, at run-time. A Java program is, therefore, aware of the classes and their members, although Java code is compiled into an intermediary code, called *byte code*, which is interpreted by the Java execution environment (the Java Virtual Machine, or JVM). The advantage of this approach is flexibility, as described. The disadvantage of this approach is reduced efficiency in comparison with execution of compiled systems.

OOIS UML is a highly reflective language, meaning that the program can get most of the information about its source form (that is, the model) at run-time. Therefore, the execution environment can dynamically interpret the actions upon the object space, issued interactively. This is the basic

¹C++ supports a rudimentary form of reflection that is used for dynamic type casting, which cannot be treated as a full-fledged reflection mechanism.

Part III: Concepts

explanation of the features provided by the generic OOIS UML GUI and run-time environment described in earlier chapters. For example, the GUI can create an object of a class or a link of association specified dynamically, commands and GUI settings can be dynamically configured to work with certain classes or properties, and so on. In fact, all these dynamic features are possible because of reflection. For the same reason, the API of OOIS UML allows access to the information about the model from within the detail-level code of methods. More precisely, the reflectivity of OOIS UML includes the access to only those modeling concepts that have direct effect on the object space at run-time (that is, that have formal run-time semantics).

However, OOIS UML does not exclude compilation. It allows both approaches to be used in a combined, complementary manner. If a traditional programming language is used as the host language, the implementation of the model can be generated in that language and then compiled. The profile itself does not specify the way the system is implemented and to what extent compilation or interpretation is used. It just guarantees the reflection.

It should be emphasized once more that compilation does not necessarily exclude reflection. Compiled systems can still be reflective, as already explained. On the other hand, dynamic interpretation assumes at least some kind of reflection. A concrete implementation of OOIS UML can be based on compilation, but must ensure reflection. Compilation limits one important capability of reflective systems, however: the capability of dynamic self modification. However, this is not expected from OOIS UML implementations.

Section Summary

- ❑ Compilation is an automatic transformation of the source program or model, written in a more abstract textual or visual language, into a less abstract, implementation form. The implementation form uses lower-level concepts and is often not aware of the initial high-level representation of the program.
- ❑ A modeling or programming language is *reflective* if its execution environment is aware of (at least some) elements from the source representation. Consequently, the program can get the information about its own structure at run-time.
- ❑ The advantage of compiled systems is efficiency because the execution of the low-level implementation form is generally more efficient than interpretation of a higher-level representation. The advantage of interpretation based on reflectivity is that it is more flexible because it allows dynamic interpretation of the actions that are issued for the source-level concepts at run-time.
- ❑ OOIS UML is a highly reflective language, meaning that the program can get most of the information about its source form (the model) at run-time. However, OOIS UML allows that concrete implementations combine compilation with interpretation by means of reflection.

Basic and Derived Concepts

In order to have formal semantics that would allow models to be executable, the OOIS UML profile separates two kinds of its concepts, basic and derived.

The *basic* concepts are the concepts that have elementary, executable semantics and form the fundamentals of the profile. The *derived* concepts have their semantics defined in terms of some other concepts. The latter are introduced to enrich the expressiveness of the language, but their execution semantics can be derived from the semantics of the basic concepts.

For example, classes, associations, and constraints are basic concepts of OOIS UML, but the multiplicity of association ends is a derived concept — the semantics of the multiplicity of an association end can be expressed in terms of constraints attached to the association end.

This separation was necessary to clear the otherwise flawed execution semantics of some initial concepts of standard UML that are intuitively clear and practically very useful, but imprecise and thus not formally interpretable.

Section Summary

- The basic concepts have elementary execution semantics.
- The derived concepts have their semantics defined in terms of some other concepts. The derived concepts are introduced to enrich the expressiveness of the profile, but their execution semantics can be derived from the semantics of the basic concepts.

Formal and Informal Concepts

Some concepts of OOIS UML have precise run-time semantics that allow their formal interpretation. For example, classes, attributes, associations, or formal OCL constraints fall into this category. These are called *formal concepts*. However, some concepts of OOIS UML do not have executable semantics. For example, use cases or interactions fall into this category. These are called *informal* or *illustrative concepts*. As described later in this chapter, informal concepts are useful for specifying or illustrating the intention of the modeler about the structure or behavior of the system, but do not fully and formally define or ensure such structure or behavior. Instead, they can be used to sketch or document the structure or behavior, or explain some subtle parts of the model to the reader, but not to define the executable properties of the model.

Section Summary

- Formal concepts of OOIS UML have precise run-time semantics that allow their formal interpretation.
- Informal concepts are used for specifying or illustrating the intention of the modeler about the structure or behavior of the system, but do not fully and formally define or ensure such structure or behavior.

Structure and Behavior

On the one hand, some concepts of the standard UML and OOIS UML specify the *structure* of the system. These concepts include classes, attributes, associations, and so on. They specify the shape of the object structure.

On the other hand, some concepts deal with the *behavior* of the system. These concepts include operations, methods, state machines, use cases, commands, and so on. They specify the dynamic behavior of the system, as a series of changes to the system over time. In other words, they describe the functionality of the system.

The structure-behavior dichotomy is one of the fundamental dichotomies in UML. The “structure” side of this dichotomy corresponds to the “data-modeling” or “data-definition” side of the earlier modeling and design methods of information systems. The modern terminology suggests usage of the term “structure,” because it is more general, and the structure is now constructed of objects, not simple data, as explained before. Similarly, the “behavior” side corresponds to the “functional” side in the earlier methods. These two terms can be used interchangeably because they refer to the functionality of the system.

Section Summary

- Some concepts specify the structure of the system.
- Some concepts deal with the behavior of the system.

Core and Extended Parts

The *core part* of the profile comprises the language concepts, basic and derived, formal and informal, structural and behavioral. They are general enough to form the language itself.

The *extended part* of the profile, on the other hand, comprises some ready-to-use and commonly needed models, designed using the core part. They form the OOIS UML model library. They are not tailored to a specific problem domain. Instead, they are general enough to be useful in many concrete applications, but not so primitive to be part of the language. Very often, the concepts defined in the extended parts can be defined in many different ways, of which only some are suitable for a particular problem. This approach illustrates how the core part can be used to create many useful extensions of the profile for specific application domains. For example, commands and GUI configuration form the extended part of the profile.

Section Summary

- The *core part* of the profile comprises the language concepts, basic and derived, formal and informal, structural and behavioral.
- The *extended part* of the profile comprises some ready-to-use and commonly needed models, general enough to be useful in many concrete applications, and designed using the core part. They form the OOIS UML model library.

Model Elements and Diagrams

UML is primarily a *language*, which encompasses the definition of the concepts — their semantics, properties, and relationships. A model of a system consists of *model elements*, which are instantiations of the language concepts. For example, class is instantiated in Course, Person, and Teacher in the Easylearn school model, whereas association is instantiated in attends and teaches in the model. The model is, therefore, a collection of many model elements, such as classes, associations, operations, methods, generalization/specialization relationships, and many others. These elements of a model comprise the M1 level and are instantiations of the language concepts defined at the M2 level. The set of the language concepts and their properties and relationships, along with the rules about how model elements can be combined to form correct models, is often referred to as the *abstract syntax* of the language.

On the other hand, the *notation* for the language concepts defines their representation in a human- or machine-readable form. The definition of the notation is also referred to as the *concrete syntax* of the language. The notation defines how model elements are written or depicted, but it does not define their semantics. UML combines textual and visual (graphical) notations. The combination tends to present the underlying model elements in the most appropriate and readable forms.

It should be particularly emphasized that the most important aspects of the language are its concepts and their semantics, and that the elements of a model, not their notation, carry the information that will be manifested in the running system. Moreover, there can be many possible notations that represent the same model elements.

First, standard UML defines several different human- or machine-readable notations [UML2]. One is the usual, combined graphical and textual notation that is aimed for modeling and that is used throughout this book. Another is a textual (sequential) XML-based notation aimed at interchanging models between modeling tools (the so-called *XML Model Interchange* standard notation, or XMI [XMI]).

Second, even the basic UML notation allows several notational options for some concepts. For example, a class can be rendered as a simple rectangle, or as a rectangle divided into several compartments with its features and constraints, and so on. Moreover, there are different *style guidelines* that do not constitute mandatory rules for the notation, but simple recommendations of how different notational styles can be used in practice. For example, one may show all features of a class within their compartments when this is needed, and suppress them in other contexts where merely referring to the class.

Elements of a model are rendered in UML *diagrams*. Diagrams are pictorial views of parts of the model (that is, *projections* of the underlying model). Diagrams do not carry semantics — they are just partial graphical representations of the underlying model. Model elements carry the semantics, and diagrams contain their *symbols* in the defined notation. Such an approach has several important implications.

UML takes the *model-centric* approach, instead of *view-centric*, as illustrated in Figure 7-1. In some other modeling approaches that take the view-centric approach (see Figure 7-1a), the model is exactly what is depicted in a diagram. In view-centric modeling approaches, there is often only one single diagram that is de facto the model. In such approaches, the model elements are implied from the symbols depicted in the diagram(s). In UML, this is the opposite (see Figure 7-1b). The model consists of model elements that are created, manipulated, and stored in a way under the control of the modeling tool. The diagrams, on the other hand, offer different views to the model. Note that

Part III: Concepts

the model-centric approach is closer to the approaches used in other engineering disciplines, such as mechanical or civil engineering, as shown in Figure 7-1b. A model in UML corresponds to the physical device or building in mechanical or civil engineering (or their physical scale models), while UML diagrams correspond to the projections of these physical things from different viewpoints.

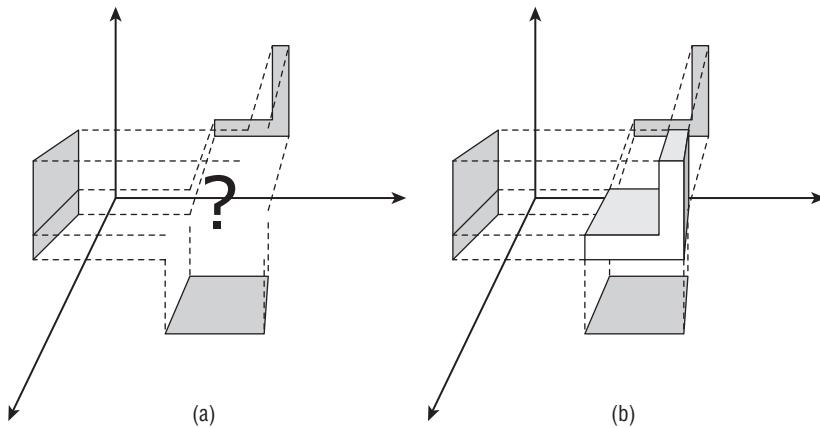


Figure 7-1: View-centric vs. model-centric approach. (a) In a view-centric approach, the content of the diagram is de facto the model. (b) In a model-centric approach, the diagrams represent partial pictorial views or projections of the model.

A diagram may depict just one part of the model. Basically, except for some trivial systems, individual diagrams generally render just fragments of models. Complex models are always displayed in many diagrams. Each diagram focuses on a part of the model to which the modeler wants to draw attention. For example, the diagram in Figure 7-2a focuses on the two classes, `Course` and `Person`, from the model of the Easylearn School system, their features, and their relationship. Another diagram in Figure 7-2b introduces a new specializing class `Teacher`, its features, and relationships to the other classes. The other classes are thus just referenced in this diagram, and their features are suppressed, although they still exist in the model.

The same model element can be rendered in many diagrams. Each of those diagrams contains a symbol of the model element, not the model element itself. In other words, diagrams do not contain the actual model elements, but merely their graphical representations (symbols). A symbol from a diagram can be removed without removing the corresponding model element. Even the entire diagram can be removed without affecting the model.

A model element can be depicted in several diagrams in different ways, even using different symbols. For example, in Figure 7-2a, the class `Course` is shown with all its features, while in Figure 7-2b, its features are suppressed, although they exist in the model. This is because the diagram in Figure 7-2b emphasizes only the new class, `Teacher`, with its features and relationships with the other classes, and thus simply refers to the classes `Course` and `Person`. Moreover, a symbol for a class can contain only some of the class's features, those on which the diagram is focused. Similarly, the same model element can be rendered in the same diagram with several symbols, if this is needed to improve the readability of the diagram.

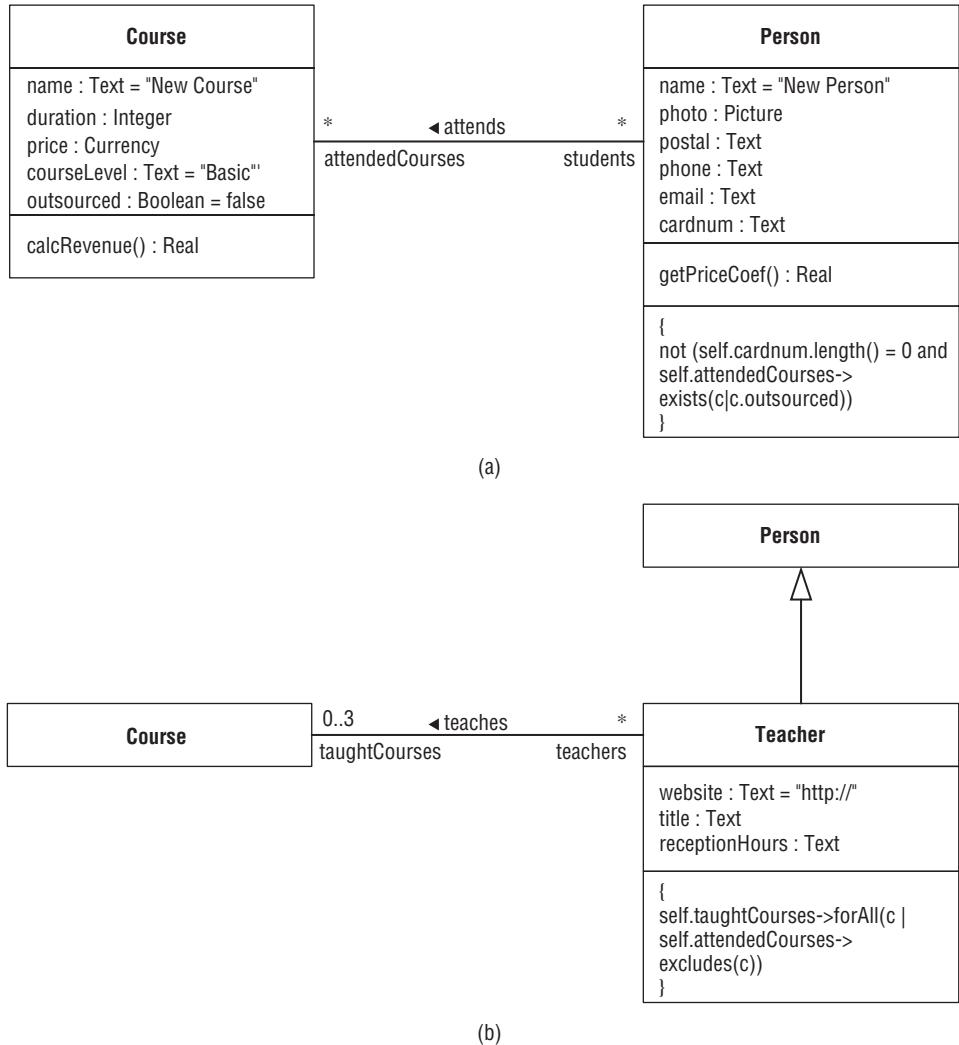


Figure 7-2: The same model element can be rendered in an arbitrary number of diagrams. In every diagram, the symbol of the model element can be different and thus, offer a different projection of the same contents. (a) This diagram focuses on the classes `Course` and `Person`, their features, and their relationship. (b) This diagram introduces a new specializing class `Teacher`, its features, and relationships to the other classes. The other classes are thus just referenced, and their features and relationship are suppressed, although they exist in the model.

Diagrams are not elements of the model — they display the model elements instead. Nevertheless, modeling of complex systems always encompasses managing many diagrams within the modeling tool. In order to help the modeler to cope with lots of diagrams, modeling tools often organize diagrams into different kinds of hierarchies, sometimes in the same hierarchies of model elements. However, this is not a language requirement.

Part III: Concepts

A diagram should be focused and not cluttered. Because UML diagrams usually comprise some graphical nodes connected by arcs, a diagram should not contain too many nodes — at most, five to eight nodes are recommended, because this is the amount of information a person can handle at a glance. Cluttered diagrams should always be avoided. Instead, several diagrams should be used to offer different projections onto parts of the system, each from a different viewpoint, focused on a different kind of information, and with a clear intention and message to the viewer.

In general, UML does not constrain which kinds of model elements can appear in one diagram — any kind of model element can appear in any diagram. However, modeling of a certain structural or behavioral aspect of the system usually assumes using only some kinds of model elements to be shown in a diagram. Therefore, UML prescribes 13 kinds of diagrams,² each of which assumes solely or primarily some kinds of model elements. OOIS UML uses only a subset of these kinds of diagrams, as shown throughout this book.

There are two general kinds of diagrams in UML. *Structure diagrams* show the static structure of the objects in the system. They depict those model elements that specify the system's characteristics irrespective of time. Structure diagrams do not show the details of dynamic behavior, but may still show the relationships of classifiers in the diagram to the behaviors they exhibit. *Behavior diagrams* show the dynamic behavior of the system as changes to the system over time. Certain diagrams may show both structure and behavior, however.

Section Summary

- ❑ The *language* defines the concepts, their semantics, properties, and relationships. The *notation* defines the textual, graphical, or combined representation of the language concepts.
- ❑ A model consists of *model elements*, which are instantiations of language concepts and which carry the semantic information. *Diagrams* are graphical representations of the model and comprise *symbols* of model elements.
- ❑ Diagrams have the following characteristics:
 - ❑ They do not carry semantics; they simply display parts of the model.
 - ❑ They do not contain model elements, but contain symbols. A symbol can be removed from a diagram without affecting the model.
 - ❑ They are projections of parts of the model, meaning that they contain symbols of just some model elements, those on which the modeler wants to focus the viewer's attention. Different symbols for the same model element can be used in different diagrams.
 - ❑ They should not be cluttered. They should emphasize only parts of the model of interest in the context.

²This number has been changing from version to version of UML. However, it is not part of the core language, but a matter of the way of its common usage in practice.

- UML does not constrain the use of symbols of different modeling concepts in the same diagram. However, some aspects of the system are depicted in different kinds of diagrams that solely or primarily contain just some kinds of model elements.
- *Structure diagrams* depict static structure of the system. *Behavior diagrams* depict dynamic aspects of the system changing over time. Certain diagrams may show both structure and behavior.

General Language Concepts

This section introduces some core concepts of UML that represent generalizations of many other language concepts. These general concepts gather common characteristics of some other language concepts. These are the concepts of element, namespace, named element, multiplicity element, and packageable element. Their characteristics and the related concepts of comments and visibility are presented, too. Finally, this section describes packages and other related elements of UML that are aimed at organizing complex models into hierarchies and that support reuse of parts of models.

Elements and Comments

A *model element* (or *element* for short) is an abstract generalization of all concepts in UML. In other words, all other language concepts are kinds of elements and inherit all its language features. Therefore, the concept of element generalizes the characteristics of all other language concepts — whatever holds for elements in general, holds for any concrete language concept. An element is a constituent of the model.

One such common thing is a *comment*, which is a textual annotation that can be attached to a set of elements. A comment carries no semantics, but is simply a remark useful to the human reader of the model.

A comment consists of an arbitrary text, not interpreted by the modeling tool or run-time environment because it carries no formal modeling or run-time semantics. Any element can *own* several comments. A comment exclusively belongs to the owner element, and is removed when the owner element is deleted from the model. A comment can be owned by at most one element, but it can also be independent. Additionally, a comment can *annotate* arbitrarily many elements. A comment is also a model element, so it also can be commented.

The symbol for a comment is a “dog-eared” rectangle with the upper-right corner folded over, as shown in Figure 7-3. It is also known as the “note” symbol. The content of the comment is shown as the text within the rectangle. The connection to each annotated element is shown as a separate dashed line. The dashed line can be suppressed when it is clear from the context, or not important in the diagram.

The content of a comment can be, for example, a description or documentation of the annotated element(s), the reference to the version or date when the element was introduced, the author of the element, or any other descriptive text informative to the reader or the modeler.

Part III: Concepts

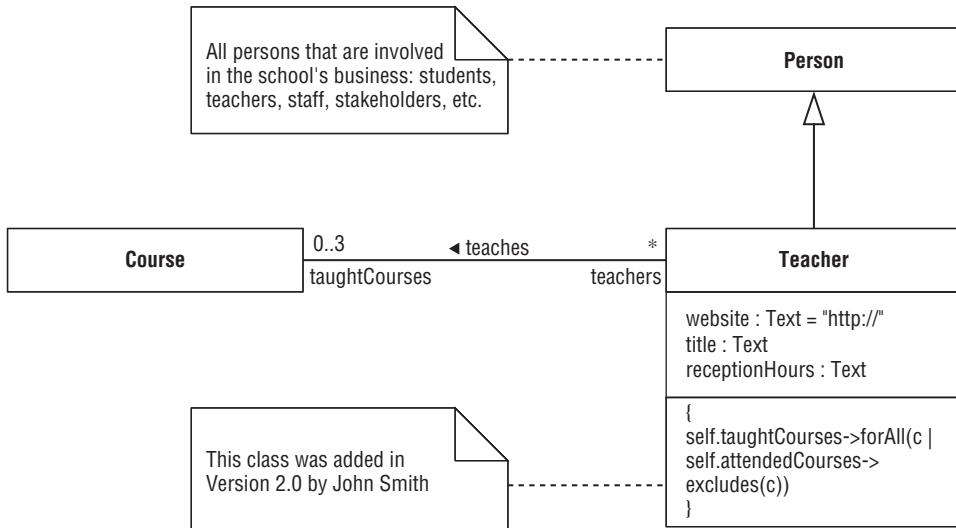


Figure 7-3: A comment is shown as a “dog-eared” rectangle with the upper-right corner folded over (the “note” symbol) and with its textual contents written inside the rectangle. The dashed lines connect the comment with the annotated elements.

Section Summary

- ❑ An *element* is a generalization of all other language concepts. An element is a constituent of the model.
- ❑ A *comment* is a textual annotation that can be attached to a set of elements and that can be owned by an element. It carries no semantics, but represents information useful to the human reader of the model.

Packages

Models of information systems are typically very complex. They consist of dozens or even hundreds of classes, relationships, and many other elements. If such huge collections of model elements were flat and could not be organized into a structure that is easy to understand and manage, the problem of maintenance of models would quickly become intractable.

Basically, the problem is similar to the problem of maintaining a huge number of files on a disk. Suppose you have a flat file system that does not recognize the concept of folder (or catalogue, or directory), but in which all files are placed in a unique, unstructured storage. After some use, the user’s disk would become cluttered, storing hundreds of files. This is why folders exist — a folder is an element of the file system that can own many other elements of the file system, including files and other folders. In that way, the file system is organized as a hierarchy of folders, which can be rendered as a tree-shaped structure, as in

modern graphical file browsers. It has been proven that such hierarchical structures improve navigability and manageability of huge amounts of items.

Because the problem of organizing complex models is similar to the problem of organizing file systems, the solution is also similar. Much like file systems have folders, UML introduces the concept of a *package*. A package is a model element that can own some other kinds of model elements, such as classifiers and other packages. This is why the entire set of model elements that constitute the model can be viewed as a hierarchical structure of packages, which, in turn, can contain other packages or other kinds of model elements. Packages represent the general grouping mechanism in UML.

A package cannot own all kinds of model elements. For example, it cannot own an operation or a property, because an operation is owned by its classifier, and the classifier itself can be owned by a package. Instead, the language concept of *packageable element* generalizes all those kinds of elements that can be directly owned by a package. Classifiers and constraints are packageable elements, for example. Of course, packages are also packageable elements, so they can be owned by packages. The packages owned by a package are referred to as the *nested packages* of the latter, whereas the latter is called the *nesting* (or *enclosing*) *package* of the former.

A packageable element can be (directly) owned by at most one package. Only the topmost packages in the hierarchy can be independent elements that are not owned by any other packages — all other kinds of packageable elements must be owned by a package. A package can own many packageable elements. Of course, such an owning relationship cannot be cyclic, meaning that a package cannot directly or indirectly own itself. The elements owned by a package are existence-dependent on the owner package — when the owner package is removed from the model, all its owned elements are also removed, including all nested packages recursively.

A package is a modeling concept that has no direct manifestation at run-time — packages have no run-time semantics.

The symbol for a package in diagrams is a larger rectangle with a smaller rectangle (“tab”) attached to its top-left, akin to a symbol for folder, as shown in Figure 7-4. The same figure shows several notational options for rendering packages. In Figure 7-4a, the package is shown without displaying its contents (that is, its owned elements). In Figure 7-4b, some of the contents of the package (for example, the owned classes) are shown within the large rectangle. Finally, in Figure 7-4c, some of the contents of the package are shown outside the large rectangle, connected by a branched line with the package, and with a plus sign within a circle at the end attached to the package. If the contents of the package are shown within its symbol, the name of the package should be written in the small rectangle. If the contents of the package are not shown within the large rectangle, the name of the package should be written within the large rectangle.

As is the case with organizing files into folders, there is no general strict rule about how to organize model elements into packages. However, one general recommendation is that a package should own those elements that are coherent, that is, strongly coupled between themselves, and loosely coupled with the elements outside the package. One other criterion for grouping a set of elements into a separate package is a prospect or an existing need for reuse of the elements in different modeling contexts. In other words, a package can represent a unit of model reuse. Of course, these are highly subjective criteria whose interpretation is up to the modeler. Practical experience with complex models can help the modeler establish a good feeling for this matter.

Part III: Concepts

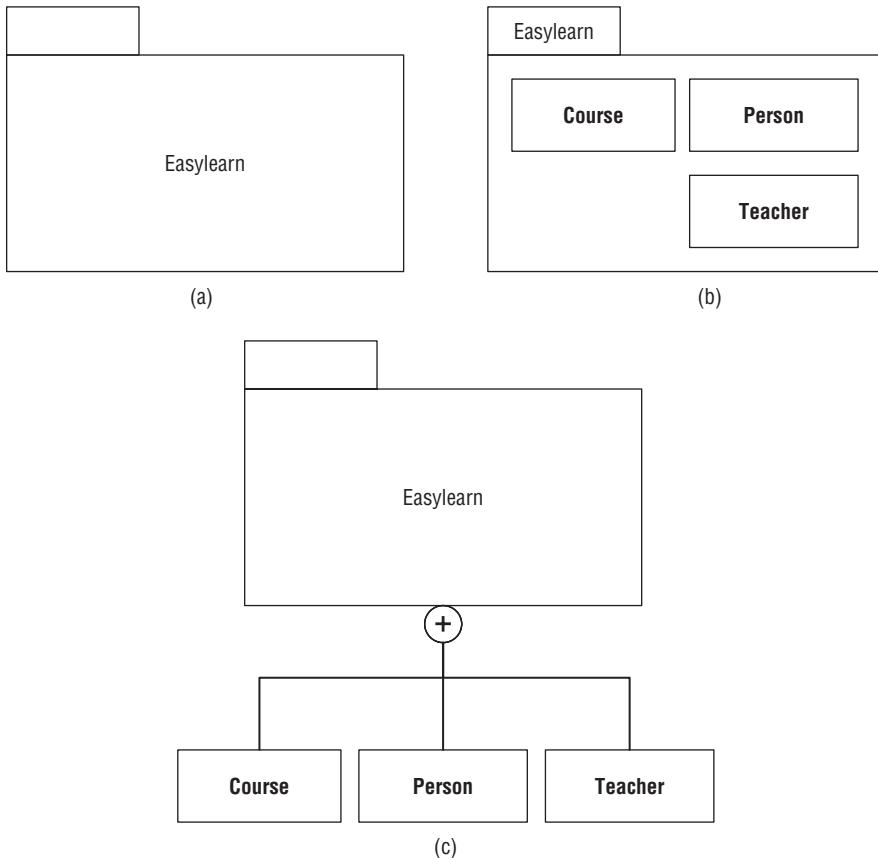


Figure 7-4: Notational options for package. (a) A symbol for a package without displaying its owned elements. (b) A symbol for a package with some of its contents displayed within the symbol. (c) A symbol for a package with some of its contents displayed outside the symbol.

Section Summary

- ❑ Packages represent a general grouping mechanism of UML. A package can own some other kinds of elements, including other packages, too. This way, models are organized into hierarchies of packages.
- ❑ Elements that can be directly owned by a package are called *packageable elements*. A packageable element can be owned by at most one package. Only the top-most packages in the hierarchy can be independent elements that are not owned by any other package.
- ❑ A package can own many packageable elements. The elements owned by a package are existence-dependent on the owner package.

- Packages have no run-time semantics.
- A package should own coherent elements, loosely coupled with the elements outside the package. A package may represent a unit of reuse.

Namespaces and Visibility

This section examines named elements, namespaces, visibility, importing of elements, and importing of packages.

Named Elements

Elements of most kinds can have *names* in UML. The name of an element is a string of characters that can be used to identify the element within the given context. For example, classifiers, operations, and properties may have names. The name of a class identifies that class within the package that owns that class. The name of the operation can be used to identify the operation within its owning class.

Elements that may have names in UML are called *named elements*. Examples of named elements include classifiers, operations, and properties. Packageable elements are named elements.

A name is a string (that is, a sequence of characters in a certain character set). UML does not specify a concrete character set to be used for names. Instead, some assumptions are made:

- The character set is sufficient for representing multi-byte characters in various human languages. In particular, the traditional 8-bit ASCII character set is not sufficient.
- Tools and run-time environments manipulate and store strings correctly, including escape conventions for special characters.
- A string is displayed as a textual graphic. Normal printable characters should be displayed correctly. The display of nonprintable characters is unspecified and platform-dependent.
- There are no language-defined limitations in lengths of names. All characters in names are significant.
- Every character in the character set is different from all others. In particular, names are case-sensitive.

Therefore, the treatment of strings is up to the implementation. Unfortunately, this may lead to a reduced portability of models. For that reason, OQIS UML gives the following guidelines when naming elements in order to avoid portability problems, especially when a traditional programming language (such as C++, Java, or C#) is used as the detail-level and implementation language of models:

- Always use a standard character set like Unicode.
- Do not use non-printable characters and white spaces (blanks, tabs, new lines, and so on) in names of elements. Limit yourself, if possible, to Roman alphanumerical characters in names of elements.
- Always start names of elements with a letter from the Roman alphabet.

Part III: Concepts

- ❑ Avoid extremely long names of elements. Especially avoid names longer than 255 characters. After all, extremely long names, although readable by tools and computers, are not understandable by humans.
- ❑ Do not use empty strings as names.

Following are other important style conventions (not only in UML, but also in other modern OO approaches):

- ❑ Names of classes, data types, and some other classifiers start with an uppercase character. Names of other elements (such as associations, operations, and properties) start with a lowercase character.
- ❑ Instead of using an underscore (_) or a blank to separate words of a human language within a name, use capitalization, as in the following names: `allStudentsOfCourse`, `BankAccount`, `getPriceCoef`, `UserConfiguration`, and so on.

Note that a named element does not have to have a name. Moreover, the absence of a name is different from an empty name. An empty name assumes that the name exists, but is an empty string (a string of zero length).

Section Summary

- ❑ A *named element* is a model element that may have a *name*. The name of an element is a string of characters that can be used to identify the element within a certain namespace.
- ❑ The name of an element is a string (that is, a sequence of characters in a specific character set). The language does not specify the character set.

Namespaces

The context within which a named element can be referred to by its name is called the *namespace*. A namespace is an element in the model that can own a set of named elements. These elements can be identified by name within their owning namespace. For example, a package is a namespace that owns packageable elements, and a class is a namespace that owns features. Namespace is yet another generalization of other UML concepts.

Namespaces are also named elements, meaning that namespaces may have names. Therefore, namespaces can own namespaces, among other named elements. This does not mean, however, that any kind of namespace can own any kind of named element, but just some of them, according to its specific kind. This relationship simply specifies the potential of namespaces to own named elements, by virtue of being some more concrete kinds of namespaces. For example, a package is a namespace, but can own packageable elements only. It cannot own operations directly, although operations are also named elements. Similarly, a class can own features only, but not packages, although packages are also named elements. Operations, although they are namespaces, cannot own packages, classes, or other operations, although these are also named elements.

The named elements owned by a namespace are called *owned members*. For example, package is a namespace that can own other packages or classes as its owned members, which are also namespaces. In other

words, namespaces can be nested. A namespace can own many other named elements, including nested namespaces, and can be owned by at most one enclosing namespace.

A named element can be referred to by its name within the contexts that use textual representations. For example, a method specified in a textual detail-level language can refer to named elements by their names. Similarly, OCL constraints refer to named elements by their names because such constraints are written in a textual language. It is a question, however, of how to identify a certain named element from within one namespace, while the referred element belongs to another namespace.

There are several language rules that define how a named element can be referred to by its name from within a certain namespace. To illustrate the rules, let's assume a sequence of nested namespaces — three nested packages named `pkg1` (the outermost), `pkg2`, and `pkg3` (the innermost), and a class named `C` owned by the innermost package `pkg3`. Let `p` be a property owned by `C`. Following are the rules:

- ❑ Within the same owning namespace, a named element can be identified directly by its (unqualified) name. For the given example, the property `p` can be referred to directly by its name within the namespace `C`.
- ❑ A named element is implicitly introduced into a namespace that inherits the namespace owning the named element. For the given example, within a class `D` that is a specialization of the class `C`, the named element `p` can be referred to directly by its name, unless there is another named element directly owned by `D` with the same name as `p`, which then hides that inherited element.
- ❑ Within a nested namespace, a name from the enclosing namespace can be referred to directly by its name, unless it is hidden by another named element with the same name in a more nested enclosing namespace. For example, all names from the packages `pkg1`, `pkg2`, and `pkg3` can be referred to from within the namespace `C`.
- ❑ Within any namespace, a named element can be referred to by its *qualified name*, which represents the full path to the named element starting from the outermost namespace, in the following syntax:

Namespace1::Namespace2::...::namedElement.

For the given example, the property's fully qualified name is `pkg1::pkg2::pkg3::C::p`.

- ❑ The qualified name can represent a relative path to the named element from the given namespace. The given named element can be referred to by its partially qualified name from a certain namespace if the first name in the qualified name can be referred to from that namespace. For the given example, the property `p` can be referred to from the namespace `pkg2` by its qualified name `pkg3::C::p`, or from the namespace `pkg3` by its qualified name `pkg2::pkg3::C::p`, or by `C::p`. It cannot be, however, referred to by `pkg3::C::p` from `pkg1` because `pkg3` cannot be directly referred to from `pkg1`.

The default language separator between the names in qualified names is the double colon (`::`). However, an implementation may use another separator in certain contexts. For example, if the implementation uses Java or C# as the detail-level language, within the code of methods the separator is a dot (`.`).

As named elements, namespaces can (but need not) have names. If a namespace does not have a name, all its direct or indirect members do not have qualified names.

Because constraints are packageable elements, and therefore, also named elements, a namespace can own constraints. Those constraints do not necessarily apply to the namespace itself, but may also apply to

Part III: Concepts

the members of the namespace. The constraints owned by a namespace are evaluated within the scope of that namespace, meaning that the names referenced within the constraints are resolved according to the same given rules. For example, a constraint attached to a class as its context can refer to all those and only those elements that can be referred to from that class as a namespace. In particular, the constraint can directly refer to the class's features.

Any two named elements within the same namespace must be *distinguishable*, meaning that they can logically coexist within the same namespace. In general, two named elements are distinguishable if they are of unrelated kinds, or if they have different names. For example, a class and a package owned by the same package can have the same name because they are different kinds of elements. On the other hand, two classes in the same package must have different names. Similarly, any two properties of the same class must have different names.

However, this rule is overridden for some kinds of named elements. For example, operations of the same class do not have to have different names. Operations are distinguished by their *signatures*, which, apart from their names, include the types of their parameters. In other words, two operations of the same class are distinguishable if they have different names, or if they have different sequences of the types of their parameters. For example, all of the following three operations are distinguishable and thus, may coexist in the same class:

```
doSomething(X, Y, Z) :T1  
doSomething(X, Z, Y) :T1  
doSomethingElse(X, Y, Z) :T1
```

An implementation may further constrain this rule. For example, if the implementation uses C++ as the detail-level language, two operations are not distinguishable if they differ only in their return type. This is why such cases should be avoided in models. Anyway, they are very rarely needed, and always reduce the clarity of classes.

Section Summary

- ❑ A *namespace* is a named element that can own other named elements, including nested namespaces.
- ❑ A named element can be referred to by the following:
 - ❑ Its unqualified name from within its owning namespace, or within a namespace nested in its owning namespace, or within a namespace that inherits its owning namespace
 - ❑ Its fully qualified name from within any namespace
 - ❑ A partially qualified name that is a relative path to it from a namespace, provided that the first name in the qualified name can be referred to from the namespace
- ❑ Any two members of a namespace must be distinguishable. Two named elements are distinguishable if they are of different kind, or if they have different names. Operations of a class do not have to have different names, but different signatures.

Visibility

UML supports encapsulation at the level of a classifier in a similar way as in traditional OO programming languages. A classifier's feature (property or operation) can have one of the following visibility types:

- Public** — The feature is accessible from any place where the classifier is accessible.
- Protected** — The feature is accessible from the same namespace (classifier), the nested namespaces (operations and methods), and the inheriting namespaces (specializing classifiers).
- Private** — The feature is accessible from the same namespace (classifier) and the nested namespaces (operations and methods), but not from inheriting namespaces.
- Package** — The feature is accessible from the nearest enclosing package of the classifier, but not outside it. For example, such a feature can be accessed from the classes owned by the same package.

Moreover, UML recognizes bigger encapsulation entities — packages. Elements of a package have their visibility, which can be either public (accessible from any place from where the package is accessible) or private (accessible only from the same package).

The visibility of a named element is denoted with the symbol + for public, # for protected, - for private, and ~ for package visibility, preceding the textual representation of the element, as shown in Figure 7-5. Figure 7-5a shows how the visibility of classes' features can be indicated, and Figure 7-5b shows how the visibility of a package element can be indicated by preceding the name of the element by the visibility symbol.

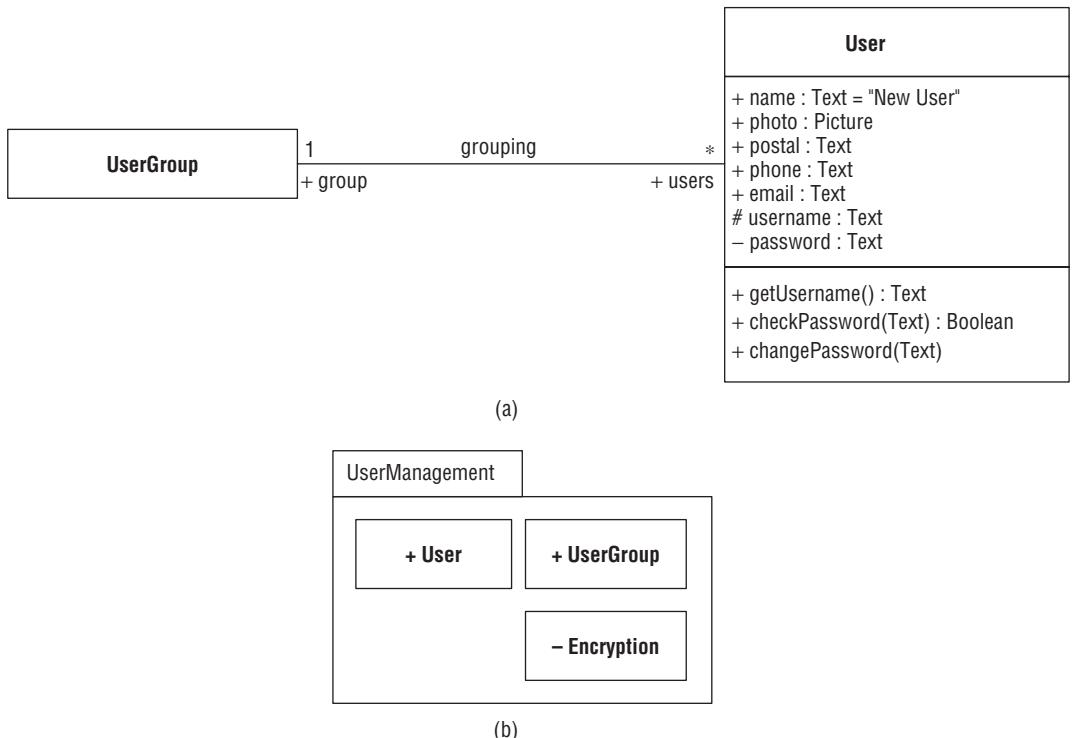


Figure 7-5: Notation for visibility: (a) features of classes; (b) elements of packages.

Part III: Concepts

Figure 7-6 illustrates the visibility rules on several examples. It is important to notice that the reference to `EncryptedText`, `Encryption::EncryptedText`, and `DataTypes::Encryption::EncryptedText` in Figure 7-6 a, b, c, and d, as the type of the attribute `User::password` is not a reference by name. Instead, the type of the attribute refers to the named element directly, by an internal relationship embodied in the model. It is the responsibility of the modeling tool to provide the means to specify, read, and modify this reference. The name of the attribute type displayed in the diagram is simply a textual representation of this internal relationship. Consequently, if the name of the referenced element is changed, for example, the tool will automatically update the textual representation displayed in the diagrams. This is yet another manifestation of the model-centric approach of UML, whereby the textual representation is just a partial perspective of the internal structure of related model elements.

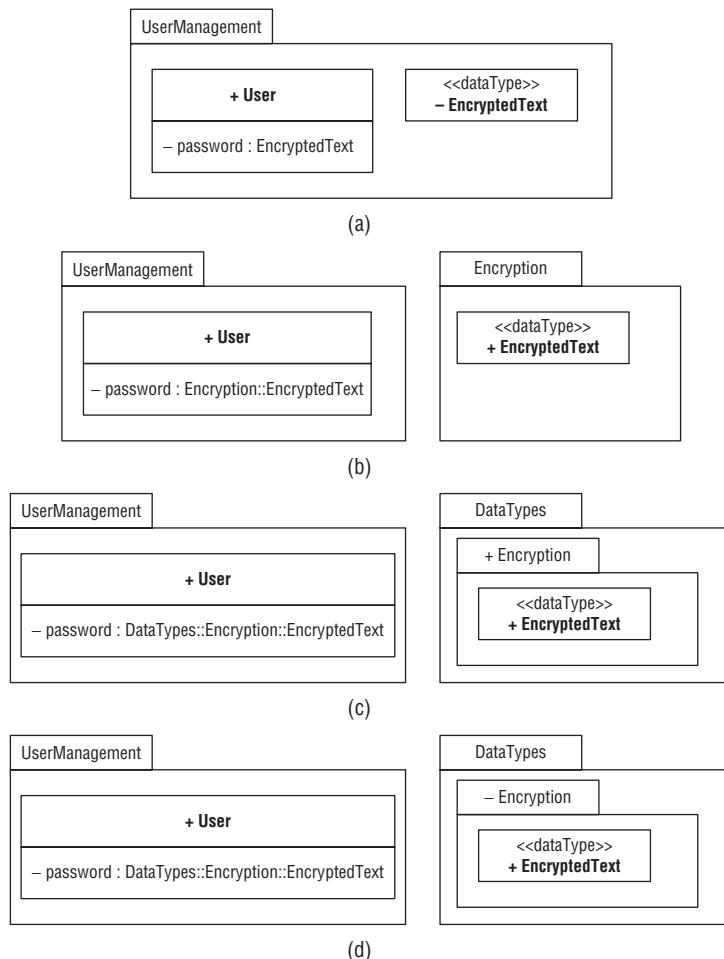


Figure 7-6: Examples of visibility. (a) Correct: the type of the attribute `User::password` refers to the (private) element of the same package. (b, c) Correct: the type of the attribute `User::password` refers to an accessible element of another package. (d) Incorrect: the type of the attribute `User::password` refers to an inaccessible element of another package, because `DataTypes::Encryption` is private.

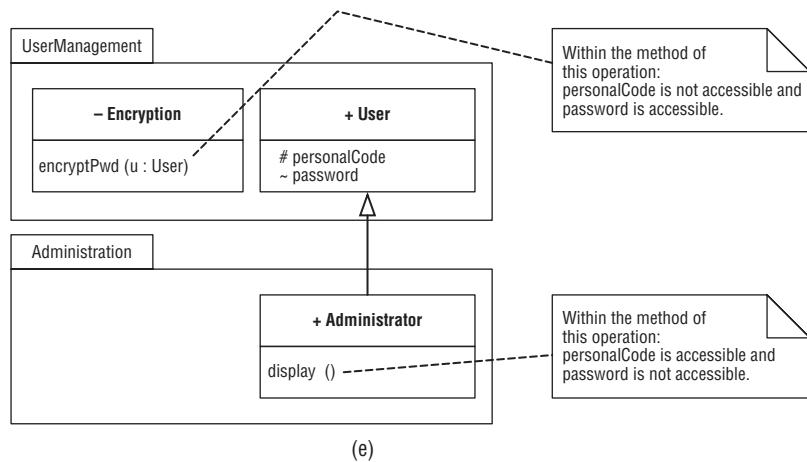


Figure 7-6: (e) Illustration of protected and package visibility.

Following are the models shown in different parts of Figure 7-6:

- a** — Correct, because the type of the attribute `User::password` refers to the (private) element of the same package.
- b, c** — Correct, because the type of the attribute `User::password` refers to an accessible element of another package.
- d** — Incorrect, because the type of the attribute `User::password` refers to an inaccessible element of another package because `DataTypes::Encryption` is private.

Figure 7-6e illustrates protected and package visibility.

In general, the encapsulation principle is supported in UML by the means of *visibility* at the level of all kinds of namespaces. Each named element may have its visibility defined within its owner namespace. In other words, a namespace provides encapsulation for its members because each member of a namespace may define its visibility. Although a member of a namespace can be referred to by its name according to the rules provided in the earlier section, “Namespaces,” or directly by other means of referring to model elements from other elements, it may not be *accessible* (or *visible*) because it is hidden (encapsulated) within its namespace.

Section Summary

- Each named element may have its *visibility* defined within its namespace. The visibility types are as follows:
 - Public (denoted with +)**—A public member of a namespace is accessible to all other elements of the model that can refer to that member, and that can access its owning namespace.

Continued

- ❑ **Private (denoted with -)** — Private members are accessible within the same owning namespace and the nested namespaces. They are hidden (or encapsulated) for namespaces other than nested namespaces.
- ❑ **Protected (denoted with #)** — Protected members of a namespace are accessible from the same namespaces as private members, as well as from the namespaces that have a specialization relationship toward this namespace.
- ❑ **Package (denoted with ~)** — Only elements that are owned by a namespace that is not a package can be of this visibility type. They are accessible from the same namespaces as private members, as well as from the nearest enclosing package, given that other enclosing namespaces (as named elements) have proper visibility. Outside that nearest enclosing package, such an element is not visible.

Importing Elements

As you have seen, within a textual context such as a method of operation, a named element belonging to another non-enclosing and non-inherited namespace must be referred to by its qualified name, according to the listed rules. A name without a qualifier would refer to a named element owned either by the same namespace, or by one of the enclosing or inherited namespaces. If a named element owned by another namespace is intensively used within the given namespace, however, using its qualified name at all places may be tiresome.

For this reason, there is a way in UML to *import* a named element into a namespace other than its owning namespace. The element becomes a member of the target importing namespace and can be referenced from that namespace using its name without qualifier. Only packageable elements accessible from the importing namespace can be imported.

In UML diagrams, an element import is rendered as a dashed arrow with an open arrowhead from the importing namespace to the imported element (see Figure 7-7a). The keyword `<<import>>` is shown near the dashed arrow.

In general, an element import is a directed relationship between the importing namespace and the imported packageable element. As a result of this relationship, the imported element (`EncryptedText` in Figure 7-7a) is added to the importing namespace (package `UserManagement` in Figure 7-7a), so it becomes its member, although not directly owned by that namespace. Therefore, the imported element can be referenced from the importing namespace by its name without qualification, unless there already exists an element with that name in the importing namespace.

An element import relationship can control whether the imported element can be further imported from the target namespace. The element import relationship may have its *visibility*, which can be either private or public. If the visibility of the import relationship is public, the imported element becomes a public member of the importing namespace and can be further imported to other namespaces. If, on the other hand, the relationship has private visibility, the imported element becomes a private member of the importing namespace, and, thus, it can be used within that namespace without qualification, but it cannot be further imported from that namespace. A private element import relationship is rendered as a dashed arrow with the keyword `<<access>>` instead of `<<import>>` (see Figure 7-7b).

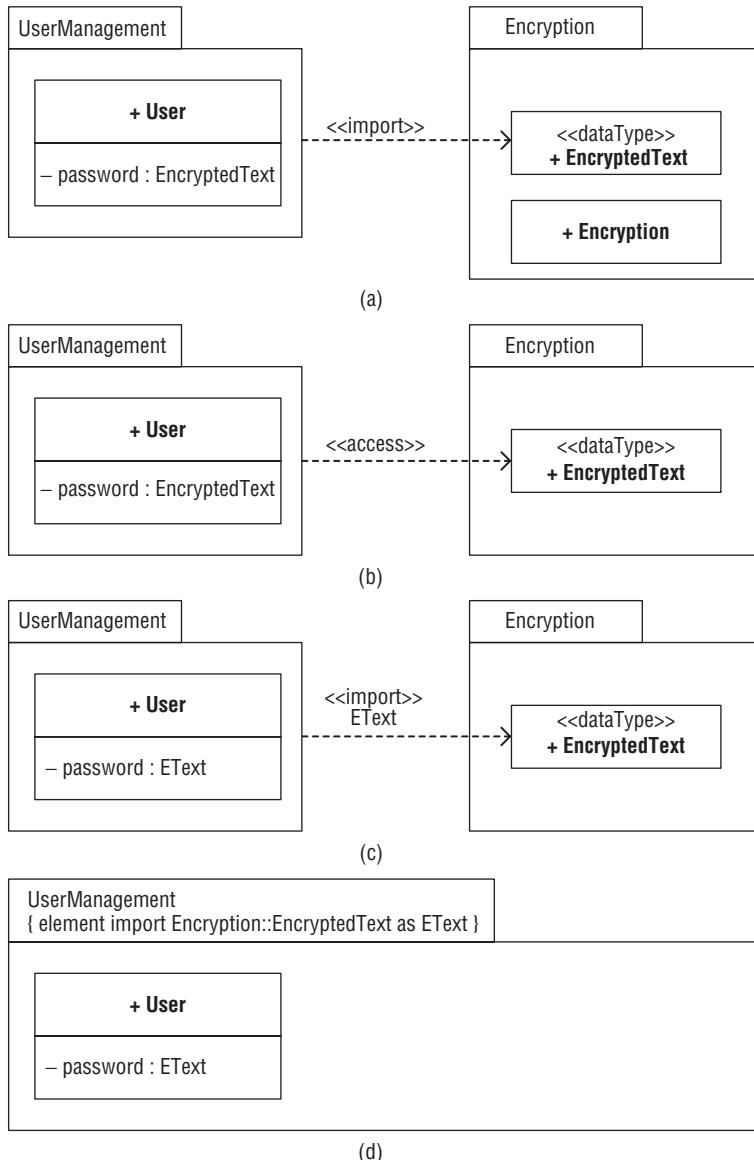


Figure 7-7: Examples of element import: (a) The package `UserManagement` imports the element `EncryptedText`, which allows the elements of `UserManagement` to refer to it by its name without qualification. However, they still have to refer to `Encryption::Encryption` by its qualified name, because it is not imported. `EncryptedText` can be further imported from `UserManagement`. (b) An example of a private element import. The imported `EncryptedText` can be used without qualification within `UserManagement`, but it cannot be further imported from `UserManagement`. (c) Example of an element import with an alias. The imported `Encryption::EncryptedText` can be referred to by its alias `EText` within `UserManagement`. (d) Alternative textual notation for element import.

Part III: Concepts

The default visibility of element import relationship, if not specified, is the same as the visibility of the imported element. The imported element must have either public visibility or no visibility at all; otherwise, it cannot be imported (because it is not accessible). The visibility of element import can be either public or private. Consequently, it can be either the same or more restrictive than the visibility of the imported element. If an element ends in having multiple visibilities within a namespace (for example, by being imported several times with different import visibility), public visibility overrides private visibility. For example, if the same element is imported twice into the same namespace, once with public and once with private visibility, it will have public visibility in the importing namespace.

The imported element can be given a new name by an element import — the *alias*. The alias is a name that is given to the imported element within the importing namespace. In other words, the alias becomes the member of the importing namespace instead of the original name of the imported element. For example, in Figure 7-7c, the imported element `Encryption::EncryptedText` can be referred to by its alias `EText` within the importing namespace `UserManagement`. In diagrams, the alias can be specified after or below the keyword `<<import>>` or `<<access>>`.

In case of a name clash of an imported name with an outer name of a named element that is a member of an enclosing namespace, in the importing namespace, the outer name is hidden by the imported name, meaning that the unqualified name refers to the imported name, not to the outer name. The outer name can still be accessed by its qualified name.

If more than one element with the same name would be imported into the same importing namespace, the names of the imported elements must be qualified to be accessed from the importing namespace, and their names are not added to the importing namespace. If the name of the imported element is the same as the name of an element owned by the importing namespace, the name of the imported element must be qualified in order to be used, and is not added to the importing namespace. In general, an imported named element will not be added to the importing namespace if it is not distinguishable from any other owned or imported member of the same importing namespace.

Element import works by reference, meaning that the imported element cannot be modified or extended in the importing namespace.

The alternative notation for element import is textual instead of symbolic. Figure 7-7d shows an example. The element import can be indicated by a text that uniquely identifies the imported element within curly braces either below or after the name of the importing namespace. The textual syntax for public element import is as follows:

```
element-import-spec ::= { element import qualified-name }
element-import-spec ::= { element import qualified-name as alias }
```

The textual syntax for private element import is as follows:

```
element-access-spec ::= { element access qualified-name }
element-access-spec ::= { element access qualified-name as alias }
```

As a result, the *members* of a namespace `N` are all those named elements that can be referred to by the qualified name `Namespace1::Namespace2::...::NamespaceM::N::aName`. These are all members owned

directly by the namespace N , but also the elements imported to that namespace (possibly with aliases). Note that this set is different from the set of elements that can be referred to by their unqualified names from within the namespace N , because that set includes also the members of the enclosing and inherited namespaces.

These are all standard UML rules. A concrete implementation of OOIS UML that relies on an existing programming language may use slightly different rules for accessing imported elements within methods written in that language, for example.

Section Summary

- ❑ An *element import* is a directed relationship between an importing namespace and an imported accessible packageable element. As a result of this relationship, the imported element is added to the importing namespace, so it becomes its *member*, although not directly owned by that namespace. The imported element can be referenced from the importing namespace by its name without qualification.
- ❑ The element import relationship may have its *visibility*, which can be either private or public. If the visibility of the import relationship is public, the imported element becomes a public member of the importing namespace and can be further imported from that namespace to other namespaces. If the relationship has private visibility, the imported element becomes a private member of the importing namespace, and thus it can be used within that namespace without qualification, but it cannot be further imported from that namespace.
- ❑ The imported element can be given a new name by an element import — the *alias*.
- ❑ An imported named element will not be added to the importing namespace if it is not distinguishable from any other owned or imported member of the same importing namespace.

Importing Packages

It is sometimes useful to import many (or even all) public members of a package to another namespace. Instead of importing the elements one by one, the entire package can be imported by another kind of UML relationship, the *package import*.

A package import is a directed relationship between the importing namespace and the imported package (see Figure 7-8a). It may also have its specified visibility, which must be private or public. The semantics of package import are derived from the semantics of element import. A package import of a package P is equivalent to a set of element imports of all public members of the package P , except for the members that are individually imported by explicit element imports, with the visibility of the package import and without aliases. Figure 7-8 shows some examples of package import relationships.

Part III: Concepts

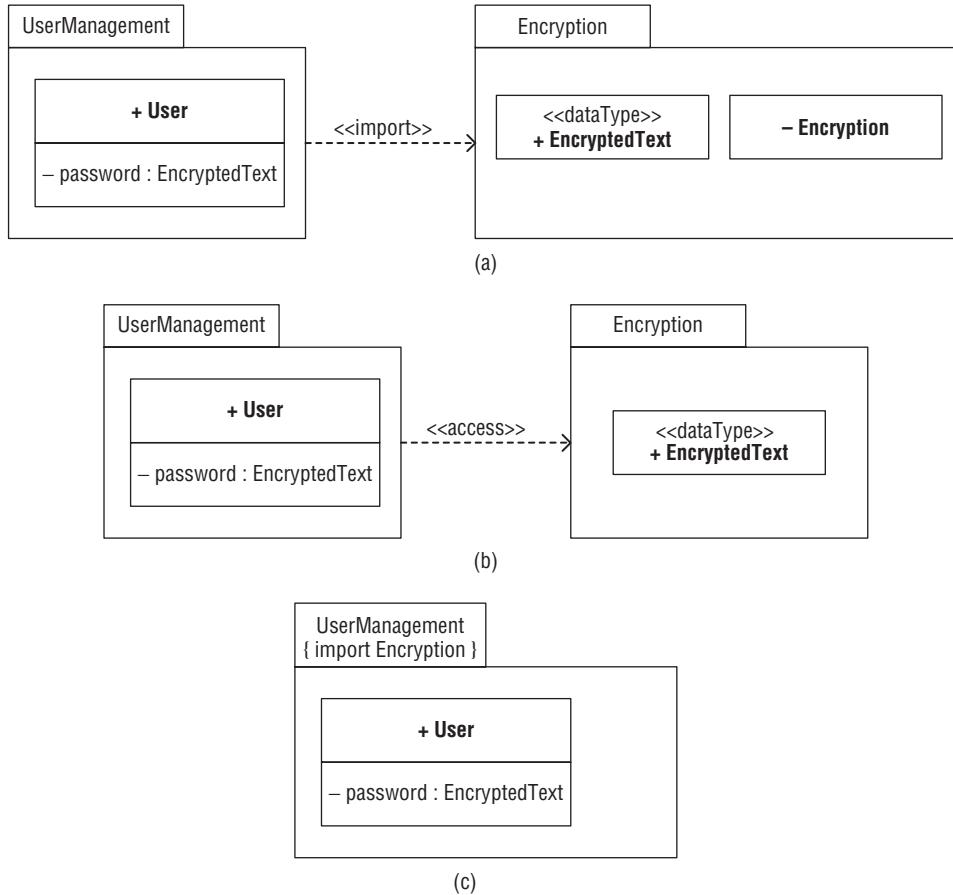


Figure 7-8: Examples of package import. (a) Public package import. The package `UserManagement` imports all public elements of `Encryption`, which allows the elements of `UserManagement` to refer to them by name without qualification. However, they cannot access `Encryption::Encryption` because it is private and not imported. `EncryptedText` can be further imported from `UserManagement`. (b) Private package import. The elements of the imported package `Encryption` cannot be further imported from the package `UserManagement`. (c) Alternative textual notation for package import.

Figure 7-8a shows a public package import. The package `UserManagement` imports all public elements of `Encryption`, which allows the elements of `UserManagement` to refer to them by simple name without qualification. However, they cannot access `Encryption::Encryption` because it is private and not imported. `EncryptedText` can be further imported from `UserManagement`. Figure 7-8b shows a private package import. The elements of the imported package `Encryption` cannot be further imported from the package `UserManagement`.

In UML diagrams, a package import is rendered as a dashed arrow with an open arrowhead from the importing namespace to the imported package (see Figure 7-8a, b). The keyword `<<import>>` for public or

«access» for private package import is shown near the dashed arrow. Alternatively, a textual notation for package import can be used, as shown in Figure 7-8c, with the following syntax for public package import:

```
package-import-spec ::= { import qualified-name }
```

The following syntax is used for private package import.

```
package-access-spec ::= { access qualified-name }
```

Because a package can be imported as a single element by an element import relationship, and also by a package import relationship, to distinguish these two, if the package is imported by an element import, the dashed arrow may be optionally labeled with «element import» or «element access».

Section Summary

- A *package import* is a directed relationship between an importing namespace and an imported package. It may also have its specified visibility, which must be private or public.
- The semantics of package import are derived from the semantics of element import. A package import of a package is equivalent to a set of element imports of all public members of the package, except for the members that are individually imported by explicit element imports, with the visibility of the package import and without aliases.

Dependencies

In a complex model, many of the model elements are tightly coupled and depend on each other in some way, meaning that the definition, implementation, or semantics of one requires or is affected by the other element. For example, the class `MailingList` in Figure 7-9a depends on the class `Message` because its operation `send` has a parameter of the type `Message`. Consequently, the specification and (obviously) the implementation of the class `MailingList` is dependent on the class `Message`: if the latter is changed — for example, if it is removed from the model or if its accessible features (that is, properties and operations) are changed, the former is most probably affected and has to be updated, too.

For many of the UML modeling concepts, such dependencies are implicit. For example, a class is (in some cases) dependent on the class at the other end of an association, an attribute is dependent on its type, an operation having a parameter of a certain type is dependent on that type, and so on. In such cases, the supplier (independent) element must be accessible to the client (dependent) element in its namespace. However, it is sometimes important for the modeler to emphasize and document such a relationship, and draw the reader's attention to it. In addition, there are circumstances in which some model elements are dependent in a way that cannot be inferred from other parts of the model, and the modeler wants to specify these dependencies.

Part III: Concepts

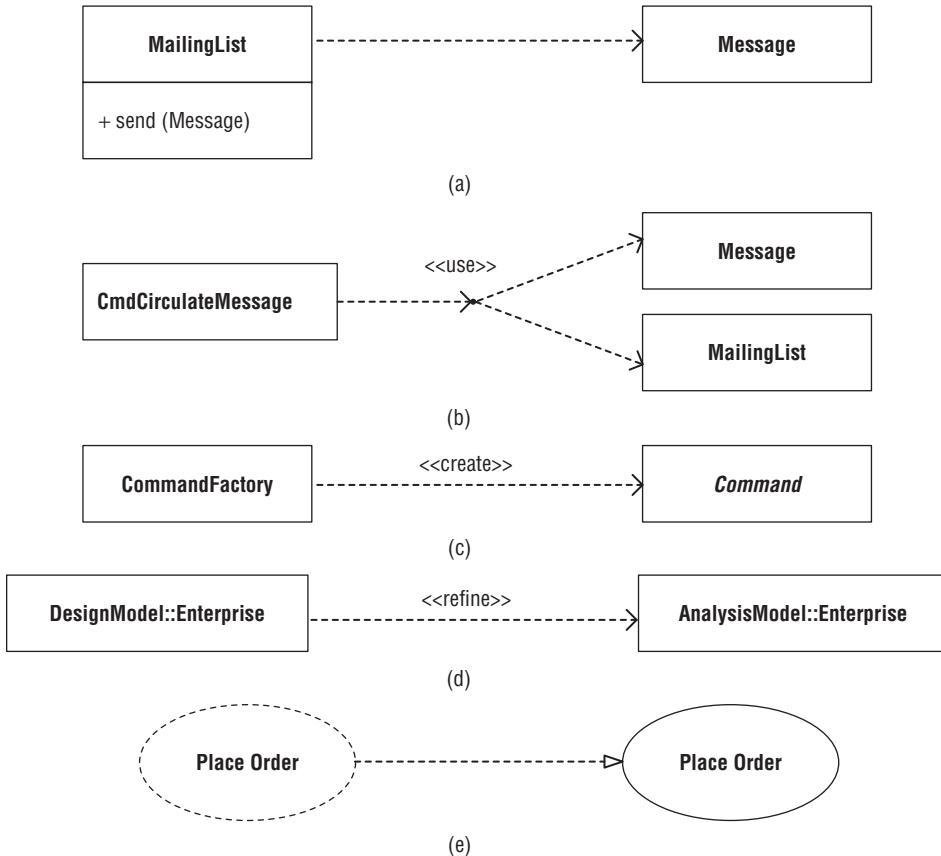


Figure 7-9: Examples of dependency relationships. (a) A general dependency relationship. (b) A one-to-many usage dependency. (c) A <<create>> kind of usage dependency. (d) A <<refine>> kind of abstraction dependency. (e) A realization dependency.

For that purpose, UML offers the *dependency relationship*. Dependency is a relationship between one or more (arbitrarily many) *client* model elements and one or more (arbitrarily many) *supplier* model elements. The relationship indicates that the clients require the suppliers for their specification or implementation. This means that the complete semantics of the client elements are either semantically or structurally dependent on the definition of the supplier elements. Simply stated, a dependency means that a modification or deletion of the supplier model elements may affect the client elements in some way. A client and a supplier can be any named element in the model. A dependency relationship is a packageable element.

It is important to emphasize that a dependency relationship has no specific formal meaning. In particular, it has no implications on the model completeness or correctness. It is not necessary to introduce dependency relationships between any model elements, even between those that have implicit dependencies, because one requires the accessibility of the other. Unlike the import relationship (which may be necessary when one element requires the accessibility of the other), a dependency relationship is completely optional. A model compiler can simply ignore it because it does not carry any formal semantics.

Specialized tools can still exploit some kinds of dependency relationships for specific purposes such as navigation, model assessments, or transformations. However, standard UML does not specify such use. OOIS UML does not exploit this opportunity, either. In OOIS UML, dependencies serve as informal annotations only and are, thus, informal concepts.

Of course, the dependency relationship has no run-time effects either. As you can see, the dependency relationship has predominantly a purpose for documenting the intention of the modeler, and drawing the reader's attention to a certain hidden, implicit, or otherwise significant relationship between model elements that can or cannot be inferred from other parts of the model.

In diagrams, a dependency is shown as a dashed arrow between model elements, as shown in Figure 7-9a. The client model element is at the tail of the arrow; it depends on the supplier model element at the arrowhead. The arrow may be labeled with a tag showing the optional kind of the dependency relationship (given by the word between guillemets << and >>) and an optional name of the relationship. In the case of more than one client and/or supplier elements, one or more arrows with their tails on the clients are connected to the tails of one or more arrows with their heads on the suppliers, as shown in Figure 7-9b. A small dot can be placed on the junction if desired.

A special kind of dependency relationship is the *usage* dependency, where the client element requires the supplier element for its full implementation or operation. For the example in Figure 7-9b, the command class `CmdCirculateMessage` uses the classes `MailingList` and `Message` because it distributes a message to a mailing list. The usage relationship does not specify how exactly the client uses the supplier. It merely represents the fact that the supplier is used by the definition or implementation of the client.

For the preceding example, the method for the operation `execute` of the class `CmdCirculateMessage` sends the given message to the given mailing list, while these two are provided through the properties of the command class. However, all these details are not specified in the usage dependency, and cannot be inferred from it by any means other than by the reader's intuition or a note that can be attached to the dependency. Usually, such a relationship can be inferred from other model elements that are normative specifications (such as the methods of operations or the definitions of the features), but the intention of the modeler is to make this dependency explicit. A usage dependency is shown as a dependency with a `<<use>>` keyword attached to it, as shown in Figure 7-9b.

There are several sub-kinds of the usage dependency that can be used in OOIS UML, including the following:

- ❑ `<<call>>` — Specifies that the client operation or an operation in the client classifier invokes the supplier operation or an operation in the supplier classifier.
- ❑ `<<create>>, <<instantiate>>` — Specify that one or more operations of the client classifier create instances of the supplier classifier (see Figure 7-9c).

Another special kind of the general dependency relationship is the *abstraction* dependency. It relates two elements or sets of elements that represent the same concept at different levels of abstraction, or from different viewpoints. It can (but need not) define a set of expressions that formally or informally specify how the client and supplier element(s) are mapped to each other. The mapping can be unidirectional or bidirectional. For example, the client element(s) can be computed from the supplier elements by a formal or informal derivation rule, or can be obtained by an automatic or manual model transformation specified in the mapping. An abstraction relationship is shown as a dependency with an

Part III: Concepts

«abstraction» keyword attached to it, or with one of the specific predefined names that may be the following:

- ❑ «derive» — Signifies that the client element(s) can be computed (derived) from the supplier element(s). The related model elements are usually (but not necessarily) of the same kind. The mapping, if present, specifies the computation (derivation rule). The client may be implemented for design reasons (such as efficiency), even though it is logically redundant.
- ❑ «refine» — Specifies a refinement relationship between model elements at different levels of detail (such as analysis and design, as shown in Figure 7-9d). The mapping may or may not be computable, and may be unidirectional or bidirectional. It can be used to model transformations from analysis to design and similar.
- ❑ «trace» — Specifies a trace relationship between model elements that represent the same concept in different models. These are mainly used for tracking requirements and changes across models. Because model changes can occur in both directions, the directionality of the dependency can often be ignored. The mapping specifies the relationship between the two, but is usually informal and not computable.

Finally, there is another special kind of the dependency relationship — the *realization relationship*. It is a specialized abstraction relationship where the suppliers represent a specification and the clients represent an implementation of the suppliers. The meaning of “implementation” is not strictly defined, but rather implies a more refined or elaborate form with respect to a certain modeling context. It is shown as a dashed line with a triangular arrowhead pointing to the realized element (the supplier), as shown in Figure 7-9e.

Section Summary

- ❑ *Dependency* is a relationship between a non-empty set of *client* elements and a non-empty set of *supplier* elements. It signifies that the clients require the suppliers for their specification or implementation, or are otherwise affected by a modification of the supplier model elements.
- ❑ The dependency relationship has no formal semantics. In OOIS UML, dependencies are solely informal annotations that are used to draw the reader’s attention to some significant relationships between model elements that can or cannot be inferred from other parts of the model.
- ❑ *Usage* is a specialized dependency relationship in which the client elements require the supplier elements for their full implementation or operation. Examples include call and create/instantiate dependencies.
- ❑ *Abstraction* is a specialized dependency relationship that relates two sets of elements that represent the same concept at different levels of abstraction or from different viewpoints. Examples include derivation, refinement, or tracing relationships.
- ❑ *Realization* is a specialized abstraction relationship where the suppliers represent a specification, and the clients represent an implementation of the suppliers.

Multiplicity Elements

As explained in Chapter 5, association ends have their multiplicity specifications, which define the allowable cardinalities of collections of objects linked at the corresponding slots. In UML, however, several other kinds of model elements can have multiplicity specifications, too. Attributes and parameters of operations also have multiplicities. This means that attributes and parameters can also be instantiated in multi-valued collections instead of single values. This language feature has its origin in traditional programming languages and represents a generalization of the concepts of multi-valued arrays or collection types.

Figure 7-10 shows some examples. The attribute `Person::title` of type `Text` has multiplicity `1..*`, meaning that each object of `Person` owns a collection of values referred to as its `title` attribute. The collection always has at least one element, and may have an arbitrary number of elements. The purpose of this multi-valued attribute is to provide a means to address a person by an array of titles (for example, "Prof. Dr." or "His Excellency Mr." and similar).

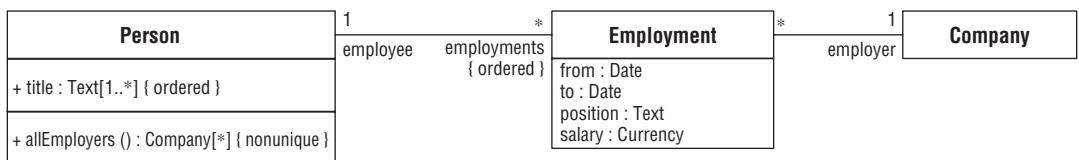


Figure 7-10: Multiplicity specifications for attributes, parameters of operations, and association ends

Similarly, a parameter of an operation has its multiplicity. In Figure 7-10, the return parameter of the operation `Person::allEmployees` has multiplicity `*`, meaning that the operation returns a collection with an arbitrary number of components. The operation returns the collection of all companies in which the given person has been employed.

In general, *multiplicity* is a definition of an inclusive interval of non-negative integers beginning with a lower bound and ending with an upper bound; the upper bound can be infinite. It specifies the range of allowable *cardinalities* of the collections of values that represent instantiations of the element to which the multiplicity belongs. Therefore, multiplicity is a (static) part of the model, and exists at design time. When an element to which the multiplicity belongs is instantiated in a collection of values, at run-time, the instantiation contains a certain number of components at each moment. That number (which thus relates to the state of the instantiation during an interval of run-time) is referred to as the *cardinality* of the collection. A multiplicity specification of an element constrains the set of allowable cardinalities of the values when the element is instantiated.

Mathematically, if C is a non-negative integer that represents a cardinality of a certain collection, the multiplicity M represents a set of cardinalities and *allows* C if and only if $C \in M$. In other words, C is a valid cardinality for M if and only if $C \in M$. One multiplicity X is said to *include* another multiplicity Y if and only if the set of allowable cardinalities Y is a subset of the set X . For example, the multiplicity `1..*` includes the multiplicity `1..7`, but does not include the multiplicity `0..*`. A multiplicity is called *multi-valued* if it allows at least one cardinality greater than one; otherwise, it is called *single-valued*. The multiplicities `1..1` and `0..1` are single-valued; all others are multi-valued.

A multiplicity is specified with its lower and upper bounds. Both must be constant expressions that can be evaluated at compile time and must result in non-negative integers. In other words, the lower and upper bounds can be non-negative integer literals or arithmetic expressions computable at compile time, which

Part III: Concepts

must evaluate to non-negative integers, and which must not have side-effects on the model or on the object space. The upper bound may be a special symbol for unbounded integers — the asterisk (*). Typically, the range of cardinalities specified by a multiplicity is not empty (that is, the multiplicity specifies at least one valid cardinality greater than zero). For example, the multiplicity 5..2 is invalid. However, UML allows a special case of 0..0 multiplicity that has a specific and limited practical use for restricting the allowed range of cardinalities for redefined properties that will be described later in this book.

A multiplicity is shown in the following syntax:

```
multiplicity ::= lower-bound .. upper-bound  
multiplicity ::= upper-bound
```

Here, *upper-bound* can be * for unlimited natural. If the multiplicity specification is part of a textual notation of an element to which it belongs, it is enclosed in square brackets (see Figure 7-10 for attributes and parameters). If it belongs to an element that is represented as a symbol, the multiplicity is not enclosed in brackets, and is shown near the symbol of the element (see Figure 7-10 for association ends). If the lower bound is equal to the upper bound, only one of them can be shown. For example, 1 is equivalent to 1..1. The multiplicity 0..* can be also displayed as *.

All kinds of elements in UML that can have multiplicity are called *multiplicity elements*. Multiplicity element is, thus, an abstract generalization of all such concepts of the language, like properties or parameters. A multiplicity element can be instantiated at run-time, and the multiplicity of the model element constrains the set of allowable cardinalities of an instantiation of the element.

In OOIS UML, the default multiplicity for association ends is * (arbitrarily many), for attributes is 1..1 (exactly one), and for parameters is 0..1 (zero to one).³

An instantiation of a multi-valued multiplicity element represents a *collection* of values, whereby the ordering of the elements in the collection is not determined by default. This means that no assumptions can be made about the order in which the values occur in the collection at a certain moment when the collection is accessed by an action. The order may even vary during time, according to the underlying implementation. In other words, the ordering is under the control of the implementation and is thus considered unpredictable (or arbitrary) in the application.

However, this is not always what is needed. In some cases, the modeler may need to assume that the values in an instantiation of a multiplicity element occur in a predictable sequential order, which can be controlled by the application. For example, the attribute Person::title in the model in Figure 7-10 should be ordered because it is important that there be a determined arrangement of the titles of a person, so that a person can be addressed in a strictly defined way (for example, “Prof. Dr.” consists of two ordered titles, “Prof.” and “Dr.”). Similarly, the property employments of an object of Person results in a collection of objects of Employment; the resulting collection is ordered, for example, by the chronology of jobs of the person.

³This rule is not part of standard UML — it is an extension of the OOIS UML profile. In standard UML, multiplicity elements do not have default multiplicities in the model, although they have notation-wise defaults: a missing multiplicity in a diagram defaults to 1..1.

Mathematically, ordering means that there is a mapping of a set of successive positive integers, beginning with 1 and ending with some natural number, to the collection of values that represents an instantiation of a multi-valued multiplicity element. Note, however, that the language does not specify the concrete mapping. In other words, the order of the values is not defined. There is no implicit sorting criterion. The type of the values does not have to define any ordering relationship for its instances either. Ordering simply means that the sequential arrangement of the values in a collection is preserved between different actions that operate on those values. Consider the preceding example — ordered titles of a person — the values in the collection always remain in an order determined by the effects of the actions that have manipulated with those values.

As a result, a multiplicity element can be ordered or unordered. It is unordered by default. If a multiplicity element is not multi-valued, ordering does not have any semantic impact. In general, this property of a multiplicity element is denoted with the keywords `ordered` or `unordered` within curly braces next to the textual notation for the element, or near the symbol of the element, as shown in Figure 7-10.

In a similar way, a multiplicity element can be either *unique* or *non-unique*. If a multiplicity element is unique, then each instantiation (as a collection of values) must not have duplicate values (that is, it represents a set). If it is non-unique, on the other hand, it can contain duplicates. For example, the operation `Person::allEmployers` returns a collection of (references to) objects of `Company` in which the person has been employed. Because the same person could have been employed in the same company several times during different periods, the return parameter of this operation is non-unique.

In OOIS UML, an association end is unique by default, and an attribute or parameter is non-unique by default. If a multiplicity element is not multi-valued, uniqueness does not have any semantic impact. In general, this property of a multiplicity element is denoted with the keywords `unique` or `nonunique` within curly braces next to the textual notation for the element, or near the symbol of the element, as shown in Figure 7-10.

In summary, a multiplicity element can have four combinations of unique and ordered properties. As a result of each combination, the collection of values represented by its instantiation has the type as follows:

Is Ordered?	Is Unique?	Collection Type
No	Yes	Set
Yes	Yes	Ordered set
No	No	Bag
Yes	No	Sequence

The multiplicity of a multiplicity element has derived run-time semantics in OOIS UML. It represents a constraint implicitly attached to the multiplicity element. Therefore, it is checked at run-time as all other constraints attached to the same element. For example, the multiplicity of each association end is checked at both link ends whenever a link is created or destroyed. Uniqueness and ordering affect the semantics of actions on multiplicity elements, as will be described later for these actions.

Section Summary

- ❑ *Multiplicity* is a definition of an inclusive interval of non-negative integers beginning with a lower bound and ending with a upper bound (can be unbounded, too). It specifies the range of allowable *cardinalities* of the collections of values that represent instantiations of the element to which the multiplicity belongs.
- ❑ A model element with a specified multiplicity is called a *multiplicity element*.
- ❑ A multiplicity element can be *ordered*, which means that there is a mapping of the set of successive positive integers starting from 1 to the collection of values that represents an instantiation of the multiplicity element.
- ❑ A multiplicity element can be *unique*, meaning that each of its instantiations, as a collection of values, must not have duplicate values.
- ❑ The multiplicity of a multiplicity element has derived run-time semantics in OOIS UML. It represents a constraint implicitly attached to the multiplicity element. Uniqueness and ordering affect the semantics of actions upon multiplicity elements.

Variables

For an illustration of one particular kind of multiplicity elements and how they can be manipulated, let's take a look at the notion of *variables*. Although they are related to methods, and will be described in more detail later in this chapter along with methods in OOIS UML, they are introduced here because the sample code snippets throughout the book will often include variables and actions with them.

Within methods, the results of actions can be stored in local *variables*, which serve as local temporary storage for passing data between actions indirectly. The values of variables are local to the methods they belong to, and cannot be accessed from other contexts, although they can refer to objects of the common object space.

In accordance with modern OO programming languages, OOIS UML variables always store *references* to instances of classifiers (classes or data types), thus representing intermediaries to those instances, allowing substitution. For example, in the OOIS UML native detail-level language, this may look like the following:

```
Integer i = new Integer;  
Person p = new Teacher;
```

In this example, the variable named `i` refers to a newly created instance of the data type `Integer`, created by the action “Create Data Type Instance” denoted with `new` in the language. The variable `p` is declared to be of type `Person`, but it actually refers to a newly created object of class `Teacher` because the substitution takes place here.

Variables, just as some other concepts of UML, fall into the category of *typed elements*, which means that they have a type. The type of a typed element is the classifier that constrains the values represented by the typed element by defining the set of those values. For example, an attribute is a typed element, whose

type is a data type; an association end is a typed element, whose type is the class at that end; a parameter of an operation is a typed element, whose type is a class or a data type; a variable is a typed element, whose type is a class or a data type, and so on.

The aforementioned examples of variables are single-valued variables, storing at most one reference to a type instance. However, the concept of variable in UML is generalized to a multiplicity element, so that a variable has multiplicity, ordering, and uniqueness specifications, similar to attributes, association ends, or parameters of operations. Therefore, variables can be multi-valued. Following is an example in the OOIS UML native detail-level language:

```
Car[0..3]{unique,unordered} aSmallParking = { aCar1, aCar2, new Car };
Vehicle[*]{unique,unordered} aHugeParking;
```

Parameters of operations are seen and used as local variables in methods.

Note that instantiations of variables, regardless of whether they are single-valued or multi-valued, are *not* instances of classifiers. They always store *references* to such instances. If a variable is multi-valued, its instantiation represents a collection of references, for which additional constraints apply (multiplicity and uniqueness).

In OOIS UML, a variable is introduced by its declaration. Each variable must be declared before it is used. A declaration of a variable introduces the variable into the current namespace and optionally initializes its value as shown in the given examples.

If the multiplicity is not specified, the default multiplicity for variables is 0..1. If ordering or uniqueness is not specified, the default settings are ordered and nonunique.⁴ For example, both of the following variables are of multiplicity 0..1:

```
Integer i = ...
Car[0..1] c = ...
```

A variable with multiplicity 0..1 may be empty, meaning that it has no value (that is, its cardinality is zero). A shorthand for the state of having no value is the special symbol `null`, as shown in the following example:

```
Car[0..1] c = null;
if (c==null) ...;
```

In OOIS UML, actions upon variables are invoked using the arrow notation (`var->action(...)`), instead of the dot notation (`value.operation(...)`) because the dot operator is used when its left operand represents a single value, which is a reference to an instance of a class or a data type. The actions (after `->`) work on the variable itself, which can represent a collection of values at run-time. Following is an example:

```
Car aCar = ...
Vehicle[*]{unique,unordered} aHugeParking;
aHugeParking->add(aCar);
```

⁴This rule is not part of standard UML — it is an extension of the OOIS UML profile. In standard UML, multiplicity elements do not have default model-wise multiplicities, although they have notation-wise defaults: a missing multiplicity defaults to 1..1.

Part III: Concepts

The last line of this code fragment performs the action that adds the value of the variable `aCar` to the variable `aHugeParking` (that is, to that collection of references).

The dot (.) operator, on the other hand, is used to access a member of the object or data type instance that is the value of the variable. For example, the following code invokes an operation `move` of the object referred to by the single-valued variable `aCar`:

```
aCar.move(...);
```

Similarly, the `forEach` construct iterates through all values in the given variable:

```
Car myCar = ...;
Vehicle[*]{unique,unordered} aHugeParking = ...
aHugeParking->forEach(c) {
    if (c==myCar) ... // Is this my car?
    ...
}
```

Section Summary

- ❑ A *variable* is a local temporary storage for passing data between actions indirectly. Variables serve to store results of actions within methods.
- ❑ A variable is a *typed element*. A typed element has a type, which is the classifier that constrains the values represented by the typed element by defining the set of those values.
- ❑ A variable is a multiplicity element, meaning that it has multiplicity, ordering, and uniqueness specifications.
- ❑ At one moment in run-time, an instantiation of a variable stores zero or more references to instances of data types or classes. The cardinality and uniqueness of the collection of references conforms to the specification of the variable as a multiplicity element. The type of each referenced instance must conform to the type of the variable (substitution is assumed).

Conformance Rules

The substitution rule is, as already explained, a fundamental principle of the object paradigm. Whenever and wherever an instance of a generalizing classifier is expected, an instance of a specializing classifier can occur. In other words, instances of a specializing classifier can substitute instances of a generalizing classifier. The opposite does not hold, of course. This rule is an axiom of virtually any statically typed OO language.

Of course, the substitution rule holds for typed elements:

- ❑ An instance of a type specializing the type of an attribute can occur as the value of the attribute. For example, if `TextFile` is a type specializing `File`, then an instance of `TextFile` can occur as

the value of the attribute of type `File`. This is reasonable because `TextFile` is a kind of `File`, since everything that holds for a `File` holds also for a `TextFile`, and everything that can be done with a `File` can also be done with a `TextFile`. In terms of actions written in the OOIS UML native detail-level language, this may look like the following:

```
Person p = ...           // p refers to an instance of Person
TextFile tf = ...         // tf refers to an instance of TextFile
p.contract = tf;          // An instance of TextFile is written as the value
                         // of the attribute contract of type File
File f = p.contract;     // f actually refers to a TextFile
```

Of course, even though the actual value of the attribute `contract` is an instance of `TextFile`, only the features of `File` are accessible over the attribute. This is because the attribute is of type `File`, thus allowing any (direct or indirect) instance of that type to be its value.

- An instance of a type specializing the type of an association end can occur as the instance linked at a slot of that end. For example, if `Teacher` specializes `Person`, an instance of `Teacher` can occur as the student linked to a `Course`. This is again reasonable because `Teacher` is a kind of `Person`, because everything that holds for a `Person` holds also for a `Teacher`, and because everything that can be done with a `Person` can be also done with a `Teacher`. In terms of actions written in the OOIS UML native detail-level language, this may look like the following:

```
Course c = ...           // c refers to an instance of Course
Teacher t = ...           // t refers to an instance of Teacher
c.students->add(t);      // An instance of Teacher is linked to the
                         // slot 'students' of type Person
```

- An instance of a type specializing the type of an operation parameter can occur as the actual argument in an invocation of the operation.
- A variable of a generalizing type can refer to an instance of a specializing type:

```
File f = new TextFile;
```

On the other hand, all the mentioned typed elements are also multiplicity elements, meaning that they have multiplicity specifications that potentially allow multi-valued instantiations of those elements. What about the substitution rules with regard to multiplicity? What if two multiplicity elements have conformant (that is, substitutable) types, but have different multiplicities?

The answer to this question is easy to deduce if the substitution rule regarding typed elements is considered as follows. A specializing type (classifier) *inherits* and *extends* the features of a generalizing type, with the consequence that the set of instances of the specializing type is a subset of the set of instances of the generalizing type. Consequently, all that holds for the instances of the generalizing type must also hold for the instances of the specializing type. In other words, the specializing type may just enhance or add some rules or constraints that hold for the generalization, thus possibly narrowing the set of instances that satisfy those rules and constraints. It cannot relax, remove, or contradict the rules and constraints on its instances, because this would possibly broaden the set of instances.

Multiplicity is actually a constraint associated with a multiplicity element. Consequently, a multiplicity M_1 is stricter than a multiplicity M_2 only if M_2 includes M_1 ; M_1 defines a subset of possible cardinalities included in M_2 .

Part III: Concepts

As a result, the common notion of *conformance* is introduced to define the necessary and sufficient conditions for “mixing” typed multiplicity elements. Informally, a type T_1 conforms to a type T_2 if T_1 is a direct or indirect subtype (that is, specialization) of T_2 . A multiplicity M_1 conforms to a multiplicity M_2 if M_2 includes M_1 . A typed multiplicity element E_1 conforms to a typed multiplicity element E_2 if the type and multiplicity of E_1 conform to the type and multiplicity of E_2 , respectively.

The ordering and uniqueness properties fit into this frame in a similar way. They should be considered as additional constraints to the multiplicity elements. In order to conform to another multiplicity element, one multiplicity element may just add constraints, not remove or contradict them.

More formally, the rules for conformance of types are as follows:

- Specialization** — If a type T_1 is a specialization of a type T_2 , then T_1 conforms to T_2 .
- Reflexivity** — Each type conforms to itself.
- Anti-symmetry** — If a type T_1 conforms to a type T_2 , and T_2 conforms to T_1 , then T_1 , and T_2 are the same. In other words, two different types cannot conform to each other at the same time.
- Transitivity** — If a type T_1 conforms to a type T_2 , and T_2 conforms to a type T_3 , then T_1 also conforms to T_3 .

The rule for conformance of multiplicity elements is as follows. A multiplicity element M_1 conforms to another multiplicity element M_2 if and only if all of the following conditions are satisfied:

- The multiplicity of M_2 includes the multiplicity of M_1 .⁵
- The ordering properties of M_1 and M_2 are the same (both are either ordered or unordered), or M_1 is ordered and M_2 is unordered.
- The uniqueness properties of M_1 and M_2 are the same (both are either unique or non-unique), or M_1 is unique and M_2 is non-unique.

Consequently, a multiplicity element always conforms to itself.

Finally, a typed multiplicity element E_1 conforms to a typed multiplicity element E_2 if and only if the type and multiplicity of E_1 conform to the type and multiplicity of E_2 , respectively.

The following examples illustrate these rules. Let’s assume that a type T_1 conforms to T_2 . Then:

- $T_1[*]\{\text{ordered}, \text{unique}\}$ conforms to $T_1[*]\{\text{unordered}, \text{unique}\}$
- $T_1[1]\{\text{ordered}, \text{unique}\}$ conforms to $T_1[1]\{\text{ordered}, \text{nonunique}\}$
- $T_1[1]\{\text{ordered}, \text{unique}\}$ conforms to $T_2[1]\{\text{unordered}, \text{nonunique}\}$
- $T_1[0..1]\{\text{ordered}, \text{unique}\}$ conforms to $T_2[*]\{\text{ordered}, \text{unique}\}$
- $T_1[*]\{\text{ordered}, \text{nonunique}\}$ does not conform to $T_2[*]\{\text{ordered}, \text{unique}\}$ because $\{\text{nonunique}\}$ does not conform to $\{\text{unique}\}$

⁵It is assumed that if two multiplicities are equal, each one includes the other (that is, a multiplicity includes itself).

- ❑ $T1[0..1]\{\text{ordered}, \text{unique}\}$ does not conform to $T2[1]\{\text{ordered}, \text{unique}\}$ because the multiplicity $0..1$ does not conform to the multiplicity $1..1$
- ❑ $T2[0..1]\{\text{ordered}, \text{unique}\}$ does not conform to $T1[*]\{\text{ordered}, \text{unique}\}$ because $T2$ does not conform to $T1$.

As an illustration of how this rule works in the OOD UML native detail-level language, a typed multiplicity element $T1$ can be *assigned* to a typed multiplicity element $T2$ if and only if $T1$ conforms to $T2$.⁶ Assignment of typed multiplicity elements basically copies the collection of references from the incarnation of the source multiplicity element to the incarnation of the destination multiplicity element one by one. If both the source and the target multiplicity elements are ordered, the order of the source incarnation is preserved in the target. The conformance rule guarantees that the multiplicity and uniqueness constraints will be satisfied for the destination element after the assignment.

For example, $\text{Car}[0..3]\{\text{unique}, \text{unordered}\}$ conforms to $\text{Vehicle}[*]\{\text{unique}, \text{unordered}\}$ because Car is a specialization of Vehicle , and thus, the former can be assigned to the latter. This may look like the following:

```
Car[0..3]{unique,unordered} aSmallParking = ...;  
Vehicle[*]{unique,unordered} aHugeParking = aSmallParking;
```

In this example, within the initialization of $aHugeParking$, the collection $aSmallParking$ of references to Car is copied to the collection of references to Vehicle . This is correct because the target references will still refer to vehicles, in fact, to their specializations — cars. The multiplicity constraint of the target will be satisfied because the size of the small parking “fits” into the size of the huge parking, and their uniqueness and ordering also conform (that is, are the same here).

The same holds for parameters of operations. When an operation is invoked, the actual argument (as a typed multiplicity element) is assigned to the corresponding formal parameter, which behaves like a variable local to the invoked method. This may take place only if the actual argument conforms to the formal parameter. Assignment of a typed multiplicity element to a property (attribute or association end) has slightly more complex semantics that will be described later.

Section Summary

- ❑ Following are type conformance rules:
 - ❑ **Specialization** — If a type $T1$ is a specialization of a type $T2$, then $T1$ conforms to $T2$.
 - ❑ **Reflexivity** — Each type conforms to itself.
 - ❑ **Anti-symmetry** — If a type $T1$ conforms to a type $T2$, and $T2$ conforms to $T1$, then $T1$ and $T2$ are the same type. In other words, two different types cannot conform to each other at the same time.

Continued

⁶There is an exception to this rule in relation to the constrained action groups. It will be discussed later in this book.

Part III: Concepts

- Transitivity** — If a type T_1 conforms to a type T_2 , and T_2 conforms to a type T_3 , then T_1 also conforms to T_3 .
- A multiplicity element M_1 conforms to another multiplicity element M_2 if and only if all of the following conditions are satisfied (a multiplicity element always conforms to itself):
 - The multiplicity of M_2 includes the multiplicity of M_1 .
 - The ordering properties of M_1 and M_2 are the same (both are either ordered or unordered), or M_1 is ordered and M_2 is unordered.
 - The uniqueness properties of M_1 and M_2 are the same (both are either unique or non-unique), or M_1 is unique and M_2 is non-unique.
- A typed multiplicity element E_1 conforms to a typed multiplicity element E_2 if and only if the type and multiplicity of E_1 conform to the type and multiplicity of E_2 , respectively.

8

Classes and Data Types

This chapter examines the details related to classes and data types, as well as to their instances. First, the common and the discriminating characteristics of classes and data types are discussed. Then, creation and destruction of their instances is examined. Finally, some specific properties of data types are considered.

Common Characteristics of Classes and Data Types

Classes and data types are classifiers in UML and, thus, have many things in common. This section examines the common characteristics of classes and data types.

Notions of Class and Data Type

Classes and data types are language concepts for modeling abstractions from the problem domain. Abstraction is one of the basic instruments for prevailing against inherent complexity of a problem domain. It assumes that the modeler reveals commonalities of entities from the domain, and neglects those particularities that are not relevant for the software solution. This way, particular entities from the problem domain can be *classified* into categories of entities sharing common features and characteristics. This allows the software implementation to treat them in a unique way. Consequently, the infinite diversity of entities from the real world becomes tractable by treating it as a finite set of classifiers with possibly an infinite number of elements.

Abstraction can be used to classify tangible things from the real world (such as persons, vehicles, apartments, inventory, and so on). It may also represent roles that tangible entities play in the context of the problem (such as employees, library members, patients, clients, callers, recipients, and so on). Similarly, it may represent purely conceptual things that may or may not appear in tangible (usually electronic) forms, such as orders, bank accounts, medical records, e-mail messages, telephone calls, tasks, integers, strings, dates, and so on.

Part III: Concepts

Instead of dealing differently with each particular instantiation of such a concept, the software solution can treat each member of one classifier in the same way, allowing a certain degree of their diversity through slots held by each of the instantiations. What is and what is not going to be modeled with a classifier in the software solution is a matter of conceptual modeling and not of the language itself.

This section focuses only on the semantics of the language concepts, and the last part of this book will give some hints on how these concepts could be used. However, the way a concept should be used depends heavily on its semantics, and this is why the semantics must be completely clear and precisely defined from the beginning.

However, classes and data types do not only serve to model abstractions from the problem domain. They also can be used to model abstractions from the domain of the very solution. During the design of the software system, modelers often invent or discover suitable abstractions that can provide a proper solution to the user's requirements. Those abstractions do not need to be directly visible to users as classifiers whose instances the user can explicitly manipulate at run-time. Instead, those abstractions can be instantiated in the back-end of the application, or have implicit manifestation in the system's appearance and behavior. However, those abstractions can be modeled using the same language concepts of classes and data types because they have the same semantic interpretation.

For example, commands, as described in Chapter 6, fall into this category. A command is not part of the problem domain, but it is an abstraction suitable for generalizing a user's interaction requests and is responsible for their accomplishment. Complex software systems often incorporate many such abstractions.

Classes and data types are classifiers, meaning that they classify instances into sets, according to their common features and constraints that apply to them. Therefore, classes and data types can be seen as sets of their instances. Classifiers denote sets in terms of the criteria of membership, which are their features and constraints. Every instance of a classifier has all those features that are specified in the classifiers to which it belongs. Similarly, all constraints that are specified for a classifier must hold for all its (direct or indirect) instances. In other words, an instance belongs to a set denoted by the classifier if and only if the instance has all features and satisfies all constraints specified for the classifier.

Such an interpretation turns abstract notions from the problem domain into concrete conceptual things from the domain of the particular software solution (that is, into the model and subsequent implementation). For example, a person is a pretty abstract concept, and it might be a very complex ontological problem to define the notion of a person in the real world. Fortunately, however, this is not necessary when software is being constructed. Instead, the modeler must reveal or invent the appropriate set of features of the proper classifier that will meet the business requirements, and nothing more.

Although this is a much easier task than ontologically defining the real-world concept of person, it still might be a difficult problem. This is again a matter of conceptual modeling, to which a later part of this book is devoted. As a result, the physical software implementation of the problem domain will often represent a very dramatic simplification of the real world, but no one will mind this if it meets the requirements. In this simplification, instances of classes and data types will be represented in a very concrete manner, through bits and bytes of the computer memory, and will be managed by the system.

Finally, classes and data types (as with almost all other modeling concepts) can be understood in yet another way. They have emerged through evolution of some other earlier programming concepts, whereby the evolution was driven by the application of those earlier concepts on real problems, and by creativity of software engineering visionaries. Over time, those concepts have been improved and clarified. Once they have been proven in practice as useful for solving many practical problems, and their semantics have been clearly defined, they are offered to the wide software development community as a language concept available for modeling software. Hence, software modelers should simply try to understand their meaning, get trained with how they could be used, and finally adopt them as available tools for designing software.

In other words, the modeler has a clear task: “Here is what you have as your tools, here is the problem — solve it. You must shape the entire problem in the form of the programming concepts available to you.” Although such an approach may sometimes be dangerous because it may limit creativity (“When you have a hammer as your only tool, every problem begins to look like a nail”), in most cases, it is very useful because it makes the developer’s life easier and the solution tractable. After all, in a somewhat simplified sense, software development is all about finding proper concepts from your software development toolbox to solve the specified problem.

Section Summary

- Classes and data types are language concepts for modeling abstractions from the problem domain, as well as from the solution domain.
- Classes and data types are classifiers, meaning that they represent sets of instances grouped according to their common features and constraints.

Classes and Data Types as Classifiers

In UML, classes and data types are both *classifiers*, meaning that they denote sets of *instances* according to their common *features*. Features are characteristics of classifiers and their instances, and can be *structural* (for example, properties) or *behavioral* (for example, operations). Structural features of a classifier specify slots carrying values in each instance of the classifier. Behavioral features describe aspects of behavior of a classifier’s instances. Therefore, each instance of a classifier has all features specified in the classifier.

Classifiers are packageable elements, meaning that they can be (and usually are) directly owned by packages. In practice, classifiers should be grouped into packages according to the general rule of cohesive and loosely coupled packages’ contents.

Classifiers are also namespaces, whose members are their features. In other words, features of a classifier belong to the classifier as a namespace. The already described naming and visibility rules for namespaces and their members apply equally to classes and data types and their features.

Classifiers are types, meaning that they can serve as types of typed elements (for example, variables or parameters). Additionally, type conformance rules apply equally to both classes and data types. Briefly, a specializing classifier conforms to a generalizing classifier.

Part III: Concepts

Finally, both classes and data types have the same notation in UML. That is actually the default notation for all classifiers, while some other specific kinds of classifiers have their own distinct notation. Figure 8-1 shows some examples. The default notation for a classifier is a solid-outline rectangle containing the classifier's name, with optional compartments separated by horizontal lines containing features or other members of the classifier. The specific kind of the classifier can be shown in guillemets above the name — for example, `<<class>>` (default) or `<<dataType>>`, as in Figure 8-1. The keyword in guillemets and the classifier name are usually centered, and the classifier name is usually printed in boldface.

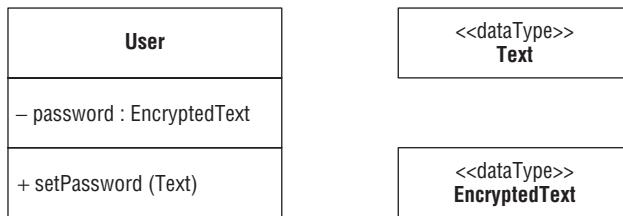


Figure 8-1: Default notation for classifiers

Any compartment with features may be suppressed. A separator line is not drawn for a suppressed compartment. Nothing can be concluded about the presence or absence of elements in the suppressed compartment. For example, the absence of the attribute compartment in the symbol for the class `EncryptedText` in Figure 8-1 does not mean that the classifier does not have any attribute, but just that the attributes (if any) are not shown in the diagram. Compartment names can be used to remove ambiguity (for example, a compartment can be entitled `attributes` or `operations`). Usually, compartments are shown in some diagrams (for example, where the class is emphasized with all its details), and suppressed in other diagrams (for example, where the class is related with other classes).

The name of an abstract classifier is shown in italics. Optionally, the keyword `{abstract}` can be written after or below the classifier's name.

Section Summary

- ❑ Classes and data types are *classifiers*, meaning that they denote sets of *instances* according to their common *features*. Features are characteristics of classifiers and their instances, and can be *structural* (properties) or *behavioral* (operations).
- ❑ Classifiers are packageable elements, meaning that they can be (and usually are) directly owned by packages.
- ❑ Classifiers are namespaces whose members are their features.
- ❑ Classifiers are types, meaning that they can serve as types of typed elements (for example, of variables or parameters).
- ❑ The default notation for a classifier is a solid-outline rectangle containing the classifier's name, with optional compartments separated by horizontal lines containing features or other members of the classifier.

Discriminating Characteristics of Classes and Data Types

The described characteristics of classifiers make classes and data types similar modeling concepts. Of course, these two concepts would not exist separately if they did not have characteristics that distinguish them. Some of those characteristics are related to modeling constraints (applicable at design time), while others affect their run-time semantics. Those distinguishing characteristics also dictate their usage in models.

In brief, the following are the distinguishing characteristics of classes and data types:

- ❑ **Identity** — Instances of classes (that is, objects) have identity, which is independent of their attribute values. Instances of data types do not have identity. They are pure values.
- ❑ **Features** — Classes may have attributes and operations, and may take part in associations. Data types can have only attributes and operations. They cannot take part in associations.¹ Moreover, operations of data types should not have side effects (that is, they do not modify the state of the instances), but should be pure functions.
- ❑ **Lifetime** — The lifetime of instances of classes (that is, objects) is explicit. They are explicitly created and destroyed by actions. The lifetime of instances of data types is implicit. They are implicitly destroyed when they are not needed (that is, when they are not referenced anymore).²
- ❑ **Copy semantics** — Classes do not have default copy semantics, but the modeler must implement copy operations if needed. Data types have implicit, default copy semantics, which are the deep (propagated) copy semantics.³

To distinguish the instances of classes and data types, the former are also referred to as *objects*, whereas the latter are also referred to as *data values*. The rest of this chapter provides detailed information on these characteristics.

Identity

This section explores the identity of class instances and data type instances.

Class Instances

Every instance of a class has its own *identity*, which is an inherent characteristic of that instance that distinguishes it from any other instance of the same or any other class. The identity of an object is inherent, meaning that it is ensured by the very existence of the object, and does not require any special activity by the modeler or user at modeling or run time.

For example, one instance of the class `Person` is, by default, different and *can* be distinguished from any other instance of the same class or any other class. *How* it can be distinguished is a matter of the concrete

¹This distinction is not part of standard UML — it is an extension of the OOIS UML profile.

²This distinction is not part of standard UML — it is an extension of the OOIS UML profile.

³This distinction is not part of standard UML — it is an extension of the OOIS UML profile.

Part III: Concepts

technique, notation, and implementation. For the time being, it is only important to emphasize that the two instances are distinguishable simply because they are two separate identities by their nature and existence (that is, because they have been created by two separate executions of the Create Object action).

The identity of an object is its inherent characteristic, and is independent of any attribute value of the object or any other part of its state. In other words, two objects can have exactly the same attribute values and exactly the same state, but they are still two different identities. For example, two instances of the class `Person` may have the same name, home address, date of birth, and all other properties, but they are still treated as two different and independent entities. In addition, the identity of an object is not affected by any modification of its attribute values.

The semantics may appear strange or even wrong to some readers experienced in and used to the traditional relational paradigm, where the identity of records is value-based (instead of existence-based, as in UML). In the relational algebra, two tuples with the same values of all their fields represent the same entity and are indistinguishable. To provide identity, the modeler must define a subset of fields that inherently ensure uniqueness of values, or to introduce a new field whose value is guaranteed to be unique. That is, the modeler must define the primary key. This is not the case in UML. Objects have their inherent existence-based identity regardless of their attribute values, and without any need to define an explicit “primary key” attribute.⁴

Of course, even for the existence-based identity semantics described for UML, an implementation can use the same technique for providing inherent identity. For example, an implementation can introduce an implicit ID field with an automatically generated unique value in each object, or use a unique pointer to the record that stores the entity in the computer memory. However, it is important to realize that this is only a matter of implementation, and not a concern of the modeler or user. Such a field (if it exists) is always hidden, and never needed in the model.

Quite naturally, those who are familiar with the traditional relational-based approaches to modeling information systems will have lots of questions at this stage. For example, why are the semantics defined in this manner? Why isn't it value-based as in the traditional approach? Why don't two objects with all attributes having equal values represent the same entity? How can the user distinguish two objects with all attributes having equal values? For example, how can the user distinguish two persons in an information system if they have their names, addresses, and all other data equal? What about the traditional understanding and the use of codes, identifiers, and so on? How do they fit into these semantics? Should they be used at all? Let's address these questions one by one.

The first logical question is why the identity of objects in UML is existence-based and not value-based, as in the relational algebra. Of course, it could have been defined that way, but it has been proven through practice that existence-based identity is better suited for modeling abstractions from the real world.

The notion of identity of objects stems from objects' historical origins in simulation languages (most notably, Simula). Objects were originally intended to model physical entities, which inherently have identity. Thus, even if you clone a cell, such that it is indistinguishable from the original cell in terms of its feature “values,” each cell is still physically distinct, composed of distinct physical matter. Two cars may be exactly alike, but they are physically distinct. Consequently, identity has very much to do with the notion of possessing or occupying an individual physical space and independent existence. This is why identity is key to modeling real-world things.

⁴This is often done by modelers who mistakenly mix the relational paradigm and UML modeling. One goal of this discussion is to show that this is unnecessary and wrong in true UML modeling, especially in OOS UML.

Indeed, to achieve the same effect, it appears that modelers of traditional information systems that rely on value-based identities usually introduce automatically generated ID fields as primary keys of entities because they either cannot rely on unique values of other fields, or value-based identity is not efficient (for example, because of propagated multi-field values as foreign keys in other tables, which may lead to clumsy foreign keys and inefficient indexing). On the other hand, when value-based identity is needed for a database table, it is likely that the table represents what is now called a data type in UML, which actually assumes value-based identity. Therefore, UML provides better separation of two kinds of identity, and spares the modeler and the user from dealing with implementation-related issues (such as existence and uniqueness of the identity field).

Second, it is reasonable to ask why it is allowed at all that two objects have all attribute values equal. A simple answer is that this is sometimes needed and useful. A classical example is the very usual and useful copy-paste operation in applications. When an original object is copied (actually, cloned), the newly obtained object intentionally has all attribute values equal to the original's ones, but it is still a new, independent entity. Even though the user cannot distinguish the original and the clone by their attribute values, the user has a clear understanding that they are two independent entities, at least through the effect of operations on them. For example, if the clone object is modified, deleted, or otherwise affected by a user action, the original object remains unaffected.

Examples of this approach are more than numerous: files in a file system, documents in a document management system, parts in a part list, items in an order, graphics in a diagram editor, and so on. All of these could be copy and pasted to improve a user's efficiency. The copy-paste operation is so frequent in contemporary applications that users are very used to it and expect it in most interactive systems. After all, it is a very useful usage pattern that may significantly improve application usability, especially when massive data must be input by users. And existence-based identity of objects inherently supports it.

Of course, there are also many cases where the system should not allow two objects to exist with some or all of their attributes having equal values. For example, it would be difficult to distinguish two persons recorded in the information system if they have all their attributes equal. Although the system treats them as separate entities, it is a matter of how the user can distinguish them. In order to cope with such a situation, the modeler should carefully analyze the problem domain and answer the following questions:

- Should it be allowed that objects of a certain class have some or all attribute values equal, or should some (combinations of) attributes have unique values?** In general, constraints should not be introduced unless there are compelling reasons because constraints make the implementation more complex and less efficient.
- If there is no constraint on uniqueness of attribute values, how can the user of the system distinguish two objects having the same attribute values?** Different techniques can be used to solve this problem without introducing uniqueness constraints. Some of these techniques tune the GUI as the user's view to the underlying object space in order to produce the right impression. They will be described in the discussion that follows.

The first approach tries to tune the GUI so that the user of the system is always aware that two different objects are separate entities. For example, they can always appear as two separate icons, or rows in a table/list when the user searches for all persons having the given name or home address.

The second approach tries to introduce new attributes that might help the user distinguish two objects. For example, if the system allows two people to have the same name, home address, and so on, why not introduce a photo as another attribute that is unlikely to be the same (but still not guaranteed by the system to be distinguishable by the user) for the two people that accidentally have the same name and live in the same apartment? This way, if users come across two people that have the same name

Part III: Concepts

and home address in the system, users can look at photos to find whom they are actually looking for. Biometric personal characteristics, in general, are more advanced examples of the same approach. Of course, this approach can be impractical or error-prone.

If none of these solutions is suitable, the modeler must consider introducing a new attribute that will be guaranteed by the system to be unique. Very often, the very problem domain requires introduction of such attributes. For example, for people, such attributes include identifications unique across a country (such as Social Security Number or personal identification number), the company (employee's ID), and so on. For books, it is the international identifier ISBN. For merchandise, it is a bar code, and so on. In general, codes, IDs, identification numbers, and other kinds of unique identifiers all fall into this category.

However, it is important to say that such attributes do *not* represent identities of the entities, but their *identifiers*. The difference is conceptual and subtle, although important. Namely, the identity is an inherent characteristic of these entities, while the identifiers are (most often artificially invented) properties that have unique values in a certain extent, and can serve to quickly and efficiently *refer* to a certain entity. In other words, identifiers serve as shortcuts for pointing to entities, or quick and easy ways to point to, search for, reach, or fetch those objects from large extents of their classes.

For example, a Social Security Number/personal identification number serves to quickly and unambiguously refer to a person or fetch the person's data from a huge governmental database. A bar code for merchandise is used as a machine-readable identifier of the merchandise, and so on. Similarly, an automatically generated unique ID primary key and a memory address pointer are the mechanisms used by a relational database and a C++ program to identify an object in the database or memory. Of course, this approach is very practical and widely used, and, thus, must be supported by a modeling language.

For that purpose, OOIS UML provides a predefined constraint `{unique}` that can be attached to one or more attributes of a class (see Figure 8-2). If it exists, the run-time environment will ensure that every instance of the class will have a unique tuple of values of the attributes to which the constraint is attached. If the same constraint (as a model element) is attached to several attributes, the entire tuple of values will be unique across the extent of the class, although each of these attributes can have non-unique values. Note that this is different from attaching several `{unique}` constraints to several attributes, where each of them will have a unique value. The constraint is checked at run-time in the same way as any other constraint. Only attributes of multiplicity 1..1 can have an attached `{unique}` constraint.⁵

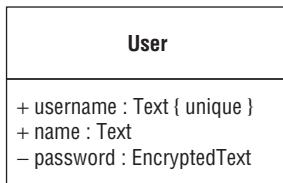


Figure 8-2: {unique} constraint for attributes

⁵Multiplicity of attributes was introduced in Chapter 7 and will be explained in detail in Chapter 9. 1..1 is the default multiplicity of attributes in OOIS UML.

Finally, there is the question of how two objects can be distinguished. The first context is the interactive manifestation — as already mentioned, it is up to the GUI to make the discrimination convenient for users. The second aspect is the programmatic manifestation — how to check whether two language elements refer to the same object. In the OOIS UML native detail-level language, the operators == (equal) and != (not equal) can be used to check whether two single-valued typed elements, whose types are classes, refer or do not refer to the same object (identity). The same holds for comparison in actions on multi-valued collections. For example, if two single-valued variables' types are classes, then the operators check the identity of the objects referred to by the variables, as shown here:

```
Car [*] parking = ...;  
Car myCar = ...;  
parking->forEach(c) {  
    if (c==myCar) ... // Is this the car I am searching for?  
}
```

Most traditional OO programming languages base the operators == and != on comparison of references to objects. Such operators, or other means of reference-based (that is, identity-based) equality comparisons can be used for objects of classes if another language is used at the level of detail. The use of built-in comparison operators of such languages is, however, dependent on the implementation, and a more reliable and portable solution is presented soon.

Section Summary

- ❑ Every instance of a class has its own *identity*, which is an inherent characteristic of that instance that discriminates it from any other instance of the same or any other class.
- ❑ The identity of an object is its inherent characteristic, and is independent of any attribute value of the object, or any other part of its state. The identity of objects is existence-based, not value-based, as in the relational algebra.
- ❑ When designing a class, the modeler should decide the following:
 - ❑ Whether some or all attributes should or should not have unique values.
 - ❑ If they do not have unique values, in what way the user will distinguish objects. Possible approaches include tuning the GUI or introducing additional attributes that will help the user.
- ❑ The predefined constraint {unique} can be attached to one or several attributes to ensure that the tuple of values of the attribute(s) will be unique across the extent of the class.
- ❑ The equality-comparison operators == and != for single-valued typed elements (whose types are classes) work on the identity base in the OOIS UML native detail-level language.

Part III: Concepts

Data Type Instances

Unlike class instances, instances of data types do not have their identity — that is, they are pure values. Two instances of the same data type with all attributes having equal values cannot be distinguished.⁶

For example, two instances of the `Text` data type, having the same strings of characters, cannot (and need not) be distinguished — the contents of the strings are important, not the instances themselves. Similarly, two instances of the `Date` data type representing the same physical day in the calendar do not need to be distinguished, simply because they really represent the same day. Again, this is because we are interested in the very physical day these instances represent and refer to, not in the instances themselves.

In fact, instances of data types, as pure values, actually represent *identifiers* of entities that have their identity, but exist *outside* the system's boundaries. Often, those entities represent some purely conceptual things. For example, instances (actually, their values) of the `Date` data type actually represent identifiers of (or references to) the physical days in the calendar, which have their identities. However, the days do not exist as tangible objects within the system, but are out of the system's scope, as shown in Figure 8-3. Similarly, instances of the `Integer` data type represent identifiers of conceptual, mathematical entities. In fact, the digit “1” can be thought of as a reference to the abstract concept of “one.”

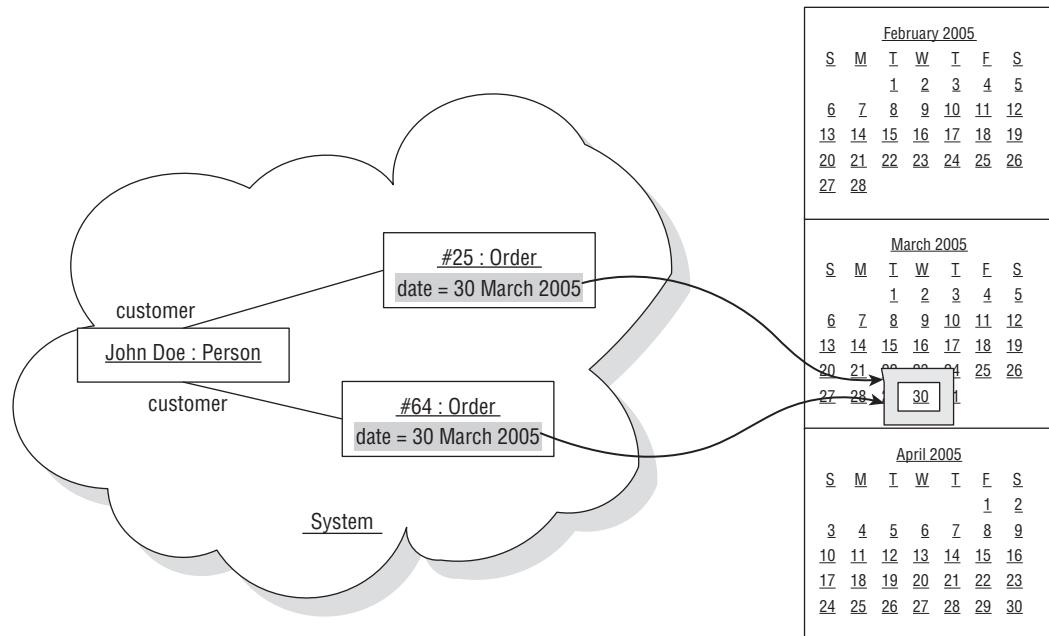


Figure 8-3: Instances of data types represent references to entities that exist outside of the system's boundaries.

The reason why you do not use classes in the model and objects in the system to represent these concepts, but use data types instead, whose instances just refer to those outer entities, is twofold. First, it is often not needed to use classes and objects with identity. You often do not need to model days with classes whose

⁶On a philosophical note, it can be argued that data values do have an identity because they occupy physical space and time. It's just that their identity is irrelevant.

objects will have identity and states modified through operations. It is enough for users to have values that unambiguously identify the entities from the real world, the meaning of which is clear to the user. The other reason is practical. It is often not practical and efficient to create an object of a class that models a physical day, and then to link many other objects to one single day in order to represent the fact that those objects have something to do with the same day. Additionally, the number of instances of such a concept may be huge or even infinite. Instead of creating objects, you use values coded in an efficient way to identify a huge (but, because of the limitation of physical resources in the computer system, limited) number of instances of such a concept.

Consequently, data type instances have no identity because the interest of the system and the user is in the values of those instances, not the instances themselves. Data type instances are used throughout the system as attributes of objects and arguments of operations, being copied and passed from one object to another and from one invocation of operation to another. All the while, only the value of the instance is important, not the instance itself. This is because the values represent references to entities that live outside the system.

It can be concluded that relational algebra uses the pure value-based semantics in which a tuple of values is always just an identifier of an entity that exists outside and independently of the information system. The system manipulates only with those values as identifiers, and not with the real entities outside the system. On the other hand, the object paradigm and UML use two separate concepts, classes and data types, whereby instances of the former have identity and are *simplified representations* of the real-world entities that the system manipulates directly, whereas instances of the latter represent identifiers that refer to the entities outside the scope of the system.

For all these reasons, in the OOIS UML native detail-level language, the equality-comparison operators == and != for single-valued typed elements whose types are data types work on the basis of values. That is, they compare the values of the referenced instances, not their identity (which is actually irrelevant). More precisely, two instances of data types are considered equal if they are instances of the same data type and all their attributes have the same values. The same holds for comparison in actions on multi-valued collections. Note that the comparison of attribute values introduces recurrence because those attribute values are also instances of data types. Consider the following example:

```
Integer i = ...;
...
Integer j = ...;
...
if (i==j) ... // The values of the integers are compared
```

Because traditional OO programming languages usually base their operators == and != on comparison of references to objects, these operators cannot be used for equality comparison of data values if such a programming language is used at the level of detail. To ensure value-based equality comparison for data values, the OOIS UML profile requires that the implementation in such a language provide the operation isEqualTo for data types. This operation accepts an argument that is a data value and returns a Boolean. It returns true if the given argument is equal to the data value whose operation isEqualTo is called. The previous example would then look like this:

```
Integer i = ...;
...
Integer j = ...;
...
if (i.isEqualTo(j)) ...
```

Part III: Concepts

The same operation exists and can be called for the same purpose even in the OOIS UML native detail-level language. In fact, the operators `==` and `!=` in this language fully rely on this operation. In addition, the method of this operation can be redefined in user-defined data types to refine the default value-based equality comparison semantics.

Although a user-redefined method for this operation can have an arbitrary implementation, it is highly recommended to keep it within the semantics of value-based equality comparison. Its purpose can be to enhance the notion of equality to comparison of abstract values instead of comparison of internal representation, or to inexact comparison, but not to change the meaning completely. It can also allow equality of instances of non-conformant types. For example, it can find that a `Complex` and an `Integer` are equal if the former represents the same integer as the latter, although they do not have the same internal representations.

Using two different notations for equality comparison of objects (`==`) and data values (`isEqualTo`) in a detail-level language is, of course, tedious and error-prone. Even worse, it makes the system extremely sensitive to changing design decisions. If, for example, a modeler changes a decision and “promotes” a data type into a class, because the entity represented by that classifier has to be moved into the system scope, the expressions in methods that rely on equality comparison would have to be rewritten (that is, the calls to `isEqualTo` should be replaced with `==`).

To avoid this problem, OOIS UML provides the same operation `isEqualTo` for equality comparison of both classes and data types, in the OOIS UML native detail-level language as well as in the API for a traditional programming language used at the level of detail. Of course, that operation works on the reference-based comparison for objects of classes and on the value-based comparison for the data values. If this operation is uniformly used for all instances regardless of their kind throughout the code of a method, the code becomes more robust, as shown here:

```
Car[*] parking = ...;
Car myCar = ...;
parking->forEach(c) {
    if (c isEqualTo myCar) ... // Is this the car I am searching for?
}
```

Section Summary

- ❑ Instances of data types do not have identity — they are pure values. Two instances of the same data type with all attributes having equal values cannot be distinguished.
- ❑ The equality-comparison operators `==` and `!=` for single-valued typed elements whose types are data types work on the value base in the OOIS UML native detail-level language.
- ❑ The operation `isEqualTo` provides equality comparison for both kinds of classifiers (classes and data types) in a uniform way. It compares the instances according to their kind (reference-based for objects and value-based for data values). Its method can be redefined.

Features

Classes and data types differ also in what structural features they can have, and in the semantics of their operations.

Structural Features⁷

On one hand, a class models an abstraction whose instances physically exist as objects within the scope of the modeled system. An association models structural connections between objects that explicitly exist within the scope of the system. Data types, on the other hand, model references to (that is, identifiers of) abstractions whose instances exist outside the scope of the system (see Figure 8-4a). Attributes, therefore, model implicit links from objects inside the system to those instances outside the system, as shown in Figure 8-4a. As a result, those implicit links cross the system's boundaries. This interpretation clearly separates the two concepts of class and data type.

To support this interpretation, in OOIS UML, types of attributes can be only data types, not classes (see Figure 8-4b). This holds true both for attributes of classes and of data types. On the other hand, participants of associations can be only classes, not data types (see Figure 8-4c).

However, standard UML does not differentiate attributes and association ends so strictly, but allows profiles to do that. In standard UML, classes and data types can both act as types of attributes or association ends. For that reason, both notations can be used for attributes and association ends even in the same diagram (see Figure 8-4d). For example, an attribute of type `Date` can be rendered using the textual representation within a compartment of the class symbol, or using the association-like notation with an open arrowhead pointing to the type of the attribute, and with the name of the attribute near the head of the arrow and without any adornments near the tail. Similarly, an association end can be rendered also using the textual attribute-like notation. The same holds true for object diagrams — attribute values can be rendered using the arrowhead notation between instances (see Figure 8-4e). However, to avoid ambiguities, these inverse notations are deprecated, although allowed in OOIS UML.

Consequently, as shown in Figure 8-4e, instances of data types or classes can be composed of other instances of data types as their attribute values, which can be further composed of other data type instances, and so on, until some primitive built-in data types are reached. Those primitive built-in data types provide elementary values whose implementation is provided by the environment. Put another way, instances of data types can be thought of as being constructed as hierarchical, tree-shaped composites of other data type instances, whose leaves are instances of primitive types. Note that the substitution rule always holds — an instance of a specializing data type can appear as a value of an attribute of a generalizing data type.

Because several actions on data type instances (such as value-based comparison or copying, to be explained later) require recursive traversal of attributes of data types, the structure of attribute values must be a tree (as shown in Figure 8-4e), and must not be a circular graph that would incur endless recursions. Therefore, a data type cannot be the type of a direct or indirect attribute of itself in a way that would implicate such endless recursions.

⁷These constraints on structural features of classes and data types are not part of standard UML, but of the OOIS UML profile. They are allowed by semantic variation points in standard UML, meaning that standard UML does not define constraints, but allows a profile to introduce them.

Part III: Concepts

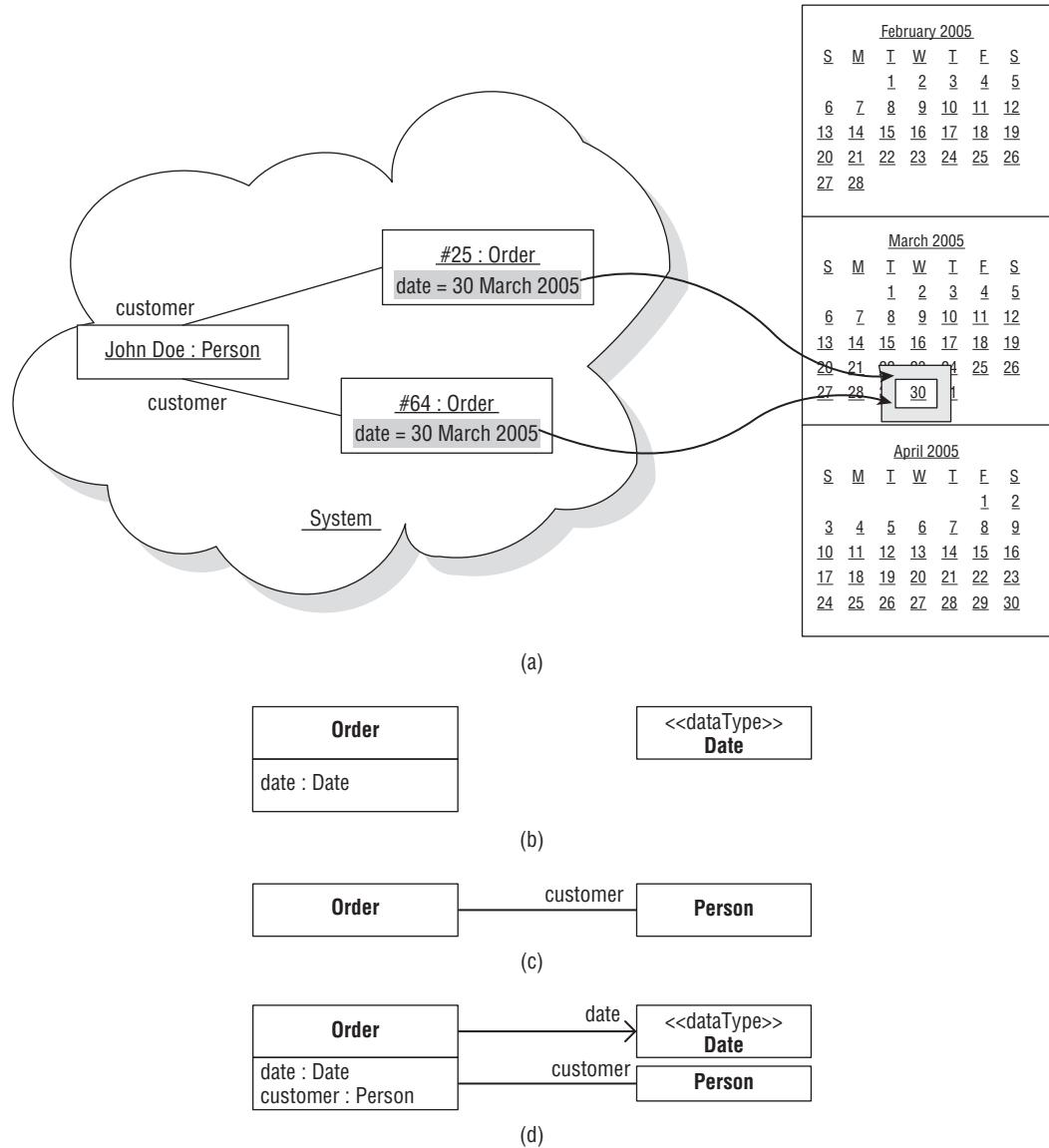


Figure 8-4: Attributes vs. associations in OOIS UML. (a) Attributes model references to conceptual entities that exist outside the scope of the system. Associations model explicit links between objects that physically exist within the system. (b) Types of attributes can be data types only. Data types cannot take part in associations. (c) Classes can take part in associations, but cannot be types of attributes. (d) Standard UML allows both notations for attributes and associations, even in the same diagram.

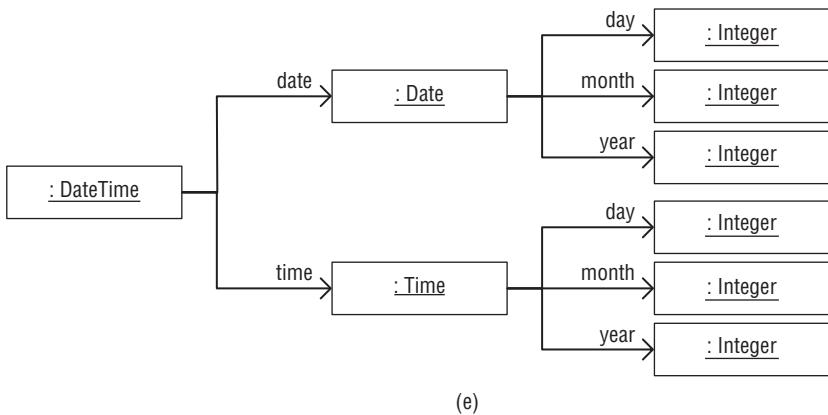


Figure 8-4: (e) Instances of data types can be composed of other instances of data types as their attribute values, which can be further composed of data type instances, and so on, until some primitive data types are reached. The arrows in this diagram represent values for attributes.

Section Summary

- ❑ A class models an abstraction whose instances physically exist as objects within the scope of the system.
- ❑ An association models structural connections between objects that explicitly exist within the scope of the system.
- ❑ A data type models references to abstractions whose instances exist outside the scope of the system.
- ❑ An attribute models implicit links from objects inside the system to instances outside the system.
- ❑ Types of attributes can be data types only, not classes.
- ❑ Participants of associations can be classes only, not data types.

Operations

Following the same interpretation of the meaning and purpose of classes and data types, both standard UML and OOIS UML differentiate the semantics of operations of classes and data types.

Because objects of classes physically exist inside the modeled system, having identity and state, operations on those objects can work on that state, possibly modifying it. Therefore, operations of classes can have arbitrary implementation — they can (but need not) modify the values of the object's attributes, and have other effects on the entire object space.

Part III: Concepts

On the other hand, instances that are pure data values represent identifiers of (that is, references to) entities that exist outside the system. Therefore, the operations of data types work on identifiers, not on the entities they actually refer to (and that live outside the system). Operations of data types must not modify attributes of the hosting instance, but they can only produce new data values as results. They must not affect the object space by any means.⁸ In other words, operations of data types must be pure functions without side effects. Consequently, instances of data types are immutable.

This is because a data type instance, as a pure value, represents a reference to an outside entity, which belongs to a certain set. Because there is no sense in modifying the very reference as a value, an operation should produce a new value as a reference to another, or the same element of the same or another set as an effect of an operation.

For example, instances of the data type `Text` may be thought of as being references to elements of the set of strings of characters. (Note that even a character, as a data type instance, may be thought of as being a reference to an elementary piece of human-interpretable information.) Concatenation of two strings produces a reference to a third element of the same set. Similarly, subtracting two instances of the data type `Date` may result in an instance of the data type `Integer`, representing the number of days between the two days referred to by the two given instances of `Date`.

Note that this holds true even if the same variable refers to an instance of a data type on which the operation works, and is then set to refer to the operation's result. Consider the following code:

```
Text t1 = "Hello ";
Text t2 = "World!";
t1 = t1.concat(t2);
```

In this example, the first two lines are declarations of two variables, `t1` and `t2`, the type of which is the `Text` data type, having the given initial values. The third line invokes the operation `concat` on the instance referred to by `t1`, with the argument referred to by `t2`. This operation produces a third instance of `Text`, having the concatenated value "`Hello World!`". The variable `t1` is then set to refer to that newly produced instance, and its reference to the initial instance "`Hello`" is released. It should be noted that the call of `t1.concat(t2)` does not have an effect on the value referred to by `t1`, which remains intact, but simply produces a new value.

A concrete implementation of the profile, of course, may incorporate some optimizations so that an operation on a data type instance works on the same memory slot, instead of allocating a new one, or by having a pool of values used throughout the program and sharing those values by references. However, an optimization should not, by any means, affect the described semantics. In other words, such optimizations are transparent to the programmer and should not be considered. Additionally, because different traditional programming languages have different semantics of instances of their types, it may happen that methods written in a classical programming language cannot support the described semantics, or that it introduces some restrictions.

⁸This is an OOIS UML-specific restriction.

Section Summary

- ❑ Operations of classes may modify the objects' state (that is, may have effects on the object space).
- ❑ Operations of data types must not modify values of their attributes, or have any effect on the object space. They must be pure functions that possibly produce new values.

Copy Semantics

As pure values without identity, instances of data types can be copied. Therefore, OOIS UML defines implicit copy semantics for data types. That is, every data type has an implicitly defined operation `clone`. This operation has no parameters, and returns an instance of the same data type. The default semantics of this operation are the deep copy of all attribute values. This means that a data type instance is copied (cloned) by propagating the invocation of `clone` to all its attribute values recursively, as shown in Figure 8-5.

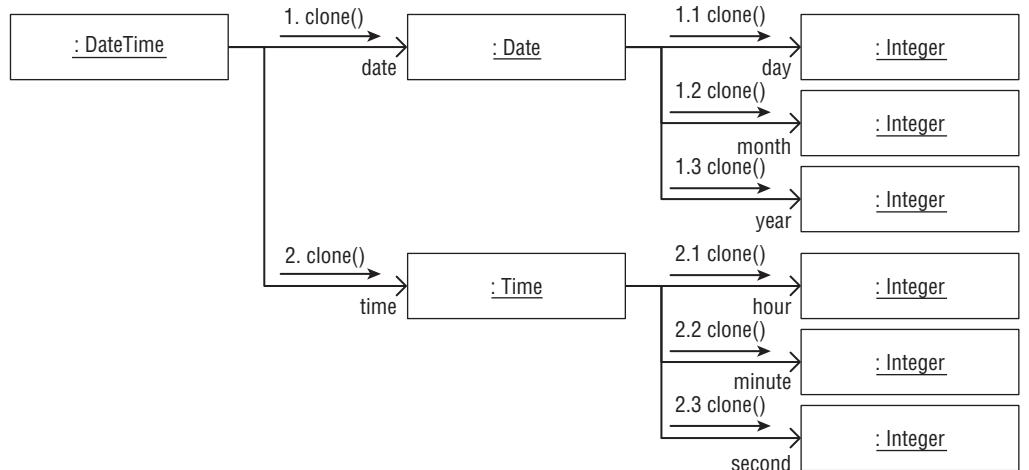


Figure 8-5: The deep copy semantics of the implicit `clone` operation of every data type

For example, in Java, C#, or the OOIS UML native detail-level language, after executing the following piece of code, two data type instances exist and `d1` and `d2` refer to them independently:

```

Date d1 = new Date(...);
Date d2 = d1; // Now d1 and d2 may refer to the same data type instance
d2 = d1.clone(); // Now d2 refers to a clone of the original instance

```

Part III: Concepts

Strictly speaking, because data type instances are pure data values, indistinguishable if they are of the same type and have all attribute values equal, the second line of the previous code snippet may also have implicit copy semantics. That is, after the following declaration, the variable `d2` can refer to a newly created clone of the instance referred to by `d1`:

```
Date d2 = d1;
```

Actually, because these two instances have no identity and cannot be modified, there is no means to distinguish between them later in the program. This is why it is up to the implementation to select which option to use, to share or clone data type instances when variables are set. Note, for example, that if Java is used as the detail-level language, the references will share a common instance after executing such a statement. Anyhow, this alteration should not affect the semantics of the program.

Because objects of classes can be linked by explicit links with other objects in graph-like forms having circular paths, there are no implicit copy semantics for classes. Hence, there is no implicit `clone` operation in classes. If a `clone` operation is needed, it must be modeled explicitly. Consequently, setting of variables that have classes as their types have pure reference-setting semantics — no objects are ever cloned. For example, these two variables always refer to the same object of the class `Car`:

```
Car c1 = ...;
Car c2 = c1; // c2 always refers to the same object as c1
```

Section Summary

- Every data type has an implicitly defined `clone` operation. It has no parameters, and returns the same data type. Its default semantics are deep (propagated) copying of attribute values.
- There are no implicit copy semantics for classes.

Lifetime

Objects of classes have an *explicit* lifetime. An instance of a class is explicitly created when an action `Create Object` is executed, and lives until an action `Destroy Object` is executed. An object will survive the execution in which a `Create Object` action was executed, and will live until it is destroyed by another action. This means that objects in OOIS UML are persistent by default. Hence, the term “*persistence*” has no particular meaning in OOIS UML.

On the other hand, instances of data types can be created explicitly, by executing the `Create Data Type Instance` action, but die *implicitly*, when they are not used any more. The modeler does not need to (and cannot) specify their destruction in methods. Instead, their lifetime is under the control of the run-time environment. When there are no more references that refer to a data type instance, it can be (but need not be) removed from the system. This corresponds to the background garbage collection mechanisms in some traditional languages.

However, because instances of data types do not modify their state, their disappearance cannot have any side effect on the object space. Therefore, the exact moment of their expiration is of no concern to the

modeler because it has no effect on the system's further behavior. For example, in the following example using the OOIS UML native detail-level language, after the execution of the addition and assignment in the third row, the data type instance (with value 1) previously referred to by variable *i* can be deleted automatically:

```
Integer i = 1;
Integer j = 2;
i = i+j;
```

However, in some other traditional languages, the described semantics cannot be supported in accordance with other semantic rules. For example, in order to support the substitution rule, copy semantics, and other rules, OOIS UML variables must be implemented in C++ through pointers to dynamic objects representing data type instances. In that case, the dynamic C++ objects must always be explicitly deleted using the `delete` operator:

```
Date* d1 = new Date;
...
delete d1;
d1 = ...;
```

Section Summary

- Objects of classes have an explicit lifetime. They are explicitly created and deleted by executing actions.
- Instances of data types have an implicit lifetime. They are created by executing actions, but are deleted implicitly, when they are not referred any more.

Creation and Destruction of Instances

Creation and destruction of instances are performed by the means of *actions* executed during the execution of behavioral features of classifiers (that is, methods). To support initialization of instances and clean-up on their deletion, OOIS UML provides *constructors* and *destructors* as specific operations of classifiers that are implicitly invoked on instance creation and destruction. To facilitate implementation of methods that exclusively create simple or complex object structures (which seem to be very frequent in applications), OOIS UML provides *creational object structures* that can be defined by static specifications or by demonstration. These are the topics covered in this section.

Actions

Standard UML defines two actions for creation and destruction of instances and their default semantics:

- **Create Object action** — Creates a new instance of the specified non-abstract classifier. This action has no other effects. In particular, no behaviors are executed, no initial expressions are evaluated, the new object has no structural features, and it participates in no links.

Part III: Concepts

- **Destroy Object action** — Destroys the specified instance. The action does not activate any behavior. It can (but does not need to) affect the links in which the instance being destroyed participates.

The idea of such simple semantics of actions in standard UML is that the actions have elementary, atomic effects on the object space, while all other more complex effects can be obtained by composing elementary actions into *activities*. In addition, standard UML is supposed to be general enough to be mapped to other conventional OO programming languages, and customized to specific application domains. However, this is not completely suitable for modeling OO information systems.

First, it does not distinguish between creation and destruction of objects and data type instances. Second, it does not define the semantics of initialization of structural features and activation of behavioral features (constructors and destructors). Finally, standard UML does not propose any unified notation for these actions.

Therefore, the OOIS UML profile tailors and extends the semantics of these actions, completely complying with the idea of profiling the UML. In other words, OOIS UML provides actions that encompass more complex semantics than standard UML actions, but can be expressed in terms of standard UML actions.⁹ Additionally, OOIS UML may differentiate between the way instances of classes and data types are created and destroyed. Finally, OOIS UML proposes a concrete notation for these actions, both in the OOIS UML native detail-level language, as well as in the conventional OO programming languages, when they are used at the level of details. The rest of this section discusses only the OOIS UML actions, not their definitions in standard UML.

When it is important to emphasize the difference between creating or destroying an object or a data value, the corresponding actions will be referred to as Create Object, Destroy Object, Create Data Type Instance, and Destroy Data Type Instance. However, at the places where the differences are not relevant or do not exist, both kinds of actions will be uniformly referred to as Create (Classifier) Instance and Destroy (Classifier) Instance.

Creation of Classifier Instances

Instances of classifiers (that is, objects or data values) are created by the Create Classifier Instance action. This action creates an object or a data value of the given non-abstract class or data type and returns (a reference to) that instance. For example, in the OOIS UML native detail-level language, the following code creates a new object of the class `Person`, and binds the variable `client` to refer to the new object:

```
Person client = new Person;
```

Therefore, the input parameter of the action Create Classifier Instance is the classifier of which the instance is to be created. That classifier must not be abstract, and must be accessible at the place of the action specification. The result of the action is provided on its output pin, and represents the reference to

⁹Formally, what is called an action in OOIS UML usually encompasses many elementary actions of standard UML, and is, thus, representing an UML *activity*. In OOIS UML, an action is defined from the perspective of the modeler (programmer) as a particular service of the run-time environment. The behavior that emerges from activating that service can be arbitrarily complex, and can encompass many elementary UML actions which have other side effects. However, this subtlety is of no particular importance to the practice, but simply ensures the alignment of the OOIS UML profile to standard UML.

the newly created instance. The new instance is a direct instance of the specified classifier. If it is an object of a class, it is added to the class's extent.

The Create Classifier Instance action, as most other OOIS UML actions, can be invoked in two different ways: by a statically modeled action and dynamically through the reflection mechanism.

In the first case, the action is specified within a method, using a detail-level language. The classifier whose object is to be created is statically specified and checked at compile-time. In other words, if any error exists in the specification of the action — such as the classifier does not exist, is inaccessible, or is abstract — it is reported at compile time. No run-time exceptions will be raised because of these reasons. In the OOIS UML native detail-level language, this kind of specification is supported by the operator `new`, which results in a value of multiplicity 1 and of type of the specified classifier. Its form is:

```
new classifier_name
```

For example, the following code excerpt creates a new object of the class `Person` and sets its attribute `name` to the given value:

```
(new Person).name = "John Smith";
```

In this example, the operator `new Person` creates a new object of the class `Person` and returns a reference to it. The dot operator `(.)` accesses its public attribute `name` and writes the value “John Smith” in it, by means of another Write Attribute action. Note that, although the reference to the new object is not remembered in any local variable in this code, the object is still accessible through other means, such as queries, or over links that the object’s constructor (to be explained in a later section) might have created as a side effect. It is part of its class extent and lives in the system’s object space.

If a conventional OO programming language is used, the same notational approach with `new` can be used, if the language and the implementation can provide the semantics described here for that notation. In other words, the examples given previously may look almost the same in such a language. Otherwise, the same effect can be achieved by means of static operations¹⁰ of classifiers provided implicitly by the implementation.¹¹ For example, to create a new object of the class `Person`, you may have to write the following in C++:

```
Person* client = Person::createInstance();
```

Or, similarly, you may have to write the following in Java and C#:

```
Person client = Person.CreateInstance();
```

The second means of invoking the same action is through the reflection mechanism of OOIS UML. In that case, the classifier whose instance will be created is provided dynamically, as an input pin of the

¹⁰Static operations are described in detail later. In brief, they are operations of classifiers that can be invoked independently of any particular instance, as in the examples given here.

¹¹The built-in operator `new` may be inadequate because these languages are required to support the additional OOIS UML semantics. For that reason, static operations provide wrappers and can be generated in the target code by model compilers. However, some implementations may overcome these limitations and still support the basic notation with `new`.

Part III: Concepts

action. Therefore, all violations (such as when the classifier is abstract or non-accessible) are reported through run-time exceptions. In the OOIS UML native detail-level language, this is specified as, for example:

```
Person client = (Person)UML::Action::createInstance("Person");
```

In this example, the OOIS UML reflection library (that is, the library of elements that provide access to the OOIS UML reflection mechanism), stored in the package named `UML`, in particular, the class `Action` from this library package, contains the (static) operations for all actions executable through reflection. The operation `createInstance` from this class accepts a string with the name of the classifier whose instance will be created (`Person`, in this case). The operation returns a reference of a generalizing type, which must be converted (downcast) to the type `Person` in order to be type-conformant to the variable `client`. This downcast (or type conversion) is denoted with `(Person)`.

Another operation from the same class, `Action`, accepts (a reference to) an object of type `Classifier`, which represents a classifier from the model within the reflection structure. For each class or data type in the system's model, an instance of the class `Classifier` will exist in the reflection environment. In order to access such an object, another reflection mechanism can be used, such as:

```
Classifier personClass = UML::Reflection::getClassifier("Person");
Person client = (Person)UML::Action::createInstance(personClass);
```

In a conventional OO programming language, the accompanying OOIS UML library must support the reflection mechanism in a completely equivalent manner. For example, the last two code excerpts would look almost the same in Java, C#, and C++, with only minor syntactical differences.

Regardless of the method of invocation, the following rules must be obeyed at the place of invocation of the action:

- The specified classifier must exist and must be accessible at the place of the action invocation (that is, within the namespace of the method in which the action occurs), according to the visibility rules for named elements.
- The specified classifier must not be abstract.
- The specified class must not be an association class (to be explained later).

In the case of static invocation, a compilation error will be reported if any of these rules is violated. In the case of dynamic invocation through reflection, a run-time exception of an adequate type is raised if a rule is violated.

Regardless of the method of invocation, the action has the following effect at run-time, in this order:

- 1.** A new direct instance of the specified classifier is created. If it is an object, it is added to the extent of its class.
- 2.** Attributes of the created instance are initialized by their default values. The order of initializing the attributes is undefined (that is, depends on the implementation).
- 3.** A predefined operation name `constructor` that exists in every classifier is invoked. This operation is called the `constructor` and may be used to initialize the instance (to be explained in section "Constructors" later in this chapter).

This entire process, including the execution of nested actions (because the constructor invocation, which implies an execution of a method, can incorporate execution of many other actions), is considered as a *constrained group* of actions, meaning that checking of all constraints is deferred until the entire process of executing the action (with all nested actions) is completed. In particular, the constraints attached to the structural features of the newly created instance (such as multiplicity constraints of its properties) are not checked until the end of this process. This process is, therefore, considered as transitional, during which some constraints may not hold, because the instance is still being initialized. This period is referred to as *(object) initialization*.

For most of the built-in data types, the OOIS UML native detail-level language supports *literals* of that type. Literals are constants statically specified within the code, which represent (references to) anonymous instances of the corresponding data types at run-time.

For example, a literal 2.5 represents a reference to an instance of the data type `Real` that is implicitly created at run-time at the place of its occurrence, and the literal `true` represents a reference to an instance of the data type `Boolean`. In other words, a literal encompasses an implicit Create Data Type Instance action at the place of its occurrence, and represents a reference to the created instance, just as the operator `new` does explicitly.

Of course, the instance of a built-in data type has the specified value that represents the specified element of the corresponding set. Here are some examples:

```
Boolean b = false;
Integer i = 2;
Text txt1 = "Hello ";
Text txt2 = txt1.concat("world!"); // txt2 will be "Hello world!"
```

Section Summary

- ❑ An instance of a classifier is created by the Create Classifier Instance action.
- ❑ This action can be invoked as follows:
 - ❑ Statically (`new classifier_name`).
 - ❑ Dynamically, through the reflection mechanism.
- ❑ The run-time effect of this action is as follows:
 - ❑ A new direct instance of the specified classifier is created. If it is an object, it is added to the extent of its class.
 - ❑ Attributes of the created instance are initialized in an implementation-dependent order.
 - ❑ The constructor of the instance is invoked.
- ❑ This entire process of initialization (including the execution of nested actions) is a *constrained group* of actions, meaning that all constraints are not checked until the entire process of initialization is completed.

Destruction of Objects

An object is destroyed using the Destroy Object action, which can be invoked in the OOIS UML native detail-level language in several different ways:

- Using a predefined operation `destroy` that exists in every class:

```
Person p = ...;  
...  
p.destroy();
```

- Using the operation `destroyInstance` of the `UML::Action` class from the OOIS UML library:

```
Person p = ...;  
...  
UML::Action::destroyInstance(p);
```

In other conventional OO programming languages, the way of invocation can be similar, with only minor syntactical differences.

Regardless of the way of invocation, the action Destroy Object has the following effects at run-time, in this order:

1. A predefined operation `destructor` that exists in every class is invoked. This operation is called the *destructor* and may be used to clean up the object (to be explained later in this chapter).
2. The objects to which destruction is implicitly propagated are destroyed by the Destroy Object action, in an implementation-specific order (to be explained later in this chapter).
3. The links linked to the object are destroyed by the Destroy Link action, in an implementation-specific order.
4. The object itself is destroyed and removed from the extent of its class. Of course, the attribute values of the object disappear along with the owner object.

Similar to the Create Classifier Instance action, this entire process, including the execution of nested actions (because it may encompass many Destroy Link and Destroy Object actions, and the destructor invocation, as an execution of a method, can incorporate execution of many other actions) is considered a constrained group of actions, meaning that all checking of constraints is deferred until the entire process of executing the Destroy Object action (with all nested actions) is completed. In particular, the constraints attached to the structural features of the affected object structure (such as multiplicity constraints of the properties of the objects to which the object being destroyed was linked) are not checked until the end of this process. This process is, therefore, considered as transitional, during which time some constraints may not hold. This period is sometimes called *object finalization*.

Section Summary

- An object is destroyed using the Destroy Object action.

- The run-time effect of this action is as follows:
 - The destructor of the object is invoked.
 - The objects to which destruction is implicitly propagated are destroyed by the Destroy Object action, in an undefined order.
 - The links linked to the object are destroyed by the Destroy Link action, in an undefined order.
 - The object itself is destroyed and removed from the extent of its class.
- This entire process of object finalization (including the execution of nested actions) is a constrained group of actions, meaning that all constraints are not checked until the entire process of finalization is completed.

Destruction of Data Type Instances

As already explained, instances of data types are destroyed implicitly rather than explicitly. In general, a data type instance is destroyed when nothing refers to it anymore, which happens at a certain point in time under the control of the run-time environment. There is no need for any explicit action or other specification for this in OOIS UML. In particular:

- When an instance of a class or data type is destroyed, all its attribute values (being instances of data types), are implicitly destroyed because nothing refers to them anymore. Note that these attribute values (as instances of data types) may also have their attribute values, which are then also destroyed, and so on recursively. This way, the destruction of attribute values is implicitly propagated down their hierarchical structure, in an undefined order.
- When a variable or argument referring to instances of data types expires, because the execution exits its scope (that is, the enclosing method), the referred data type instances disappear in an undefined order, too. Note, again, that the destruction is propagated implicitly to the attribute values of the data type instances.

The destruction of a data type instance has no effects other than the propagation of the destruction to its attribute values in an undefined order, and the disappearance of the instance itself at a certain moment in time.

The described semantics can be understood in yet another, more formal way. Namely, an attribute, variable, or parameter of a method formally defines a *mapping* from an object (that is, its identity), or a name of the variable/parameter to a data value (or a collection of data values, depending on its multiplicity). For example, an attribute name of the class Person defines a mapping from the set of (live) objects of that class to the values of the type Text. In that sense, the value of the attribute or the variable/parameter can be understood as the pure result of that mapping performed at run-time (for example, the result of the Read Attribute Value or Read Variable action), not as a certain structure of bits that is stored somewhere in the computer memory. In other words, it can be understood purely functionally, not structurally.

Consequently, when the owner object or variable/parameter disappears, the mapping also disappears — that is, becomes undefined for that object or variable/parameter. Formally, this does not mean that any structure has to be destroyed (as is the case with objects), but simply that the function

Part III: Concepts

becomes undefined for the given object or variable/parameter. Of course, the implementation will most probably rely on a structure of values stored in computer memory to define the mapping in an explicit form. But this is not a formal implication.

When a traditional language with automatic garbage collection is used as the detail-level language, the semantics are inherently supported by the language's native mechanisms. However, in some other traditional languages, the semantics cannot be supported in accordance with other semantic rules. For example, in order to support the substitution rule, copy semantics, and other rules, OOIS UML variables must be implemented in C++ through pointers to dynamic objects representing data type instances. In that case, the dynamic C++ objects must always be explicitly deleted using the `delete` operator, as shown here:

```
Date* d1 = new Date;  
...  
delete d1;
```

Section Summary

- ❑ Instances of data types are destroyed implicitly, rather than explicitly. In general, a data type instance is destroyed when nothing refers to it anymore, at a certain point in time under the control of the run-time environment.
- ❑ The destruction of a data type instance has no effects other than the implicit propagation of the destruction to its attribute values in an undefined order, and the disappearance of the instance itself at a certain moment in time.

Constructors¹²

Let's consider several examples whose models are shown in Figure 8-6. Assume that the requirements from the problem domain impose their form and interpretation as follows:

- ❑ **Figure 8-6a** — When a car is created, it should also create four wheels and an engine for itself (see Figure 8-6a). In other words, whenever an object of the class `Car` is created, the appropriate objects of the classes `Engine` and `Wheel` should also be created and linked to the `Car` object.
- ❑ **Figure 8-6b** — When an employee is created, it can (but need not) be assigned to a given department. In other words, whenever an object of the class `Employee` is created, the appropriate object of the class `Department` may be supplied as a parameter, and the `Employee` object is linked to that `Department` object in that case.
- ❑ **Figure 8-6c** — When a user is created, it must be assigned a unique username and a user profile. In other words, it must be ensured that whenever an object of the class `User` is created, the appro-

¹²This is exclusively an OOIS UML concept. It does not exist in standard UML.

priate value for the attribute `username` is supplied and stored in that attribute, and an object of the class `UserProfile` is supplied and the `User` object is linked to that `UserProfile` object.

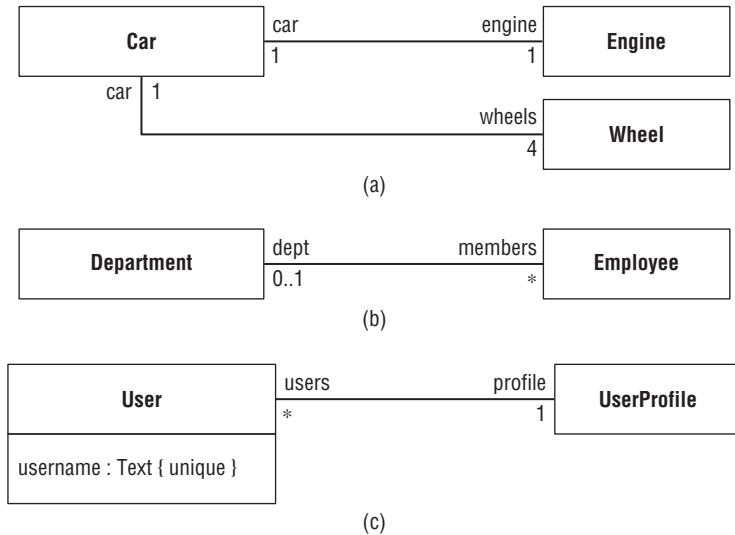


Figure 8-6: Examples of needs for a constructor. (a) When a car is created, it should also create four wheels and an engine for itself. (b) When an employee is created, it can (but need not) be assigned to a department. (c) When a user is created, it should be assigned a username and a user profile.

All these examples have in common that something must be ensured or possible whenever an object of a class is created — that is, that some activities are done during the very initialization of the object. For that purpose, *constructors* are available in OOIS UML. A constructor is a specific operation of a classifier that is implicitly invoked during every initialization of its instance. It is invoked as the final step in execution of the Create Classifier Instance action.

In OOIS UML, any operation of a class or data type that has the name `constructor` is a constructor. A classifier can have an arbitrary number of constructors. These constructors may have different number and/or types of parameters, thus allowing different means of object initialization. For example, several constructors of the class `Employee` in Figure 8-6b can exist to provide different ways of initialization:

- ❑ A constructor that accepts no parameters will simply do nothing (that is, leave the default initial state of the `Employee` object).
- ❑ A constructor that accepts a textual parameter by which the `name` attribute of the employee will be initialized at the very creation has the following specification:

```
Employee::constructor (nm : Text[1])
```

Its method coded in OOIS UML native detail-level language can simply be as follows:

```
name->set (nm) ;
```

Part III: Concepts

- A constructor that accepts an object of the class `Department` will link the newly created object of `Employee` to the given department. Its specification can be as follows:

```
Employee::constructor (d : Department[1])
```

Its method coded in the OOIS UML native detail-level language can simply be as follows:

```
dept->add(d);
```

- A constructor that accepts both the name and the department and does what the previous two constructors do.

Every class or data type in OOIS UML always has at least one predefined constructor that has no parameters. This one is called the *default constructor*. The default (inherited) method for this constructor is empty (does nothing), but can be redefined in every classifier.

The constructor is invoked as the final step in the execution of the Create Classifier Instance action. By default, if nothing is specified for the invocation of the Create Classifier Instance action, the default constructor is invoked. Otherwise, if the caller of the Create Classifier Instance action wants another constructor to be invoked, the arguments for that constructor must be provided with the action. For the given example of creating an object of `Employee` in Figure 8-6b, the following action in OOIS UML native detail-level language will call the default constructor:

```
new Employee
```

The following action will call the constructor that accepts only the name of the employee:

```
new Employee("John Smith")
```

The following action, where `headquarters` is a previously defined variable of type `Department`, will call the constructor that accepts both the name and the department parameters:

```
new Employee("John Smith", headquarters)
```

As you can see, the parameters of the constructor invocation are simply specified in parentheses after the name of the class, as with any other operation invocation.

In general, the action invokes the constructor that best matches the types of the actual arguments specified in the action. What “best matching” means (that is, the precise operation resolution policy), is the same as for any other operation invocation, and will be explained later in this book for operations in general. Anyhow, if there is no constructor that matches the actual arguments specified in the action, the action is illegal and a compilation error occurs. Therefore, the set of constructors of a classifier specify all allowable means of object initialization, and no other means are possible than those covered by the constructors of the classifier. For example, if the listed constructors are all constructors of the class `Employee`, the following action would be illegal:

```
new Employee("John Smith", 25) // We wanted to specify age of the employee
```

For dynamic creation of objects through reflection, OOIS UML does not support invocation of constructors with arguments. In other words, if an object is created dynamically through the reflection mechanism, the default constructor is always called.

Constructors are, in all other ways, equal to ordinary operations. They can be of any visibility kind, having the usual consequences on their accessibility from the place of their invocation (where the action Create Classifier Instance is specified). Constructors are polymorphic operations, meaning that they (as operations) are inherited in derived classes, and their methods are inherited by default, but can be redefined in derived classes.

For example, if the class `Manager` specializes the class `Employee`, there is no need to define any new constructor in that class, and an object of that class can still be created in all ways as employees:

```
new Manager;
new Manager("John Smith");
new Manager(headquarters);
new Manager("John Smith", headquarters);
```

Unlike in some other traditional OO programming languages, the redefining method of a constructor in the derived class does not implicitly call the method from the base class. This must be explicitly ensured if needed, as with any other operation. This is done by an explicit call of the constructor of the base class, in the same way as it is done in the particular detail-level language for any other inherited but overridden method. For example, the inherited method of an operation can be invoked from the overriding method in the OOS UML native detail-level language by simply referring to the keyword `super` like this:

```
super("John Smith", headquarters);
```

Traditional OO programming languages provide similar means for the same purpose.

Interestingly, constructors may also be invoked explicitly, having the same effect as any other invoked operation for an object. For example, the method for the last constructor of the class `Employee` (which initializes the name of the employee and assigns him or her to the given department) may be simply coded as follows:

```
constructor(nm);
constructor(dept);
```

Constructors of data types have one specific characteristic. As with all other operations of data types, the methods of constructors must not (and cannot) modify the attributes of its host instance. Instead of modifying the attributes of its host instance, the methods should produce a new data type instance with the properly set attribute values. They can do that through actions on attributes that will be described in Chapter 9. These actions always create new data values with the properly modified attribute values, instead of modifying the values they work with "in place."

For that reason, constructors of data types must always return a reference to a data value that also will be returned as the result of the Create Classifier Instance action that invoked the constructor. The return type of such a constructor is always its owner data type, and does not have to be explicitly specified in the model.

For example, let's assume that the data type `Point` has two attributes, `myX` and `myY` of type `Real`. The constructor `constructor(x:Real[1],y:Real[1])` of this data type, which initializes a point with the given coordinates, can be coded as follows:

```
Point toReturn = this.myX->set(x);
toReturn = toReturn.myY->set(y);
return toReturn;
```

Section Summary

- ❑ A *constructor* is an operation of a class that is implicitly invoked as the final step in execution of the Create Classifier Instance action.
- ❑ Any operation of a classifier that has the name `constructor` is a constructor. A classifier can have an arbitrary number of constructors, possibly having different number and/or types of parameters.
- ❑ Every classifier always has the *default constructor* that has no parameters. The default (inherited) method for this constructor is empty (does nothing), but can be redefined in every classifier.
- ❑ If the caller of the Create Classifier Instance action wants a constructor other than the default one to be invoked, the arguments of that constructor must be provided with the action. The appropriate constructor must exist in the classifier (possibly inherited); otherwise, the action is illegal.
- ❑ Constructors of data types must return a newly created data value with the attributes initialized as needed.
- ❑ Constructors are, in all other ways, equal to ordinary operations.

***Creational Object Structures*¹³**

Creators are methods that just create pieces of object structure (that is, create objects and links) and set up attribute values of the created objects, but do not have other side effects. Apart from using a detail-level language as with any other method, creators can be specified using creational object structures, defined either by static specifications or by demonstration.

Creators

Let's consider a part of the information system for supporting the production process in an imaginary car manufacturing company. For the purposes of this example, significant simplifications will be made to the real-world system. However, these simplifications will not affect the general nature of the example and the main point of explanations. Only the irrelevant subtleties will be omitted for the sake of simplicity.

According to the orders submitted to the sales department and long-term production plans, the production management of the company issues a *production order* for the factory. An example of the production order form filled-in by the management is shown in Figure 8-7a.

The production order has its date and other data related to the order as a whole (omitted here), and a list of *items*, each of which specifies the quantity and the characteristics of one type of car to be produced. The specifications include the following:

- ❑ **The basic type of the car** — Can be picked up from a list of car types produced by the company.
- ❑ **The variant (or model) of the car (for example, cargo, convertible, limousine, and so on)** — Can also be picked up from a list of the variants for the selected type of the car.

¹³This is exclusively an OOIS UML concept. It does not exist in standard UML.

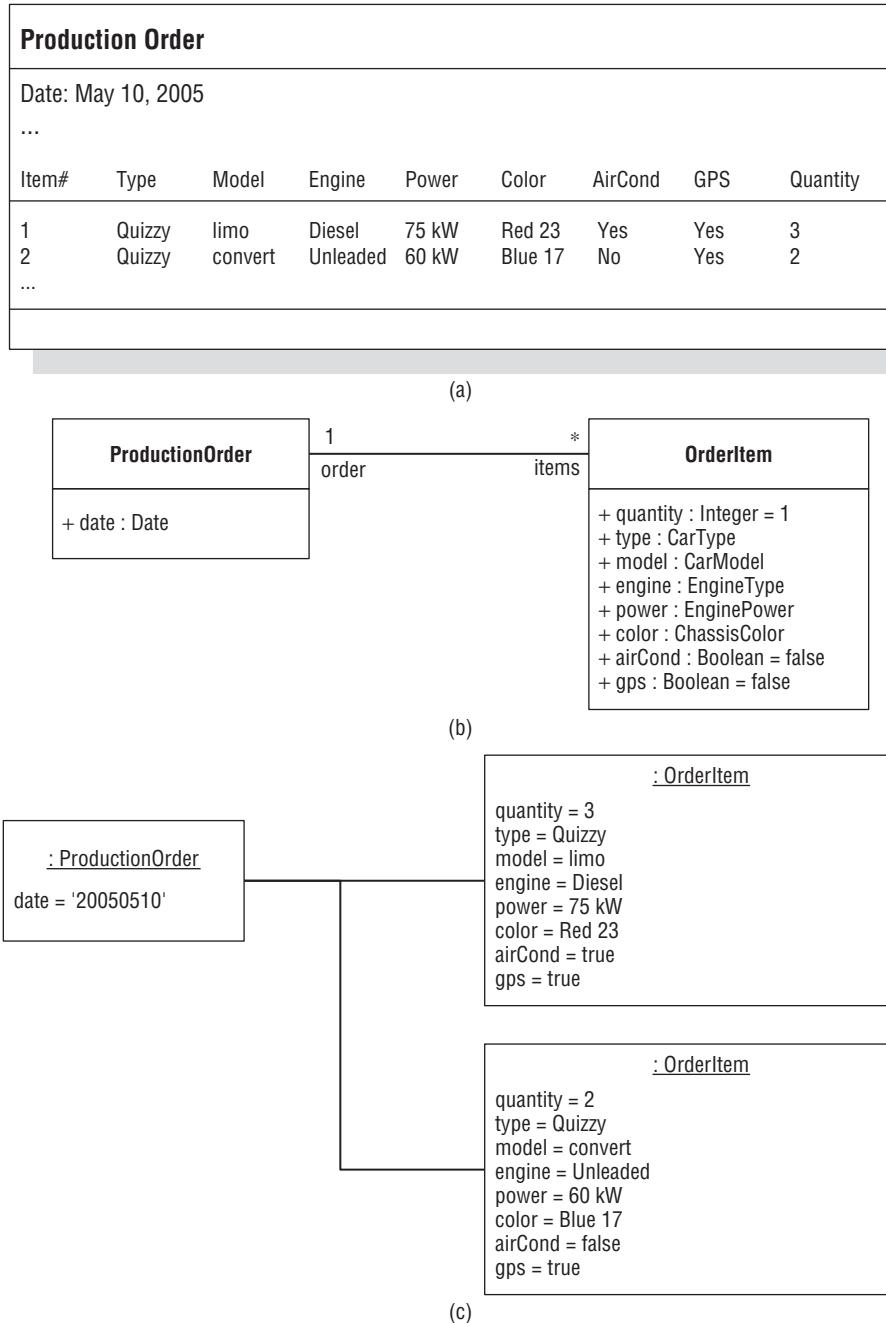


Figure 8-7: Car manufacturing example — the source domain of the transformation. (a) A sample filled-in production order form. (b) The conceptual model of the production order. (c) Object diagram showing the structure of objects carrying the information from the form in figure (a).

Part III: Concepts

- ❑ **The type of the engine in the car** — Can be either unleaded or diesel.
- ❑ **The power-rating of the engine** — Can be selected from a predefined list of available variants of the selected engine type.
- ❑ **Color of the car** — Can be selected from a list of the colors from the palette offered by the company.
- ❑ **Air conditioner** — Should the ordered car have an air conditioner or not?
- ❑ **Global Positioning System (GPS)** — Should the ordered car have a GPS navigation device or not?

The last two of these are offered to the customers as options, so the customers can select which of the optional equipment should be mounted on the ordered car. (Of course, a real-world list of options would be much longer, and the production order more complex.)

Figure 8-7b shows the conceptual model of the described production order. In the information system, each production order will be an object having the attribute values from the order form. It is linked to an arbitrary number of objects of the class `OrderItem`, each one carrying the information from one line of the form in its attribute values. The types of the attributes for which the form offers a predefined list of options will be modeled as domain-specific data types (more precisely, as enumerations). They will be sets of predefined values, one value existing for each option from the corresponding list (these data types are not shown in the diagrams). For example, the data type `EngineType` will have only two values: `Unleaded` and `Diesel`.

Figure 8-7c shows the object diagram with the structure of objects that carry the information from the filled-in form in Figure 8-7a. Actually, the piece of software that deals with filling in the production order form has the responsibility of creating such an object structure “behind the scenes.” In other words, the production order form in Figure 8-7a is just the appropriate representation in the GUI of the application for the object structure shown in Figure 8-7c that exists in the background.

For each particular car that is to be produced, the system should create an appropriate configuration of objects that will represent the major parts of that car — the chassis, the engine, and the mounted equipment parts (air conditioner and GPS). Such a structure of objects should be created for many reasons important for the company and its internal processes:

- ❑ Aside from the general production order described so far, two partial production orders should be created for the departments that manufacture chassis and engines. Therefore, for each general production order, a separate chassis production order and an engine production order should be created, carrying the information for every part (chassis or engine) that should be manufactured. Of course, this information is appropriate for the particular part. For example, the chassis production order carries the information about the color of the chassis, while the engine production order carries the information about the type and power-rating of the engine.
- ❑ The internal quality assurance policy requires that each of the car’s major parts (as well as the entire car) be tracked during its entire production, quality assurance, and exploitation. For example, the policy requires that the system tracks who mounted the part on the car and when, who checked and approved its presence and quality and when, or whether the customer has reported a problem with the part during its exploitation.
- ❑ The documentation for the product (for example, technical or user manuals) may be generated depending on its configuration.

Figures 8-8a, 8-8b, and 8-8c show the diagrams of the conceptual model of these aspects of the system. A *part* (see Figure 8-8a) is an abstract generalization of all kinds of parts constituting a car. These are *chassis*, two kinds of *engine* (unleaded and diesel), and different kinds of *equipment* (air conditioner and GPS).

Let's assume that the car manufacturer can mount different models of the air conditioner, but it does not offer this as an option to the customer. Instead, the actual air conditioner model is mounted until a new model occurs in the market. Figure 8-8b shows that each car must have exactly one chassis, one mounted engine, and an arbitrary number of pieces of equipment. Finally, Figure 8-8c shows the concepts of partial production orders for engines and chassis. For the tracing purposes, each of these partial production orders must have a link to the original production order from which it was created. This is modeled with an association between *PartProductionOrder* and *ProductionOrder*.

For the production order example shown in Figure 8-8c, a rather complex configuration of objects representing the partial production orders, cars, and their parts should be created by the system. This configuration is shown in Figure 8-8d. This diagram shows the objects, their attribute values, and links that should be created. First, three identical groups of objects should be created for the first item in the initial production order because the quantity of that item is set to 3. The objects from the group should have their attribute values set to the values taken from the corresponding attributes of the *OrderItem* source object. For example, the value for the attribute *Car*::*type* should be taken from *OrderItem*::*carType*, while the value for *Chassis*::*model* should be taken from *OrderItem*::*model*, and so on. Finally, the *Chassis* and *Engine* objects should be linked to the created objects of *ChassisProductionOrder* and *EngineProductionOrder*, which should also be created to represent the partial production orders. The same holds true for the second item in the original production order, from which two identical groups of objects should be created, as shown in the lower part of Figure 8-8d.

To sum up, the task here is to specify the method that should create an object structure like the one in Figure 8-8d out from an object structure like the one in Figure 8-8c. The method under consideration can be the implementation of the operation *ProductionOrder*::*createProducts*, so that it can be invoked for the given particular production order. The method should create an object structure according to the class model shown in Figures 8-8a, 8-8b, and 8-8c, from an object structure that conforms to the class model shown in Figure 8-7b.

In other words, the method under consideration actually performs a *transformation* of the *source* object structure to the *target* object structure, by creating the target object structure using the information from the source object structure. The source object structure conforms to the *source class model*, which represents the *source domain* of the transformation, while the target object structure conforms to the *target class model*, which represents the *target domain* of the transformation. The task is to specify the method that performs the transformation. The specification would show how the object structure from the target domain should be created from a structure from the source domain, thus representing the *mapping* of the domains (or *domain mapping*). Of course, the source and the target domains can be the same.

The method in consideration is characteristic because it only creates a piece of object structure (that is, it creates some objects and links), and sets the values of some attributes of those objects. It does not delete any objects or links, invoke any operations explicitly (constructors of created objects are called implicitly), or have any other side effects, either. Such methods are called *creators* in OOIS UML.

Creators appear quite often in information systems. Transformations like the described one are just one of their applications. Transformations are generally useful when a complex object structure should be created using the information from another structure according to a pattern that can be formally specified. Additionally, they can be used, for example, to transform an existing object structure, made according to a legacy class model, into a new structure that conforms to a modified or extended class model of a newer version of the system.

Part III: Concepts

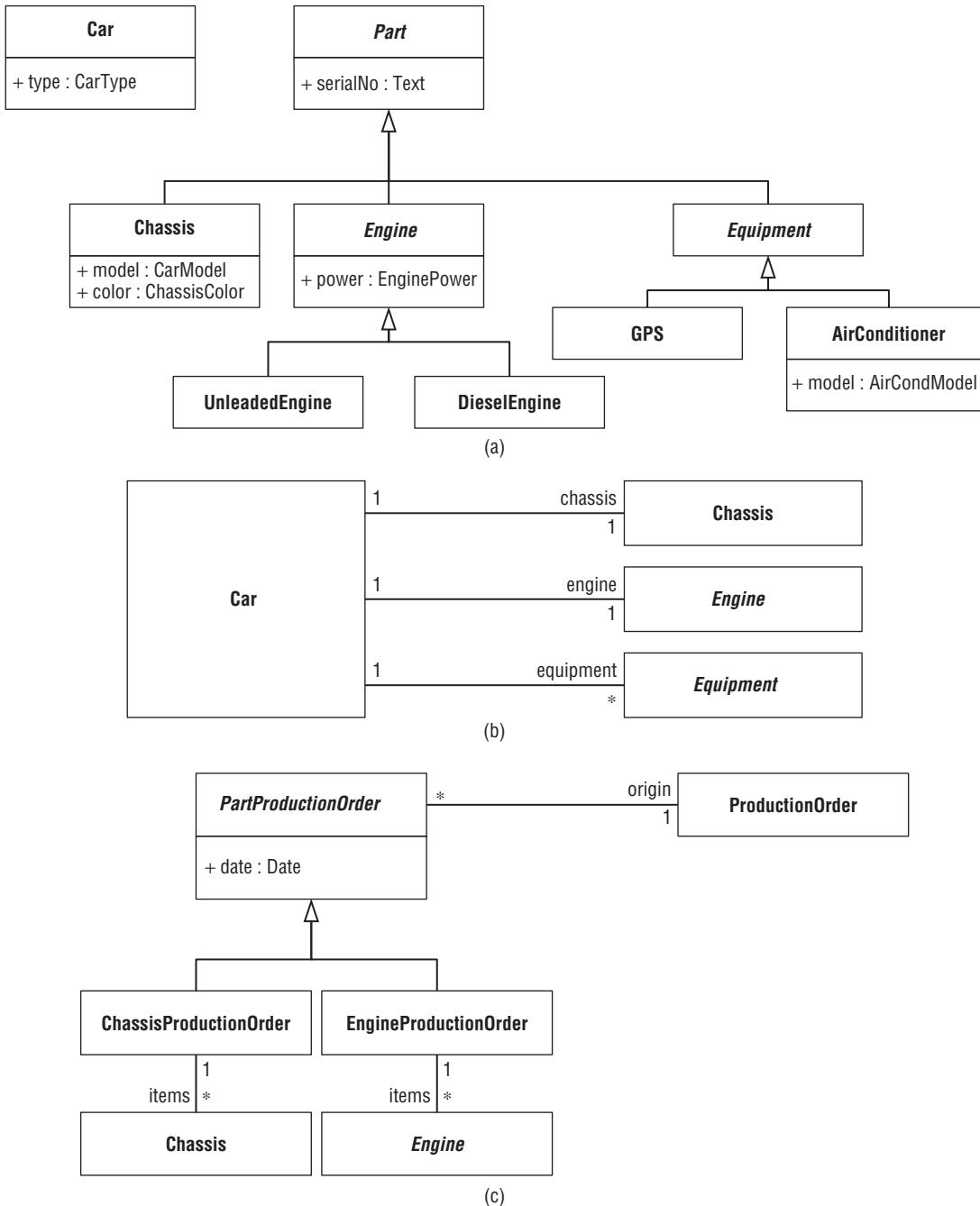


Figure 8-8: Car manufacturing example — the target domain of the transformation. (a, b, c) The diagrams that show the conceptual model of the target domain.

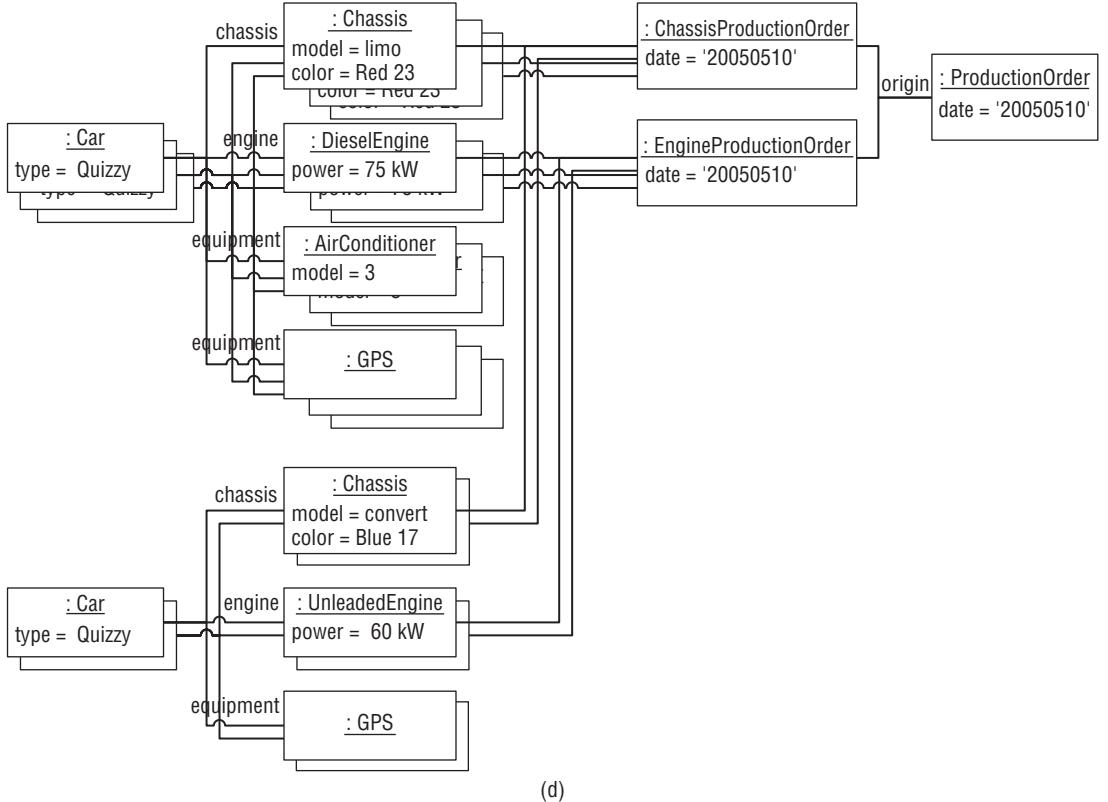


Figure 8-8: (d) The object diagram that shows the configuration of objects that should be created by the transformation from the configuration of objects in Figure 8-7c.

Moreover, creators have a wider use. They often perform very simple tasks. Typically, constructors of some classes must create some objects and links, or just link the object being constructed to other objects referred to by the parameters of the constructors. All examples in Figure 8-6 fall into this category. Additionally, GUI configuration specifications, like those shown in Figures 6-2 and 6-6, are actually specifications of creators.

Section Summary

- ❑ Methods that just create pieces of object structure (that is, create objects and links) and set up attribute values of some objects, but do not have other effects, are called *creators*.
- ❑ Creators can be used for simple structural manipulations (for example, creating a couple of objects and links), just as constructors often can.

Continued

- ❑ A creator can be used for a *transformation* of the *source* object structure to the *target* object structure, by creating the target object structure using the information from the source object structure. The source object structure conforms to the *source class model*, which represents the *source domain* of the transformation. The target object structure conforms to the *target class model*, which represents the *target domain* of the transformation. The creator specifies the *mapping* from the source to the target domain.

Creational Specifications

Now let's write the code for the creator that should perform the transformation described for the car-manufacturing example. Using the OOS UML native detail-level language, the code for the method of the operation `ProductionOrder::createProducts` may look like this:

```
ChassisProductionOrder cpo = new ChassisProductionOrder;
cpo.date = this.date;
cpo.origin->add(this);

EngineProductionOrder epo = new EngineProductionOrder;
epo.date = this.date;
epo.origin->add(this);

this.items->forEach(item)

// A simple counter loop that iterates item.quantity times:
for (Integer i = 1..item.quantity) {

    Car car = new Car;
    car.type = item.type;

    Chassis ch = new Chassis;
    ch.model = item.model;
    ch.color = item.color;
    car.chassis->add(ch);
    cpo.items->add(ch);

    Engine eng;
    if (item.engine==EngineType::Unleaded) eng = new UnleadedEngine;
    if (item.engine==EngineType::Diesel)    eng = new DieselEngine;
    eng.power = item.power;
    car.engine->add(eng);
    epo.items->add(eng);

    if (item.airCond) {
        AirConditioner ac = new AirConditioner;
        ac.model = AirCondModel::3;
        car.equipment->add(ac);
    }

    if (item.gps) {
        GPS g = new GPS;
        car.equipment->add(g);
    }
}
```

```
}
```

```
}
```

It is obvious that the use of a textual detail-level language for specifying creators (especially those for complex transformations) has some severe drawbacks:

- ❑ The work of writing the code is tedious, time-consuming, and error-prone.
- ❑ The code tends to become cluttered, difficult to understand, and difficult to maintain. Algorithmic decomposition into sub-methods may help for more complex transformations, of course, but the work of their specification then becomes even more difficult.
- ❑ The code is tied to the detail-level language used. If the code should need to be translated to another detail-level language, for any reason whatsoever, it will have to be re-written.

Let's analyze the background of this process. In order to specify a complex transformation, the developer would probably draw a sketch of the source and the target object structure first, as a sample pattern according to which the specification should be made. The sketch would encompass sample object diagrams like those shown in Figure 8-7c and Figure 8-8d. Then, the developer should perform a non-trivial conceptual mapping from such a sketch into a linear code like the provided example, by writing the actions that traverse the source object structure, retrieve the information from it, create objects and links of the target structure, and set their attribute values. This conceptual mapping is a difficult, time-consuming, and error-prone mental process because it must produce a linear specification (the sequential code that includes both control-flow constructs and structure creation actions) for a non-linear problem.

Fortunately, this mental mapping can be avoided. Why should a user transform a non-linear sample object structure that describes the required transformation into a linear, textual form? Isn't it more convenient to define a non-linear object structure that looks like the sample object structure? Of course, that specification should have *creational semantics*, instead of being just an illustrative example. In other words, an object diagram with the creational instead of illustrative semantics may specify which objects and links should be created by the method. Additionally, some extensions to basic UML object specifications are needed because they must support conditional and iterative creation.

For that purpose, OOIS UML supports *creational specifications*, which are UML object specifications extended with some control features, such as conditions and iterations, and which have the formal creational semantics instead of illustrative semantics. This means that the creational specifications specify which objects should be created, how they will be linked, and what values will be set to their attributes. Creational specifications can be used to specify a creator instead of writing the code in a detail-level textual language.

Of course, creational specifications may be presented in UML object diagrams extended with the control constructs. According to the general philosophy of UML, the same object structure (that is, the creational specification of a creator) can be shown in many diagrams, each of which possibly rendering just some parts of the structure. Therefore, one creational specification is a set of specifications for objects, attribute values, links, and control structures that form one isolated part of the entire model, and which is used to specify the method of an operation. It may be depicted in many UML extended object diagrams.

Figure 8-9 shows two diagrams of the same creational specification for the creator `ProductionOrder::createProducts` for which the code was shown. At first glance, even before all details are explained, the specification can basically be understood. Additionally, it really resembles the illustrative sample object structure in Figure 8-8d, which actually served as the template for its definition. Of course, the control structures, such as iterations, are added (rendered as rectangles with tabs having small circular arrows

Part III: Concepts

in the upper-right corners). You should compare the specification depicted in Figure 8-9 with the code given previously to see the correlating model elements and code fragments. The following discussion explains in detail all concepts used in creational specifications.

The Semantics of Creational Specifications

A creational specification indicates a set of action specifications, which, in turn, specify action occurrences that are fired at run-time. For example, an object specification within a creational specification indicates a Create Object action, a link specification implies a Create Link action, whereas an attribute value specification implies a Write Attribute Value action. In fact, creational specification is just an alternative, structural way of specifying actions within a method other than using a textual detail-level language.

Formally, both ways have the same semantics of the actions connected in a specific control structure that dictates their execution at run-time, when the method is executed. For example, the specification in Figure 8-9 has exactly the same run-time semantics as the presented code in the OOIS UML native detail-level language. Of course, the former can be obtained from the latter by an automatic code generator, or by manual coding.

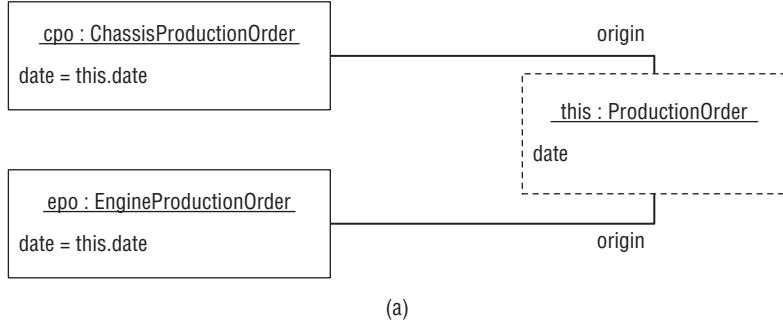
It should be noted that a single element specification (for example, an object or a link specification) may result in a set of executed actions for a single execution of the method because the specification may be placed within a loop, just as a statement within a loop in a classical code can be executed many times in a single activation of a method. Therefore, an *element specification* is clearly distinguished from what it results in at run-time.

All actions implied from one creational specification are considered as a constrained group, meaning that all implicit constraints are checked when the entire group of actions is completed. For example, multiplicity constraints of association ends are checked when the entire target structure is created according to the whole creational specification. Therefore, during the execution of the method, objects can go through inconsistent states that do not satisfy all constraints.

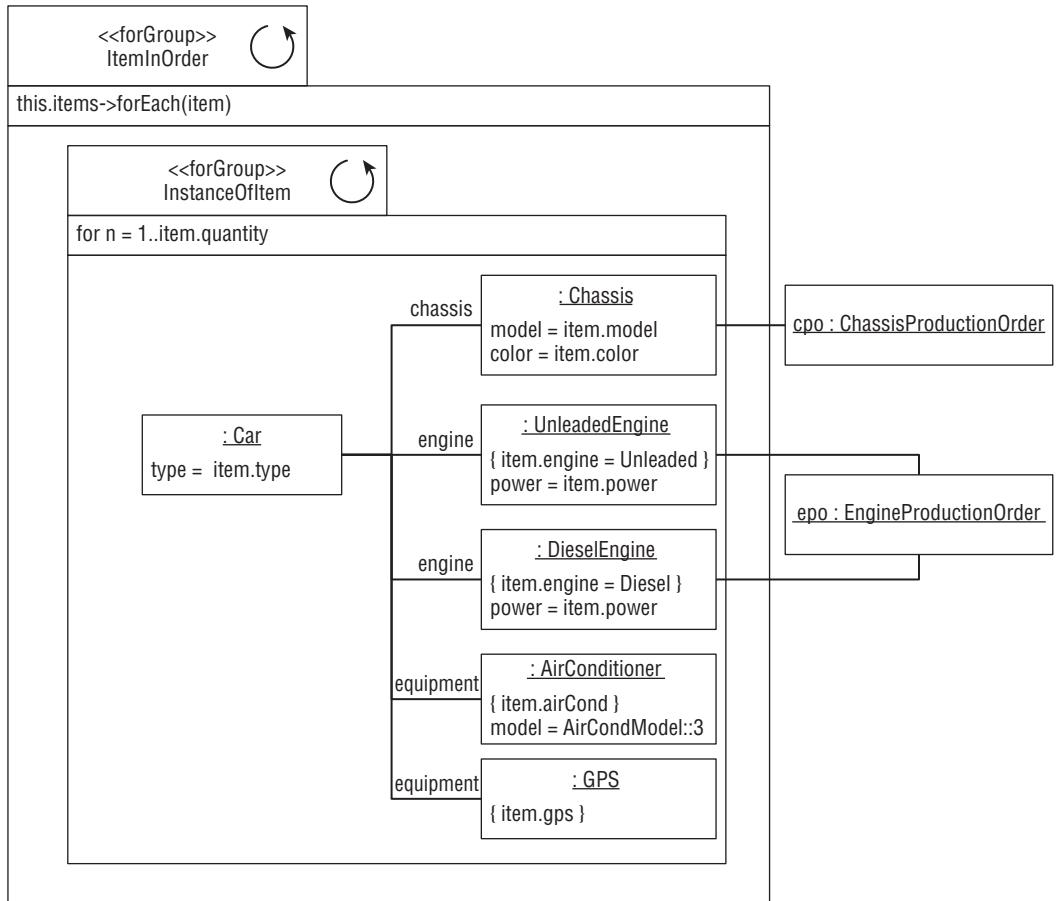
The semantics of each particular kind of specification is described in the section that follows. The section provides complete, unambiguous definitions of modeling rules and formal semantics of the presented concepts. If you are not interested in these details, feel free to skip this section and go directly to the later section, “Designing Creators by Demonstration.”

Section Summary

- ❑ *Creational specifications* are UML object specifications extended with some control constructs (such as conditions and iterations) that have formal creational semantics instead of illustrative semantics. Creational specifications can be used to specify creators.
- ❑ A creational specification implies a set of action specifications, which, in turn, specify action occurrences that are executed at run-time. All actions implied from one creational specification are considered as one constrained group of actions.



(a)



(b)

Figure 8-9: Car manufacturing example — the creational specifications for the transformation.
(a, b) The creational specification object diagrams that show how the structure of objects in the target domain should be created from the structure of objects in the source domain.

Part III: Concepts

Definitions of Creational Specifications¹⁴

This section provides precise descriptions of all kinds of elements that can be used in creational specifications, along with their modeling rules and semantics.

Object Reference Specifications

It is sometimes necessary to refer to an outer object from within a creational specification in order to create a link with it. For example, as shown in Figure 8-9a, the object specification named `this` of the class `ProductionOrder` refers to the object whose method is being executed when the operation is invoked at run-time. Actually, it refers to the production order for which the creator is activated, and for which the target object structure is being created by the creator under consideration.

In general, some object specifications in creational specifications do not have creational semantics, meaning that objects will not be created in the target structure when the method is activated. Instead, they *refer* to outer objects that already exist at the time when the method is being executed. Such object specifications are called *object reference specifications*. An object reference specification is a packageable element.

An object reference specification indicates a reference to a single object. Actually, it has the same semantics as a single-valued variable with multiplicity `0..1`, of the same type, and with the same name as that of the object reference specification. The reference is introduced as a named element into the first namespace enclosing the object reference specification. For example, the object specification depicted in Figure 8-10 is equivalent to the code in the OOIS UML native detail-level language:

```
Person[0..1] aStudent = this.students->at(i);
```

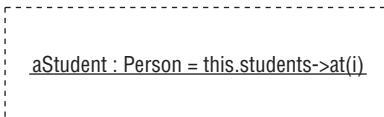


Figure 8-10: Object reference specification bound to an expression that evaluates to a single value with a conformant type

A reference is bound to the object in one of four possible ways:

- A reserved object reference specification named `this` will refer to the object that is the host of the invoked method (that is, for which the method's operation was invoked). The reference specification is of the same class to which the method belongs.
- An object reference specification can be bound to a single-valued input parameter of the method's operation. In that case, the reference will refer to the object that was supplied as the argument when the operation was invoked. Its name and class must be the same as those of the parameter.

¹⁴This is advanced reference material that can be skipped on first reading.

- ❑ An object reference specification can be bound to an arbitrary expression that results in a value of multiplicity and type conformant to the multiplicity and type of the object reference specification. The expression follows the same syntax as expressions in the OOIS UML native detail-level language. Alternatively, an expression in another detail-level language that is used for implementation can be specified for the same purpose. At each resolution of the object reference specification at run-time, the expression is evaluated, and the object reference is bound to the result of the expression. The class of the object reference specification must be accessible from the namespace owning the specification.
- ❑ An object reference specification can be bound to an output parameter of a Call Operation action from the same namespace (to be explained later). At each activation of the related Call Operation action at run-time, the object reference is bound to the output argument of the operation call. The class of the object reference specification must be accessible from the namespace owning the specification, and the output parameter must conform to the variable implied from the object reference specification.

The symbol for an object reference specification is the same as for an object specification in UML object diagrams, except that the outline of the rectangle is a dashed line. The name of the object reference and its class can (but need not) be shown within the rectangle in the usual manner. The expression to which the object reference specification is bound, if any, can be specified to the right of the name and class, after the = sign, as shown in Figure 8-10.

Object Specifications

An object specification has creational semantics, meaning that for each resolution of the object specification at run-time, an object will be created in the target structure when the method is activated. In effect, an object specification implies a Create Object action.

Consequently, a constructor is invoked at the time the object is created. As for the Create Object action, it can be either a default constructor, or a constructor that accepts a set of arguments. The actual arguments are specified as arbitrary expressions that follow the same syntax as expressions in the OOIS UML native detail-level language. Alternatively, an expression in another detail-level language that is used for implementation can be specified for the same purpose. At each resolution of the object specification at run-time, the expressions are evaluated, and the actual arguments are bound to the results of the expressions.

An object specification also implies a reference to the single object that has been created by the action. Actually, it has the same semantics as a single-valued variable of the same type and with the same name as those of the object specification. The reference is introduced as a named element into the first namespace enclosing the object reference specification. For example, the object specifications depicted in Figure 8-9a are equivalent to the code in the OOIS UML native detail-level language:

```
ChassisProductionOrder[1] cpo = new ChassisProductionOrder;  
EngineProductionOrder[1] epo = new EngineProductionOrder;
```

Object specification is a packageable element. The class of an object specification must be accessible from the namespace owning the specification.

The symbol for an object specification is the same as for an object specification in UML object diagrams. The name of the object and its class can (but need not) be shown within the rectangle in the usual manner. The list of expressions for the actual arguments of the constructor (if any) can be specified to the right of the name of the class, enclosed in parentheses.

Part III: Concepts

Attribute Value Specifications

An object specification or an object reference specification can have an arbitrary number of *attribute value specifications*, which are actually specifications of the corresponding Write Attribute Value actions that will be explained in more detail in Chapter 9.

At run-time, when the object specification is resolved into a Create Object action, the associated Write Attribute Value (actually, the Set Attribute Value) actions are executed. They modify the attribute values of the owner object. For example, the attribute value specifications depicted in Figure 8-9a are equivalent to the code in the OOIS UML native detail-level language:

```
cpo.date = this.date;  
epo.date = this.date;
```

This also holds true for attribute value specifications in object reference specifications: the attributes of the object referred to by the resolved reference will be modified at run-time.

Attribute value specifications follow the syntax used for the Set Attribute Value actions in the detail-level language used. The attributes referred to by an attribute value specification must be accessible from the first namespace enclosing the owner object (reference) specification.

Link Specifications

The existence of a *link specification* in a creational specification implies a Create Link action. For each tuple of objects that result from the object specifications at the link specification's ends, a link will be created at run-time. In other words, if a link specification crosses the boundary of an iteration group, links will be created for all objects created in the iteration. For example, for the specification in Figure 8-9b, a link will be created between the object `cpo` and every object of `Chassis` created in the nested iteration, as well as between the object `epo` and every object of `UnleadedEngine` and `DieselEngine`. Of course, a link will be created only if the objects at the ends exist.

A link specification must define the association whose link will be created and the orientation (that is, the association ends to which the objects linked by the link correspond). The association must be accessible from the namespace owning the link specification. A link specification belongs to the first common namespace enclosing all objects at its ends. Additionally, the association and the orientation of the link specification must be consistent with the class model, meaning that the classes (or their superclasses) of the linked objects must be related with that association in the appropriate orientation.

Note, however, that this does not mean that the modeler must explicitly specify the association and the orientation of a link specification. In the cases when this can be done unambiguously, the modeling tool can infer these aspects from the class model.

For example, if the objects at the link specification's ends belong to the classes that have only one connecting association, the association of the link can be inferred unambiguously. Similarly, if an object at a link's end can exist at only one of its ends according to the class model, the corresponding association end is unambiguously inferred. In all other cases, the modeler must resolve ambiguities by explicitly defining the association of the link and/or its orientation.

The symbol for a link specification in creational specifications is the same as for link specifications in UML object diagrams. It is a solid line connecting the symbols for objects, possibly consisting of several segments. The name of the association can (but need not) be written and underlined somewhere near the middle of the line, but not too close to an end to be confused with the name of the end. The name

of an association end can (but need not) be shown near the corresponding end of the line, as shown in Figure 8-9b.

Group Specifications

Sometimes it is necessary to attach the same condition to a group of objects and other specifications, or to group the specifications for other reasons (such as better organization and decomposition of the creational specification). For that purpose, *group specifications* exist in OOIS UML. A group specification is a general grouping mechanism of other creational specifications, including nested groups.

Figure 8-11 illustrates an example from the car-manufacturing sample application. Assuming that the variable `limit` represents a limit to the number of cars of one type that can be produced for one production order (because of production capacity limitations), the entire group `ProductionLimit` of object and link specifications will result in actions only if the result of the condition attached to the group is true.

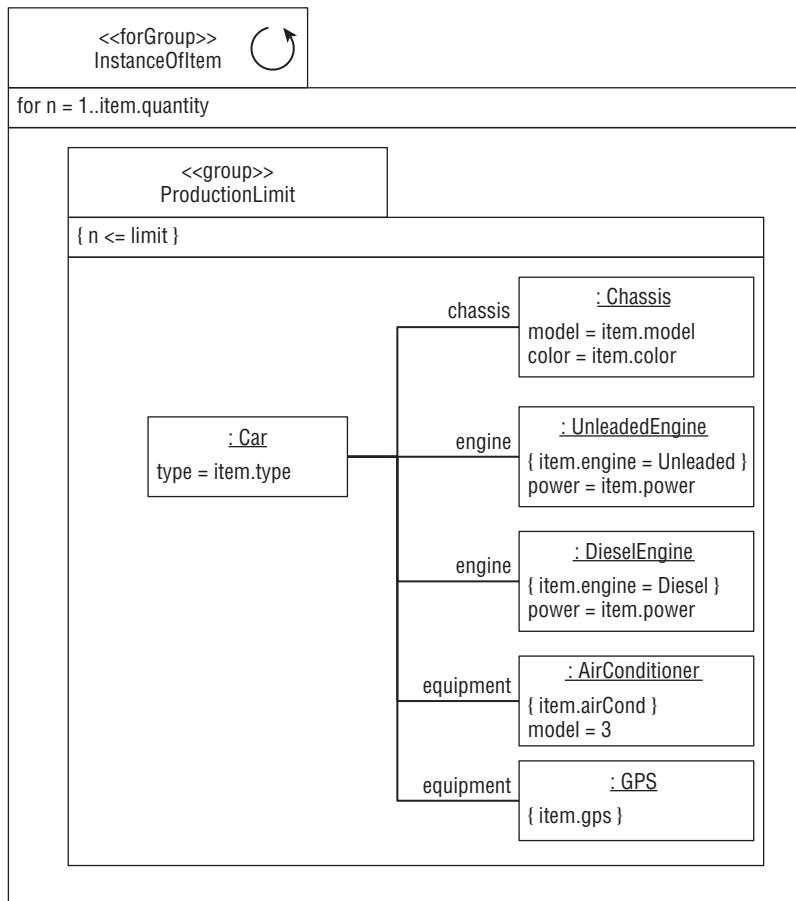


Figure 8-11: Group specification. Assuming that `limit` represents a limit to the number of cars of one type that can be produced for one production order, the entire group `ProductionLimit` of object and link specifications will result in actions only if the result of the condition attached to the group is true.

Part III: Concepts

A group specification is a kind of package, but one that can be used only inside creational specifications, and one that has creational semantics. As a package, a group specification possesses all properties of packages. A group specification can own other packageable elements, including nested groups. This way, a creator method can be hierarchically decomposed into groups to improve its organization.

A group specification is a namespace, capable of owning or importing other named elements as its members with the appropriate visibility. By default, the visibility of an element is public. A link specification that connects object (reference) specifications from different groups belongs to the first common enclosing group of the groups immediately owning the object (reference) specifications at its ends. The object (reference) specifications at the ends of a link specification must be accessible from the group owning the link specification.

As already described, the creational specifications that are elements of a group specification may introduce expressions for different purposes. Constraints can be attached to them, or expressions can be defined for object reference specifications or for iterations. These expressions are resolved in the scope of the owner group being a namespace, meaning that the named elements used in the expression are resolved according to the general naming rules of OOIS UML.

A method that is specified by means of creational specifications is actually defined by a topmost group specification that defines that method. This group specification is a namespace nested in the namespace of the operation it defines. Consequently, a group specification is analogous to a block of statements in block-structured conventional textual programming languages (although it does not imply any sequential execution of its contents, as in most conventional programming languages). Just as a block of statements can be a branch of a condition, or a body of a loop, or nested within another block, a group specification can have similar purposes. Additionally, just as the outer-most block defines the body of a method in conventional programming languages, a creator method can be specified by an outer-most group specification.

The run-time semantics of a group specification are also analogous to the conventional blocks of statements. All owned members of the entire group have or do not have the defined run-time effects, depending on the entire group. For example, if a condition attached to the group returns false, the owned members of the group will not have any effect. Note that imported elements and owned elements that are not creational specifications do not have run-time effects — they have the usual static semantics defined for elements of packages.

Group specifications are rendered in diagrams as packages, but with the keyword `<<group>>` shown above the name of the group. The attached constraints can be shown in a separate compartment of the large rectangle.

Groups are used for several purposes. First, as packages in general, they can be used to hierarchically decompose more complex creational specifications and possibly encapsulate some of their parts for better flexibility. Additionally (and more usually), group specifications are used as conditional creational structures. If the entire structure should be created depending on a condition, a group can be assigned a condition, as shown in Figure 8-11.

Iteration Specifications

Iteration specification is a kind of group specification that has iterative run-time semantics, meaning that the contents of the group will have a repetitive effect, one for each executed iteration of the loop specified by the iteration specification. For example, for each iteration of the loop specified by the group `InstanceOfItem` in Figure 8-11 (that is, for each value of `n` in the range from one to `item.quantity`) every creational specification owned by the group (the nested group `ProductionLimit` in this case) will be resolved once.

Iteration specification is a kind of group specification, which thus has all the static semantics as group specifications. The only difference is that the loop must be specified for an iteration specification. The loop can be specified as a loop statement in the OOIS UML native detail-level language, or in another detail-level language used for implementation. For example, the loop can be defined in the OOIS UML native detail-level language as follows:

- ❑ A `forEach` (or `forEachReverse` for ordered elements only) loop on a multi-valued variable, attribute, or association end that is accessible from the namespace owning the iteration specification. The iteration specification `ItemInOrder` shown in Figure 8-9b is a `forEach` loop.
- ❑ A counter loop that introduces a single-valued variable of type `Integer` that goes through the given range. The iteration specification `InstanceOfItem` shown in Figure 8-9b shows this type of counter loop.
- ❑ An expression resulting in an integer value. In that case, the loop will iterate as many times as the result of the expression indicates, if it is greater than zero. Some examples are shown in Figure 8-12.

For each iteration of the specified loop, the owned creational specifications of the iteration specification will be resolved once, having the effect defined for these creational specifications. For example, the object structure shown in Figure 8-12b will be created for the specification in Figure 8-12a.

The iterations will take effect one by one, sequentially, with the next one starting after the previous one has been completed. The ordering of the iterations is defined as follows:

- ❑ If the collection on which `forEach` operation is specified is ordered, the iterations will take place in the order of the collection's components. Otherwise, the ordering of iterations will be arbitrary (that is, implementation-dependent). For the `forEachReverse` operation, the ordering is reversed.
- ❑ For a counter loop, the loop goes from the first to the second boundary of the range. For example, if the first boundary is greater than the second boundary, the loop will go in the decreasing order of the iteration variable.
- ❑ For the loops with the number of iterations indicated, the ordering is arbitrary (that is, it is implementation-dependent).
- ❑ If the loop is specified in another detail-level language, the ordering depends on the semantics of that language.

A link specification can connect object (reference) specifications that belong to different groups or iterations — that is, it can cross boundaries of groups. Examples are shown in Figure 8-12c and Figure 8-12e. In that case, a link will be created for each tuple of objects created or referred to by the current resolution of the group that owns the link. Remember that this is the first common enclosing group of the groups owning the connected object (reference) specifications.

For example, if a link specification connects an object specification owned by an outer and an object specification owned by a nested iteration specification (as in Figure 8-12c), the link is owned by the very outer iteration. Therefore, a link will be created for the object resulting from the current iteration of the outer loop, and for all objects resulting from the inner loop for that outer iteration, and so for every iteration of the outer loop, as shown in Figure 8-12d.

If, however, a link specification connects object specifications owned by two independent iteration specifications (as shown in Figure 8-12e), the link is owned by the first group enclosing these two iterations.

Part III: Concepts

Therefore, a link will be created between all tuples of objects created from one resolution of that enclosing group, as shown in Figure 8-12f.

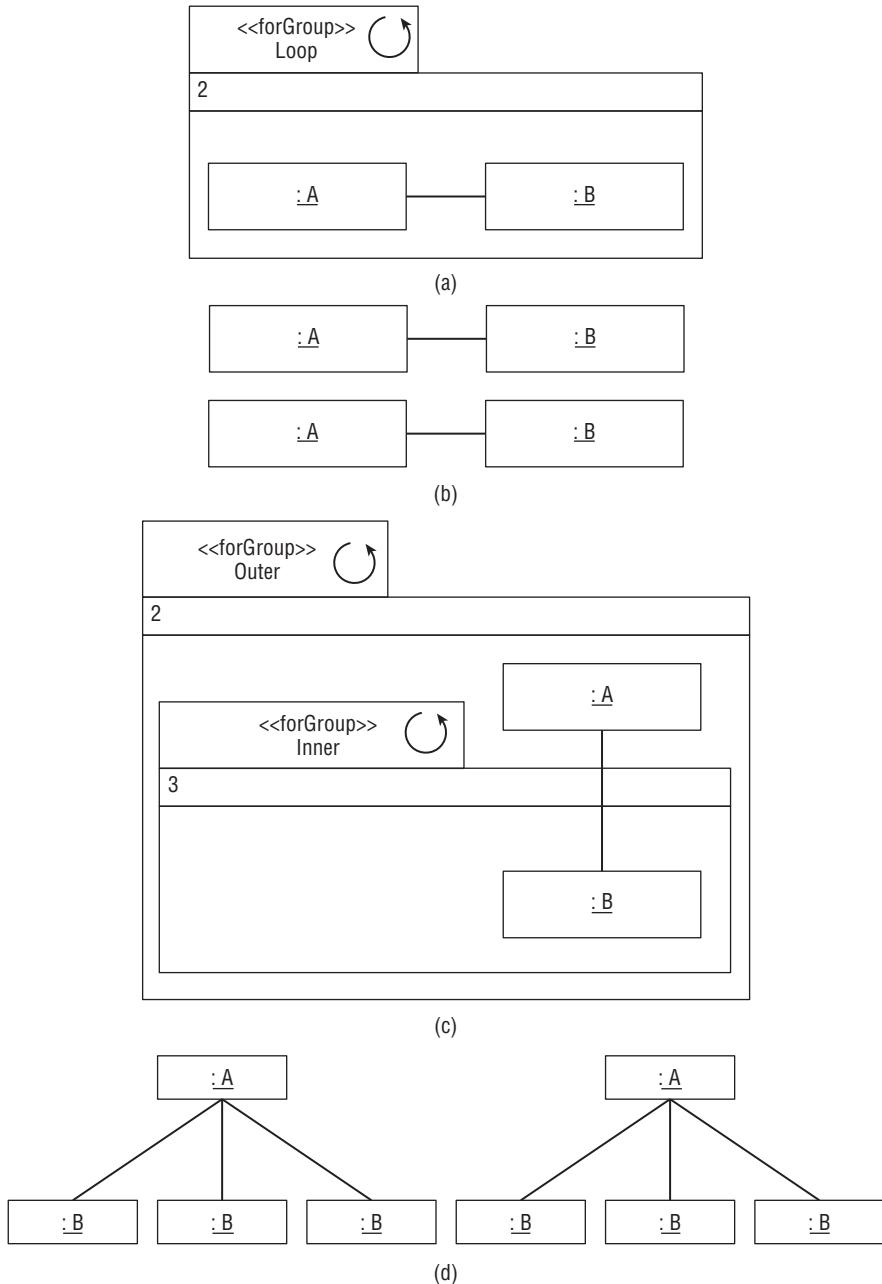


Figure 8-12: Iteration specifications with links that do or do not cross group boundaries. (a) An example of a link specification that does not cross boundaries and (b) its resulting object structure. (c) Nested iteration specifications and (d) their resulting object structure.

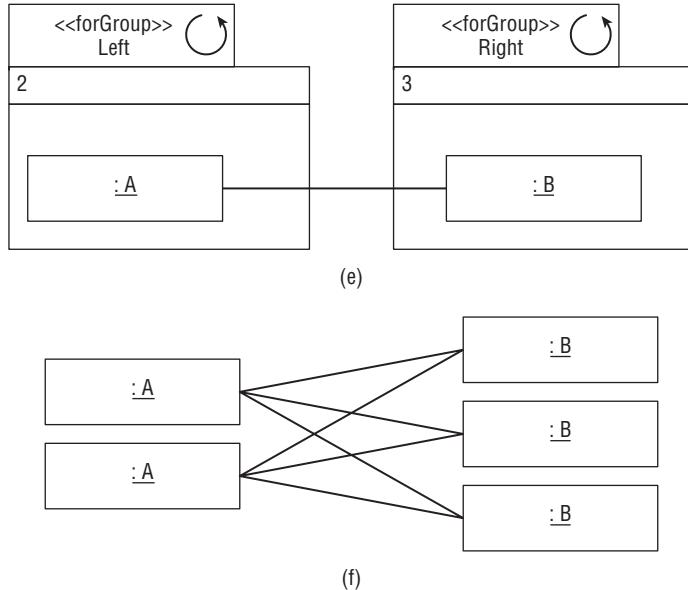


Figure 8-12: (e) Independent iteration specifications and (f) their resulting object structure.

Iteration specifications have the same notation as group specifications, except that the keyword `<<forGroup>>` is used, and a small circular arrow is depicted in the handle of the package symbol. The definition of the loop can be shown in a separate compartment of the large rectangle, along with the attached constraints.

Conditions

To all described kinds of elements of creational specifications — that is, to an object, object reference, attribute value, link, or group (including iteration) specification, an arbitrary number of *conditions* can be attached. A condition is a constraint attached to the element that is resolved at run-time at each resolution of the element. Only if the results of all attached constraints are true, will the element result in an activation of the corresponding action. This means the following:

- An object will be created as a result of an object specification only if the attached constraints evaluate to true. Therefore, the variable implied from an object specification has multiplicity 1 if the object specification has no attached constraints, and 0..1 otherwise. If the constraints evaluate to true, the object is created and the implied variable refers to it. Otherwise, the variable has no value.
- The expression of an object reference specification will be evaluated only if all conditions attached to the object reference specification evaluate to true. Otherwise, the variable implied from the specification has no value.
- The Write Attribute Value action will be activated as a result of an attribute value specification only if the constraints attached to the attribute value specification evaluate to true and if the variable implied from the owning object (reference) specification has value.

Part III: Concepts

- ❑ A link will be created as a result of a link specification only if the constraints attached to the link specification evaluate to true and if all variables implied from the object (reference) specifications at its ends have values.
- ❑ A group specification will be resolved only if all conditions attached to the group result to true.

As constraints in OOIS UML, conditions can be specified as expressions in OCL, the OOIS UML native detail-level language, or another detail-level language used for implementation. They are rendered in diagrams like other constraints in UML.

Call Operation Actions

Sometimes it is necessary to incorporate a substructure created by another creator into the structure being constructed by the creator under consideration. Because creators are just methods activated on operation invocations, in general, it might be useful to invoke an operation within a creator, to supply references to the objects of the structure being created as input arguments of the call, and to refer to the call's output arguments within the structure.

This can be done by introducing Call Operation actions in creational specifications. Like anywhere else, such an action specified within a creational specification results in an operation call at run-time, when the action is fired. Of course, input and output arguments of the operation call can be specified within the caller. Figure 8-13a shows an example. The Call Operation action is rendered as a solid-outline rounded-corner rectangle with the target object, operation name, and argument bindings written in it. This action will result in an invocation of the operation `anObject.aCreator` at run-time. The input argument `in1` of that invocation will refer to the object created from the object specification `ob1`, the input argument `in2` will refer to the same object as the reference `ref2` refers to, and, after the call returns, the reference `ref3` will have the value of the output argument `out3`. This way, the invoked operation can create or address a certain substructure from the entire object space, and the caller can refer to that substructure in the rest of the specification.

A Call Operation action has static semantics and conforms to modeling rules as if it existed within a method definition. The target object of the call and the operation to which the action is bound must be specified for the action. The target object of the call can be specified by an expression in the detail-level language. The input parameters of the target operation must be bound to expressions that could be resolved within the namespace enclosing the action. These expressions are written in the detail-level language. Alternatively, an input parameter of the operation can be explicitly bound to an object (reference) specification owned by a group directly or indirectly enclosing the action, where the variable implied from that specification must conform to the parameter. Finally, an object reference specification owned by the same group that owns the action can be bound to an output parameter of the operation, provided that the parameter conforms to the variable implied from the object reference specification.

The run-time effect of a Call Operation action is as defined for that action later in this book. In brief, the input arguments of the operation are resolved, then the call is resolved to the appropriate method, and the method is activated. When the method completes, the output arguments are provided.

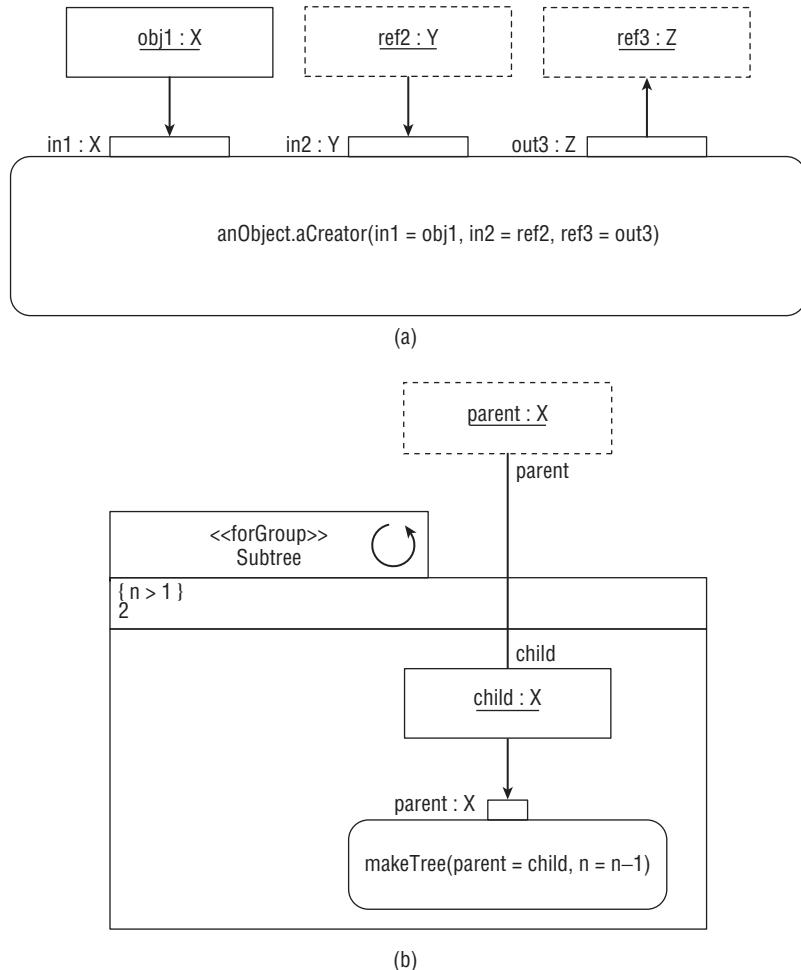


Figure 8-13: Call operation actions within creational specifications. (a) An example of an operation invocation with binding of input and output parameters. (b) An example of a recursive operation invocation that forms a binary tree of the given depth. The entire specification is for the method of the operation `X::makeTree(parent:X[1], n:Integer[1])`.

In creational object diagrams, a Call Operation action is rendered as a solid-outline, rounded-corner rectangle. The target object of the call, the operation, and optionally the argument bindings can be shown in the center of the rectangle, as shown in Figure 8-13a. The parameters can be optionally rendered as small rectangles placed on the border of the rounded rectangle, with labels placed near the small rectangles and showing the details about the parameters in the UML textual syntax for operation parameters. A binding of a parameter to an outer object (reference) specification is shown as an arrow pointing in the direction of the flow of the value, as shown in Figure 8-13a.

Part III: Concepts

As already indicated, Call Operation actions are used within creational specifications to address a substructure obtained by an operation call. As with any other operation call, this call can be polymorphic and even recursive. Using polymorphism and recursion can produce interesting effects from very simple and concise creational specifications. For example, assuming that the specification shown in Figure 8-13b defines the method of the operation `X::makeTree(parent:X[1], n:Integer[1])`, the recursive specification shown in that figure defines the method that creates a binary tree of depth n , having objects of the class `X` as its nodes, with the given root. It is left as an exercise to the reader to analyze and understand this concise specification.

Ordering of Resolution and Sequences

As you may have noticed, elements of creational specifications (which actually behave at run-time as actions within a method) do not introduce many assumptions about the sequential ordering of execution of those actions. For example, two independent object specifications result in execution of two Create Object actions at run-time in an arbitrary order — any of them can be executed first. In other words, there are no implicit total sequential ordering rules for the execution of actions (that is, the ordering is implementation-dependent).

However, there are certain ordering rules that are implied from the semantics of creational specifications and that define partial ordering of resolution of the owned members of one group specification:

- If an expression defined for a specification depends on a variable implied from another specification from the same group, the former specification will be resolved after the latter. This holds true for expressions defining actual arguments of operation calls, constructors, attribute value specifications, conditions, loops, and so on. For example, if an argument of the constructor for an object specification `os` addresses an object reference specification `ors`, then `os` will be resolved (into activating a Create Object action) after `ors` is resolved. These dependencies are determined at compile time, when the model is checked for consistency. The dependencies determine the implicit partial ordering among the dependent specifications. No circular dependencies are allowed, or the model is ill-formed.
- An attribute value specification is resolved after the owning object (reference) specification is resolved.
- Owned groups (including iterations) are resolved after the owned object specifications, object reference specifications, and operation calls are resolved. Owned groups are resolved recursively according to these same rules.
- Owned links are resolved after all other owned specifications are resolved.

There are no other implicit ordering rules — unless ordered by one of these rules, specifications can be resolved in any sequential order.

However, sometimes it is necessary to enforce a certain sequential ordering between two independent specifications that would be otherwise unordered. This can be the case when the resolutions of these specifications have some side effects because of which the ordering of these effects is important. For that purpose, explicit ordering can be introduced by *sequencing* two specifications. Figure 8-14 shows an example. The arrowhead indicates an explicit sequential ordering of resolution of the object specifications `ch` and `ac`. The object specification `ch` will be resolved before the specification `ac`.

A sequencing relationship can relate any two elements that imply actions at run-time owned by the same group specification. These are an object, object reference, link, group specification, or Call Operation

action. A sequencing relationship introduces an explicit ordering rule between the resolutions of these two elements at run-time.



Figure 8-14: Explicit sequential ordering between two implicitly unordered creation specifications owned by the same group specification. The object specification ch will be resolved before the specification ac.

A sequencing relationship is shown as an arrow pointing to the element that will be resolved later.

Section Summary

- ❑ An *object reference specification* indicates a reference to a single object, which can be the following:
 - ❑ The reserved reference named `this`, which always refers to the object hosting the execution of the method
 - ❑ Bound to a single-valued input parameter of the method
 - ❑ Bound to the single-valued result of an expression of a conformant type
 - ❑ Bound to a conformant output parameter of a Call Operation action from the same namespace
- ❑ An *object specification* indicates a Create Object action, including the invocation of a constructor.
- ❑ An *attribute value specification* is a specification of a Set Attribute Value action that modifies the value of an attribute of the object into which the enclosing object (reference) specification resolves.
- ❑ A *link specification* indicates a Create Link action. For each tuple of objects that result from the object specifications at the link specification's ends, a link will be created at run-time.
- ❑ A *group specification* is a kind of package that has creational run-time semantics.
- ❑ An *iteration specification* is a kind of group specification that has repetitive creational run-time semantics.
- ❑ To an object, object reference, attribute value, link, or group specification, an arbitrary number of *conditions* can be attached. A condition is a constraint

Continued

attached to the element. Only if the result of all attached constraints is true, will the element result in an activation of the corresponding action.

- ❑ *Call operation action* can be used within a creational specification to indicate an operation invocation at run-time, with optional binding of input and output arguments to the object (reference) specifications from the enclosing namespaces.
- ❑ Apart from implicit partial ordering of resolution of dependent specifications, an explicit ordering can be introduced by a *sequence relationship* between two members of the same group specification.

Designing Creators by Demonstration

Creational specifications provide an expressive and effective way to design complex structural transformations. However, in practice, most creators include very simple transformations. For example, a creator simply creates an object, possibly sets some of its attribute values, and links the created object to one or two other objects provided as parameters. For that purpose, designing creational specifications introduces an unnecessary development overhead because of the traditional, roundtrip programming approach: design, compile, execute, test, and modify the specification if the result of the test is not satisfactory.

Instead of taking this long roundtrip, the development of such simple creators may take a more direct and quick route — *demonstration*. For example, in order to specify a creator that creates an object of a class *X*, sets its attribute *a* to a value *v*, and links that object to a property *f* of a parameter *p* of the creator, the modeler can demonstrate the behavior of the method by issuing the intended commands at run-time, using the generic GUI of an OoIS UML framework, just as the user would do in a concrete use case.

This demonstration might look like this:

- ❑ The framework offers an option to specify a method for an operation $\text{Y} :: \text{op}(\text{p} : \text{Z})$. The modeler selects this option to specify the creator.
- ❑ The framework opens a working environment that looks the same as the generic GUI, whereby an object named *p* of type *Z* becomes accessible and represents the parameter.
- ❑ The modeler creates an object of the class *X* using a generic command for that. The new object appears in the screen.
- ❑ The modeler selects the new object of *X* and writes the desired value *v* to its attribute *a*, again using the generic command or dialog for that.
- ❑ The modeler creates a link between the new object of *X* and the object *p* (for example, by dragging the first and dropping it onto the property *f* of the latter).
- ❑ The modeler selects an option of the environment that indicates that the demonstration session is completed. The framework can then interpret the recorded demonstration and the method is specified.

It is important to note that the result of demonstrating a creator is always a specification that has exactly the same semantics as an equivalent creational specification made at design time. This result will consist

of the specifications of actions (such as Create Object, Set Attribute Value, Create Link, and so on) in the proper order and with the proper parameters provided interactively during the demonstration.

The internal representation of such a result is framework-specific, though. However, it can provide a means to correct or modify the demonstrated creator later. For example, one reasonable approach is to store the specification as the very object structure created by the demonstration. The environment can then opt to interpret this structure in terms of issuing the corresponding actions when the method is activated at run-time. Alternatively, it may opt to generate the code for the method in the detail-level or another implementation language when the model is compiled.

It is relatively straightforward to implement the support for simple creators that do not contain control structures (such as conditions, loops, or recursions). On the other hand, in order to specify these complex control structures, the framework must exhibit *inference* to some extent, possibly requiring intervention of the modeler. For example, in order to specify a loop, the modeler may repeat a certain action several times; the framework may infer that the modeler assumes a repetition, suggest a generalization into a loop, and ask the modeler to confirm that inference.

However, such an approach obviously assumes the use of heuristics and cannot be predefined by a formal method as OOIS UML tends to be. It is, therefore, left to the implementation to what extent it will incorporate the inference in modeling by demonstration. Moreover, it seems to be impractical to use demonstration for specifications that incorporate more complex control structures. Instead, demonstration should be used as a shortcut to capture the most simple (and, fortunately, the most frequent) cases. For more complex cases, creational specifications should be used.

It is not an objective of the OOIS UML profile to impose a “one-size-fits-all” method (for example, demonstration), but to provide many alternative and complementary ones, each of which has its competitive advantages over the others for certain cases. The modeler should select the method that best suits the considered problem and that allows him or her to find a solution in the most efficient way.

Consequently, OOIS UML does not define a set of features that would be completely expressive to allow demonstrating every possible creator. Instead, it requires that a framework provide the means to demonstrate the most simple creators, at least those that contain the following creational specifications (that is, actions):

- ❑ **Object reference specification** — This should be supported at least for input parameters. As described, an input parameter of the operation can be represented as an object of the appropriate type accessible within the demonstration session.
- ❑ **Create Object action for creating a new object of a class with invoking its default constructor** — As described, this could be demonstrated as in the generic GUI of an OOIS UML framework by issuing the Create Object command.
- ❑ **Set Attribute Value action** — This could be supported by at least two options: writing a constant value to a certain attribute, or copying the value from another attribute of the same specification. For example, to specify a value to be written to an attribute by demonstration, the modeler may simply write that value through the generic GUI (in which case the resulting specification assumes a constant value), or copy and paste (or drag and drop) a value from another attribute (in which case the demonstration would result in a Set Attribute Value action with the value taken from the source attribute).

Part III: Concepts

- ❑ **Create Link action** — As already described, this can also be demonstrated by creating a link between objects as in the generic GUI of an OOIS UML framework (for example, by dragging and dropping an object onto a property of another object within the same demonstration session).

Section Summary

- ❑ Instead of traditional modeling, creators can be designed by *demonstration*.
- ❑ Demonstration can be used to design simple creators, without complex control structures, but including only the following:
 - ❑ Object reference specifications that refer to input parameters of the method
 - ❑ Create Object actions with default constructor invocation
 - ❑ Set Attribute Value actions writing constant values provided interactively, or copying values from other attributes
 - ❑ Create Link actions

Destructors¹⁵

Completely analogous to initialization, sometimes there are activities to be performed during the *finalization* of objects — that is, whenever an object of a class is being destroyed. For that purpose, *destructors* are available in OOIS UML. In general, destructors are completely equivalent to constructors in OOIS UML.

A destructor is a specific operation of a class that is implicitly invoked during every finalization of object, that is, as the first step in execution of the Destroy Object action.

In OOIS UML, any operation of a class that has the name *destructor* is a destructor. A class can have an arbitrary number of destructors. These destructors have a different number and/or types of parameters, thus allowing different means of object finalization.

For example, two destructors of the class *Department* in Figure 8-6b can exist to provide different ways of finalization:

- ❑ **A destructor that accepts no parameters** — This destructor will simply do nothing.
- ❑ **A destructor that accepts a Boolean parameter** — This indicates whether all employees currently assigned to that department will be also destroyed; its specification can be the following.

```
Department::destructor (toDeleteMembers : Boolean[1])
```

¹⁵This is exclusively an OOIS UML concept. It does not exist in standard UML.

Its method (coded in the OOIS UML native detail-level language) can simply be the following:

```
if (toDeleteMembers)
    members->forEach(m)
    m.destroy();
```

Every class always has at least one predefined destructor that has no parameters. This one is called the *default destructor*. The default (inherited) method for this destructor is empty (does nothing), but can be redefined in every class.

The destructor is invoked as the first step in the execution of the Destroy Object action. By default, if nothing is specified for the invocation of the Destroy Object action, the default destructor is invoked. Otherwise, if the caller of the Destroy Object action wants another destructor to be invoked, the arguments for that destructor must be provided with the action. For the previously mentioned example of destroying a department, the following action will call the default destructor:

```
aDepartment.destroy();
```

The following action will call the destructor that accepts a Boolean parameter:

```
aDepartment.destroy(true);
```

As you can see, the parameters of the destructor invocation are simply specified within parentheses of the `destroy` operation, as with any other operation invocation. In general, the resolution policy is the same as for constructors and all other operations, and will be explained in Chapter 13. Any other operation invocation will be explained later in the book for operations in general. Anyhow, if there is no destructor that matches the actual arguments specified in the action, the action is illegal and a compilation error occurs. Therefore, the set of destructors of a class specifies all allowable means of object destruction, and no other means are possible. For example, if the listed destructors form the full set of destructors of the class `Department`, including inherited ones, the following action would be illegal because of an invalid argument type:

```
aDepartment.destroy(25);
```

If a conventional OO programming language is used as the detail-level language, the syntax can be exactly the same (for example, in Java and C#) or very similar as in the OOIS UML native detail-level language. This is the same as the previous example, but written in C++:

```
aDepartment->destroy(true);
```

Destructors are equal to ordinary operations in all other ways, except that after their execution, the object is destroyed. They can be of any visibility kind, having the usual consequences on their accessibility from the place of their invocation (where the action `Destroy Object` is specified). Destructors are polymorphic operations, meaning that they (as operations) are inherited in derived classes, and their methods are inherited by default, but can be redefined in derived classes.

Unlike in some other traditional OO programming languages, the redefined method of a destructor in the derived class does not implicitly call the method from the base class. This must be explicitly ensured if needed. Interestingly, destructors may also be invoked explicitly, having the same effect as any other invoked operation for an object.

Part III: Concepts

Destructors are operations that are called for objects when they are being destroyed, with the purpose of having some effect on the object structure or the system's environment. Because data types are destroyed implicitly, at an unknown time, and the operations of data types should not have side effects — data types in OOIS UML do not have destructors.

Section Summary

- ❑ A *destructor* is an operation of a class that is implicitly invoked as the first step in execution of the Destroy Object action.
- ❑ Any operation of a class that has the name *destructor* is a destructor. A class can have an arbitrary number of destructors with different number and/or types of parameters.
- ❑ Every class always has the default destructor, which has no parameters. The default (inherited) method for this destructor is empty (does nothing), but can be redefined in every class.
- ❑ If the caller of the Destroy Object action wants a destructor other than the default one to be invoked, the parameters of that destructor must be provided with the action. The appropriate destructor must exist in the class (possibly inherited). Otherwise, the action is illegal.
- ❑ Destructors are, in all other ways, equal to ordinary operations, except that after their execution, the object will be destroyed implicitly.
- ❑ Data types do not have destructors in OOIS UML.

Propagated Destruction of Objects¹⁶

Let's consider a somewhat simplified model fragment of a librarian information system shown in Figure 8-15a. The library may store many *book titles*, each of which is characterized by its author, international book number (ISBN), date of issue, and so on. For each of the titles, there may be many *book samples*, which are physical copies of the same title that can be lent to the library *members*. Each lending of a book sample to a member is recorded in the system, as shown in Figure 8-15a.

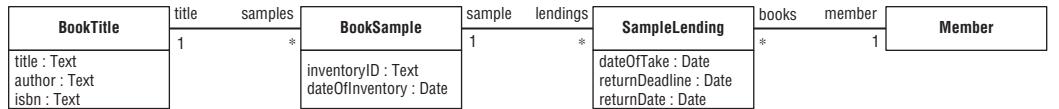
It is now a question of what happens when a book sample is to be discarded from the library (that is, the object of *BookSample* is to be destroyed). Obviously, all linked objects of *SampleLending* should be destroyed, too, because they cannot exist independently from their book samples. Therefore, when an object of *BookSample* is being destroyed, the destruction should be *propagated* to all objects of *SampleLending* linked on the *BookSample* object's property *lendings*. This can be coded within the method of a destructor of the class *BookSample* as follows:

```
lendings->forEach(len) len.destroy();
```

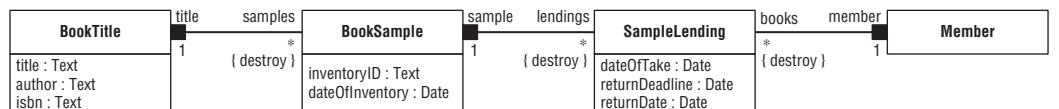
¹⁶This is exclusively an OOIS UML concept. It does not exist in standard UML.

This also holds true for the destruction of an object of `BookTitle`. If it is destroyed, all linked book samples of that title should also be removed from the system because they cannot exist independently. Consequently, a piece of code in a destructor of `BookTitle` can handle this propagated destruction:

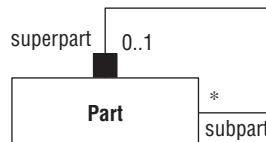
```
samples->forEach(sam) sam.destroy();
```



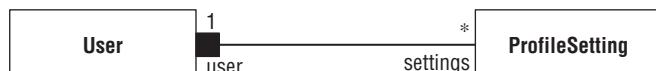
(a)



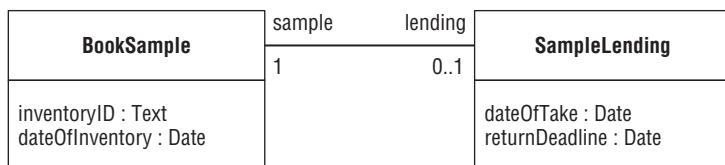
(b)



(c)



(d)



(e)

Figure 8-15: Propagated destruction. (a) A part of the conceptual model of a librarian information system. (b) The same model with declarative propagated destruction. (c) A part of the conceptual model of a spare-part management system with declarative propagated destruction. (d) A part of the conceptual model of a user management system with declarative propagated destruction. (e) A modified conceptual model of the librarian system.

Propagated destruction is very frequently needed in applications. Although it can be implemented by explicit, imperative actions within a destructor of a class, that approach may sometimes be tedious and less clear to maintain. Therefore, OOIS UML supports another, *declarative* way of specifying propagated destruction. It is shown in Figure 8-15b for the library example.

An association end (for example, `samples` in this case) that is owned by a class, and that results in a property of that owner class (`BookTitle`, in this case, because the association end `samples` results in a

Part III: Concepts

property of that class), can be tagged as being *propagated destruction*. In that case, when an object of the class owning the property (`BookTitle` in this case) is being destroyed, all objects linked to the property's slot are also destroyed (book samples in this case). This propagated destruction is performed by means of propagating the Destroy Object action to all linked objects, and takes place after the destructor of the former object is finished and before the other links of that object are destroyed.

The association end with declarative propagated destruction is tagged with the keyword `{destroy}` in diagrams. An alternative notation (which may co-exist with the previous one) is a small filled square at the opposite association end, as shown in Figure 8-15b.

Several things should be noted here. First, the declarative propagated destruction can be specified only for the association ends owned by classes. Consequently, it is allowable only for binary associations (bilateral, between two classes), and is not available for N-ary associations (associations between more than two classes).

Second, propagated destruction can be transitive because it is performed over the Destroy Object action to the linked objects, which may also encompass propagated destruction. For the given example, when a book title is destroyed, it propagates the destruction to all its book samples, which in turn also propagate the destruction to all linked sample lending records.

Third, both imperative and declarative propagated destruction may coexist. For the given example, it is completely harmless and has the same ultimate effect if the destructor of the class `BookTitle` still contains the explicit actions for propagated destruction and the association end `samples` is tagged with `{destroy}`. Because the destructor is invoked before the implicit propagated destruction takes place, it will first propagate the destruction to the linked objects of `BookSample`, and thus destroy these objects along with their links. After that, the declarative propagated destruction will have nothing to do because no objects will be linked to the property `samples`. More precisely, the following steps will occur when a Destroy Object action is executed for an object x of `BookTitle`:

1. The destructor of x is invoked. Its method propagates the destruction to all objects linked to `samples` by explicit execution of Destroy Object actions. These actions perform the entire process of destruction recursively (for example, they propagate destruction to sample lending objects), and ultimately result in destroying all these objects and their links to the object x .
2. The implicit, declarative propagated destruction over the `samples` property is activated. But because there are no more linked objects on this property, it has no effect. Otherwise, it would do the same as described in the previous step.
3. The remaining links of the object x are destroyed, if any.
4. The object x is destroyed.

Propagated destruction is used quite often in practice. Figure 8-15 shows several other examples. In Figure 8-15c, an excerpt of the conceptual model for a spare-part management system is shown. Here, when a part playing the role of super-part is destroyed, all its subparts should be destroyed, too. In Figure 8-15d, a part of the conceptual model for a user management system is shown. Here, when a user is removed from the system, the settings that make his or her profile should be removed, too.

In general, propagated destruction is just one kind of support for a broader relationship between classes known as *existence dependency*. Existence dependency denotes the situation when objects of one class

cannot exist unless some other objects of another or the same class also exist (that is, the lifetimes of the former are incorporated in the lifetimes of the latter). Propagated destruction covers one side of this constraint — it ensures that the objects die together, so that the existence-dependent objects cannot survive the death of their masters. At the opposite end of the dependent lifetimes — the dependent birth of objects must be ensured by explicit actions within some methods (for example, constructors) or by appropriate creators.

Note that the multiplicity constraint on an association end (in particular, its lower bound that is greater than zero) may (but need not) go along with propagated destruction at the opposite end. If a lower multiplicity bound of an association end is zero, the objects of the class at the opposite end (at least some of them) can be existence-independent.

The example with part-subparts hierarchies in Figure 8-15c proves this. At least the parts that have no super-parts do not existence-depend on other parts. However, propagated destruction is still present and ensures that subparts are destroyed when a super-part is destroyed. On the other hand, book samples are existence-dependent on book titles in Figure 8-15b because a book sample cannot exist without its title. The multiplicity constraint 1 at the `title` association end clearly indicates that a book sample must be linked to a book title throughout its entire life, and the propagated destruction ensures that it is destroyed along with the title.

These two examples cover situations in which (1) the lower multiplicity bound of an association end is zero, but propagated destruction exists at the opposite end and (2) the lower multiplicity bound of an association end is greater than zero, and propagated destruction exists at the opposite end (which is a full existence dependency). The third type of situation occurs when the lower multiplicity bound is greater than zero (for example, it is 1), but propagated destruction at the opposite end does not exist. It is possible, of course, and it would mean that the objects at the constrained association end simply cannot be destroyed because the multiplicity constraint would be violated. Should the object have been deleted, along with its link, but without propagating the destruction to the linked object, the linked object would remain without any links, thus violating the multiplicity constraint. Therefore, this is the means to prevent objects from existing unrelated (that is, to prevent the destruction of an object until there are other objects linked to it).

Many such examples exist in practice. For example, the librarian system can be modeled in a different way, as shown in Figure 8-15e. If only the currently open sample lending is modeled with a separate association, and all other lending objects from the past are either of no interest or modeled as before (with another association), then the multiplicity constraint 1 at the `sample` association end (because no propagated destruction exists) ensures that a book sample cannot be removed from the system until the member who borrowed it returns it. Interestingly, this would also prevent the book title from being destroyed until all its currently lent samples are returned, because of the propagated destruction from `BookTitle` to `BookSample`.

In short, a multiplicity constraint with the lower bound greater than zero at an association end can (but need not) imply the need for propagated destruction at the opposite end, and vice versa. This is why these two concepts exist separately. Additionally, multiplicity constraints have a somewhat passive role in the model because they ensure that some rules always (or in certain intervals of time) hold. On the other hand, propagated destruction, which encompasses (explicitly or implicitly) execution of actions, provides an active means to transform the system's object structure from one state satisfying the specified constraints to another such state. In other words, multiplicity constraints (as with all other constraints) are assertions on the static state of the structure. Propagated destruction, by virtue of implying execution of actions, provides an active means for dynamic transitioning between these static structural states.

Part III: Concepts

Sometimes it may happen that propagated destruction is needed from both sides of a single association, or that it makes a circle in a path of associations. For example, in the model shown in Figure 8-16, all three linked objects (the caller, the callee, and the conversation) should be destroyed if the destruction of any of them is required from outside. Because the destruction of one of them implies propagation of destruction to the others, which in turn propagates the destruction to the first one, the circular propagation of destruction may cause an endless recursion.



Figure 8-16: Circular propagated destruction. The caller, callee, and conversation objects should be destroyed when any of them is destroyed.

Fortunately, this is not the case in OOIS UML. If the destruction is propagated to an object already in the process of destruction, the propagated Destroy Object action on that object has no effect, and thus breaks the recursion. This holds both for imperative and declarative specifications of propagated destructions, and does not need any intervention of the modeler. In other words, a destructor of, for example, the Caller class may freely and unconditionally propagate the destruction to the linked Conversation object, and the latter may also unconditionally propagate the destruction to the former. The run-time environment will implicitly break the endless recursion. This feature spares the modeler from the tedious work of taking care of such issues, and prevents errors in such cases.

Section Summary

- ❑ Object destruction can be propagated to other linked objects as follows:
 - ❑ Imperatively, by explicit invocation of Destroy Object actions within a destructor
 - ❑ Declaratively, by tagging the association end with {destroy}
- ❑ OOIS UML implicitly breaks possible recursions in circular propagated destruction.
- ❑ *Existence dependency* assumes that an object cannot live without another living object. Propagated destruction is one kind of support for existence dependency. A multiplicity constraint of an association end with a lower bound greater than zero is another.

Data Types

In general, data types may have attributes of other data types, which may have their attributes, and so on. Consequently, instances of composite data types are complex, hierarchically structured values. However, such a composition must end in an elementary data type that is not decomposed further, but is

implemented with a type native for the given implementation programming language or platform. Such elementary data types are called *primitive data types*.

As already described, instances of data types represent identifiers of entities that exist outside the scope of the modeled system. Some sets of such entities are well-established in programming or mathematics, and are supported by common data types such as integers, floating-point numbers, strings, and so on. However, some sets of such entities are domain-specific and should not be mixed up with these generic sets. In such cases, a domain-specific set of entities may be modeled with an *enumeration*, which is a limited, user-defined set of *literals*.

An OOIS UML implementation may (and should) provide a library of *built-in* data types for common modeling purposes. Some of these types may be primitive (for example, integer, string, and so on), some of them may be enumerations (for example, Boolean), and some of them may be composite data types. These built-in data types are ready to use in the model directly (for example, for types of attributes, parameters, or variables), or they may be specialized or composed into other domain-specific data types. Data types defined by the modeler are often referred to as *user-defined* types.

Primitive Data Types

A *primitive data type* is a data type without any relevant substructure. That is, it has no attributes with the OOIS UML semantics, but its internal structure is under the control of the implementation or the target implementation language. The instances of a primitive data type, as with any other data type, are values that refer to mathematical or other entities defined outside the scope of the system modeled in UML. There can be algebra and operations defined outside of UML for a primitive data type (for example, mathematically). This may mean any of the following:

- ❑ The primitive data type is a built-in data type provided by the implementation, and has an internal representation (structure) that is completely hidden from the modeler. The internal representation is manifested only through the behavior of the data type (that is, through its public operations). It may be also supported in the GUI by the built-in widgets for entering and displaying the values in a human-readable form.
- ❑ The primitive data type is a wrapper around a data type native for the implementation programming language. In this case, the primitive data type does not contain any other attributes with the OOIS UML semantics, but contains only attributes of the types supported by the implementation programming language, whose semantics are defined by that language and are outside of UML.
- ❑ The primitive data type is directly taken from the implementation programming language and has the semantics of that native type.

For example, the built-in primitive data type `Integer` (which provides a set of usual mathematical operations on a subrange of integer values) may completely hide the implementation of its structure and methods, or may be simply a wrapper around the built-in Java, C#, or C++ native type `int`, as shown in Figure 8-17. On the other hand, the type `String` shown in Figure 8-17 is directly taken from Java in this case. Finally, the modeler can define a domain-specific primitive data type `SocialSecurityNumber`, and implement its structure and operations using the native programming language types. The OOIS UML implementation may pose some constraints or requirements on such data types taken from or implemented in the target programming language.

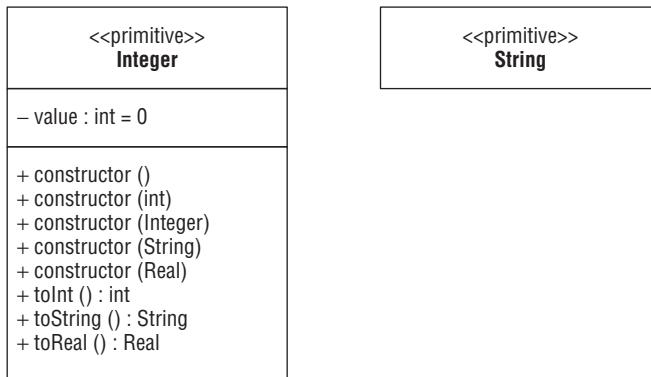


Figure 8-17: Examples of primitive data types

A primitive data type is shown as any other data type, but with the keyword `<<primitive>>` above or before the name of the primitive type, as shown in Figure 8-17.

Section Summary

- ❑ A *primitive data type* is a data type without any relevant substructure (that is, it has no UML attributes).
- ❑ There can be algebra and operations defined outside of UML for a primitive data type (for example, mathematically).

Enumerations

Let's consider an example where a data type should represent the status of a purchase order, which can be either "initiated," "processing," or "fulfilled." One approach is to select one of the available mathematical data types (such as `Integer`) and to encode the status with its values (for example, to encode the status "initiated" with 0, "processing" with 1, and "fulfilled" with 2). However, this might not be a good idea, for the following reasons:

- ❑ This is conceptually wrong, because the status of an order has nothing to do with the mathematical concept of integers. Integers are elements with an associated algebra (that is, for which operations such as addition, subtraction, and so on are defined), while an order status cannot be added or subtracted arithmetically.
- ❑ Choosing `Integer` for the type of an order status is confusing and blurs the real intention of the modeler. Moreover, the use of integer literals such as 0, 1, or 2 throughout the code of methods makes the code difficult to understand and maintain.
- ❑ Changing the design decision that `Integer` is sufficient and appropriate for modeling order status is problematic. Namely, if modelers conclude that integers are not appropriate for modeling

order status, they must search for all occurrences of literals 0, 1, and 2 in the model, especially in the code of methods, and to replace selectively and carefully these literals with something else. What is worse, an occurrence of the literal 0 may have nothing to do with the order status “initiated,” and might cause an error if inappropriately replaced.

In fact, the considered data type represents a finite set of predefined values. Such data types are called *enumerations*. An enumeration is a data type whose all allowable values are explicitly enumerated in the model as symbolic values called *enumeration literals*. For the given example, the enumeration OrderStatus may represent the set of symbolic values initiated, processing, and fulfilled, as shown in Figure 8-18. Another well-known example of enumeration is the built-in type Boolean, whose instances may take one of only two allowed values, false and true.

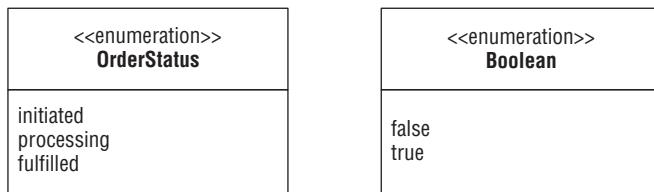


Figure 8-18: Examples of enumerations

Enumerations are special kind of primitive data types that do not have attributes. However, they may have operations like any other data type. Enumeration literals are named elements owned by the enumeration. Their names must be unique within the owning enumeration. In order to address an enumeration literal from outside its owner namespace (the enumeration), it must be qualified like any other named element. Instances of enumerations can be compared for equality and non-equality with enumeration literals. In fact, an enumeration literal in an expression or within a method represents (a reference to) an instance of the corresponding enumeration with the given value. Consider the following example:

```
OrderStatus status = OrderStatus::initiated;
...
status = OrderStatus::processing;
...
if (status == OrderStatus::fulfilled) ...
```

An enumeration is denoted as any other data type, but with the keyword `<<enumeration>>` above or before the name of the type. Enumeration literals are typically shown by their names, one in a line, in a separate compartment of the enumeration’s symbol, as shown in Figure 8-18.

Section Summary

- ❑ An enumeration is a data type whose allowable values are explicitly enumerated in the model as symbolic values called *enumeration literals*.

Built-in and User-Defined Data Types

An OOIS UML implementation should offer a library of predefined data types ready to use for common modeling purposes. These data types are called *built-in* types. The set of these types is not strictly prescribed, but an implementation should provide at least the following built-in data types:

- Boolean — Representing logical values (an enumeration with literals `false` and `true`).
- Text — Representing strings of characters of variable, but limited size.
- Integer — Representing integer numbers in a limited range.
- Real — Representing rational floating-point numbers in a limited range and with limited precision.
- TimeInterval — Representing time intervals of a limited size.
- Time — Representing the time of a day (00:00 to 23:59).
- Date — Representing a date.
- DateTime — Representing a point in time, which includes both date and time.
- File — Representing binary contents of variable and (virtually) unlimited size, which the system may or may not be able to interpret. `File` corresponds to the traditional BLOB (binary large object) type in relational databases.

Apart from these basic types, the implementation may provide more advanced built-in types, which are most often specializations of the listed types. However, as separate types, they may add information that is used by the implementation to provide a proper treatment of the values in the GUI (that is, the representation and input conversions). For example, such types may include the following:

- RichText (**which may specialize Text**) — The internal representation of the text may include formatting information. The type may be supported by the appropriate GUI widgets for rendering and editing the contents as well as its formatting.
- Currency (**which may specialize Real**) — This type may have no additional features, but may indicate to the framework to provide the appropriate representation of its instances (for example, display always with two decimal places, comma separators of thousands, and the local currency sign). On the other hand, this type may provide additional features aimed at more complex computation, such as the unit (currency) of the value, and operations such as adding and subtracting values expressed in different units according to the appropriate currency rate (possibly variable over time).
- Picture, Sound, and Movie (**which may specialize File**) — Again, these types may have no additional features, but may indicate to the framework to provide the appropriate representation of its instances (displaying it as a picture, or with the appropriate widget for playing the sound or movie on a user's request). On the other hand, a type may provide more features that can be used for presentation of the contents (for example, the format of the picture).

The set of features provided by the built-in data types is not prescribed, either. However, the expected features are as follows:

- Constructors that provide the necessary initialization, including default constructors wherever appropriate — For example, a `Date` should be initialized by the year, month, and day, assuming the appropriate validity checking.

- ❑ **Conversions from and to other expected data types, especially the native types of the detail-level language** — The conversions from other data types should be provided by the appropriate constructors accepting the corresponding parameters, and the conversions to other data types should be provided by appropriate operations that return the target types. All data types that can have textual representation should provide conversions from and to `Text`.
- ❑ **Operations of the appropriate algebra that provide the calculations within the extent of the same type** — For example, `Integer` should provide integer arithmetic operations, including integer division. These may include relational operations, too, such as comparison.
- ❑ **Operations on operands (parameters) or with results of other expected types** — For example, `Real` should provide arithmetic operations that accept integers as well as real numbers, or truncate and round operations that return integers. Similarly, there could be an operation that subtracts two `DateTime` values and returns a `TimeInterval`. These may include relational operations, too, such as comparison.
- ❑ **Other specific operations expected for the data type** — For example, `File` may provide an operation that saves the contents as a file at the given place in the file system, or the appropriate operations for accessing (reading and writing) the binary contents through a usual file programming interface.

As already stated, some built-in data types may have literals. Literals of those built-in data types represent references to instances of those types, created at the place of their occurrence, and having the specified values. Consider the following example:

```
Boolean b = false;  
Integer i = 2;  
Text txt = "Hello world!";
```

It is assumed that literals of Boolean data type are accessible without qualification.

On the other side from built-in data types are the data types defined by the modeler. These are referred to as *user-defined* types. These types are modeled according to the general rules, and may have specific features (attributes and operations). Apart from the general OOIS UML rules, an OOIS UML implementation may pose additional requirements for the construction of user-defined types. For example, an implementation may require a user-defined data type to specialize a built-in (possibly abstract) data type.

Section Summary

- ❑ An OOIS UML implementation should offer a library of predefined built-in data types ready to use for common modeling purposes. The set of these types is not prescribed, but an implementation should provide at least the following built-in data types:
 - ❑ `Boolean`
 - ❑ `Text`
 - ❑ `Integer`

Continued

- Real
 - TimeInterval
 - Time
 - Date
 - DateTime
 - File.
- An implementation may offer other specialized built-in data types (such as RichText, Currency, Picture, Sound, Movie, and so on).
- User-defined* data types are types defined by the modeler, according to the general OÖIS UML rules and optional additional rules imposed by the implementation.

9

Attributes

Attributes are structural features of classes and data types. This chapter discusses different modeling aspects of attributes. Actions on attributes perform elementary access or modifications of attribute values at run-time. This chapter also examines actions on attributes available in OOIS UML.

Attributes as Structural Features

An *attribute* is a structural feature of a classifier that describes a range of values that instances of the classifier can hold in *slots* for that feature. An attribute defines a piece of the static structure of the classifier's instances. It implies that all instances of that classifier have a corresponding slot for that feature. The set of attribute values held in slots defines the current state of a classifier's instance. This state (that is, the set of attribute values) is proprietary to the instance.

More formally, an attribute maps each classifier instance to a value or collection of values of the type of the attribute.

In UML, an attribute has the following characteristics, which are explained in detail in the sections that follow:

- ❑ It is a *property* of a classifier, and, thus, it may be *derived*, meaning that its value can be computed from other sources of information.
- ❑ As a property, it is also a *structural feature*, and, thus, it can be *read-only*, which means that its value cannot be modified after its initialization.
- ❑ As a structural feature, it is also a *multiplicity element*, and, thus, it has a multiplicity specification, meaning that its slot maps an instance of the classifier to a value or a collection of values of the type of the attribute.
- ❑ As a structural feature, it is also a *typed element*, meaning that it has its type, which constrains its values.

Part III: Concepts

- ❑ As a structural feature, it is also a *feature*, and, thus, it can be *static*, when its value is shared among all instances of the classifier.
- ❑ As a feature of a classifier, it is also a *redefinable element*, meaning that a definition of an attribute in a specializing class can redefine a definition of an attribute in a generalizing class.
- ❑ As a redefinable element, it is also a *named element*, and, thus, it has its name, which must be unique within the scope of its classifier.

Section Summary

- ❑ An *attribute* is a structural feature of a classifier that describes a range of values that instances of the classifier can hold.
- ❑ Attributes define a static structure of the classifier's instances (which is shared by these instances) and attribute values define the current states of the classifier's instances (which are proprietary to the instances).
- ❑ The run-time manifestation of a classifier's attribute is a *slot* existing in each instance of the classifier, holding the value of the attribute, or more formally, mapping its owner classifier instance to a value or collection of values of the type of the attribute.
- ❑ In UML, an attribute is a property and is a structural feature, a multiplicity typed element, and a named element, and can be static, derived, and redefined.

Attributes as Multiplicity Typed Elements

An attribute is a typed element, meaning that its type constrains its values. In standard UML, the type of an attribute can be any classifier (that is, a class or a data type), whereas in OOIS UML, the type of an attribute can be only a data type. The type of an attribute must be a data type that is accessible from the classifier that owns the attribute.

An attribute is also a multiplicity element, meaning that it has the specified multiplicity, which constrains the cardinality of the collection of values in its slot. Simply said, in the case of a multi-valued attribute, the slot can hold multiple values.

For example, let's assume that the central abstraction in a Web catalogue system of a trading company is a *product*, for which one or more pictures can be provided in the catalogue. The attribute *picture* of the class *Product* in that system can be specified to have the multiplicity $1..*$, in the UML textual notation:

```
picture : Picture[1..*]
```

At run-time, each object of *Product* will have a slot named *picture*, which holds a collection of values of type *Picture* with the cardinality one or more. Unless specified differently, the default multiplicity for attributes is 1 in OOIS UML.

More formally, an attribute of a classifier defines a *mapping* of each instance of the classifier to a value or a collection of values of the type of the attribute. For the preceding example, the attribute `Product::picture` relates every object of the class `Product` to a collection of instances of the data type `Picture`. For example, if `p` refers to an object of `Product`, then the slot `p.picture` denotes the result of the mapping `picture` for the object `p`. The multiplicity specification of the attribute constrains the cardinality of that collection. It is an implicit constraint attached to the attribute.

Interestingly, this mapping changes during run-time because the set of objects (the domain of the mapping) changes as objects are created and destroyed, and the values of the attributes (the co-domain of the mapping) changes as a result of actions performed on the slots of objects.

An attribute can (but need not) have a default value. In OOS UML, the default value is a specification of the expression that is used to initialize the slot when the owner instance is created. The expression's result must conform to the attribute. The expression is specified in the detail-level language. For example, if you want to provide a default initialization of the attribute `Product::picture` with an "empty" picture for the product, the attribute's specification can be as follows:

```
picture : Picture[1..*] = new Picture("Default Product Picture.jpg")
```

For multi-valued attributes, instead of specifying one expression that results in a multiple value, a comma-separated list of single-valued expressions can be provided within curly braces, as shown in the following example:

```
seasonDiscounts : Integer[4] = {0,10,0,20}
```

The lower multiplicity bound of the attribute can be greater than zero, while the specification of the default value can provide fewer values than required by that lower multiplicity bound. In such a case, the rest of the attribute's values up to when the lower multiplicity bound is satisfied will be initialized by the attribute type's default constructor. If the attribute is ordered, the explicitly provided default values will initialize the first values of the attribute in order, while the rest will be initialized by the default constructor. In the following example, two values of the attribute will be initialized to 10 and 20, while the other two will be initialized using the default constructor of `Integer`:

```
seasonDiscounts : Integer[4] = {10,20}
```

The values of different attributes of one classifier instance are initialized in an arbitrary (implementation-specific) order. The initialization of the attribute values takes place before the constructor of the owner instance is invoked, and has the same semantics as the action of setting the value of the attribute.

When the textual notation is used for attributes (for example, within the symbols of classes), the following syntax is used:

```
property-decl ::= visibilityopt name : type multiplicity-specopt =
    initializer modifiersopt
property-decl ::= visibilityopt name : type multiplicity-specopt modifiersopt
property-decl ::= visibilityopt name multiplicity-specopt modifiersopt

visibility is one of + # - ~
```

Part III: Concepts

```
multiplicity-spec ::= [ multiplicity ]  
  
initializer ::= expression  
initializer ::= { expression-list }  
expression-list ::= expression  
expression-list ::= expression , expression-list  
  
modifiers ::= { modifier-list }  
modifier-list ::= modifier  
modifier-list ::= modifier , modifier-list
```

Here, *modifier* is either ordered, unordered, unique, nonunique, readOnly, or unrestricted.

Section Summary

- An *attribute* is a typed multiplicity element, meaning that it relates an instance of the classifier to a collection of values of its type, with the cardinality of the collection constrained by the attribute's multiplicity. The type of an attribute is a data type.
- An attribute can have a default value, specified as an expression or a list of expressions that initialize the slot when the owner instance is being created.

Static Attributes

Suppose that you want to trace the exact number of instances of the class `Person` in an information system. One approach would be to pose a query that counts all existing objects of `Person` whenever that information is needed. However, that approach can be unacceptably inefficient if the information is needed very often because the same query may take a long time to execute, and it can be re-executed many times, even though the number of objects has not changed in the meantime.

Another approach could be to store the number of objects of the class in a slot that behaves like a counter variable, and to increment its value every time an object of `Person` is created, and decrement it every time an object is destroyed. The proper moments of creation and destruction of objects can be easily captured by the constructors and destructors. Every constructor of the class `Person` should increment the counter of objects, and every destructor of this class should decrement the counter. This way, there is no possibility that any object is missed, and the counter is constantly kept up-to-date.

However, the question is, where do you store this counter? It is obvious that the counter logically belongs to the class `Person`, but it does not belong to any individual object. Therefore, it cannot be modeled with an ordinary attribute. Instead, the information is shared among all objects of the class. That is, it belongs to the class itself, not to its objects.

For this purpose, UML (as with most other traditional OO programming languages) supports *static attributes*. A static attribute is an attribute of a classifier whose value is shared among all instances of that classifier. For the given example, Figure 9-1 shows the solution using a static attribute `Person::count`. In

UML, static features of classifiers are underlined in diagrams. (It is possible to underline just the name of the feature, or its entire declaration.) The attribute count has the initial value 0, and the constructor(s) and destructor(s) provide the proper updates of the counter.

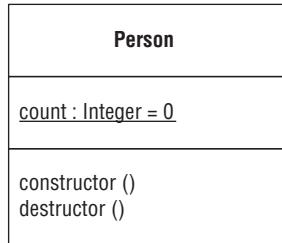


Figure 9-1: Static attribute

Person::count

In general, while a non-static attribute defines a mapping of each instance of a classifier to the values of the attribute's type, a static attribute defines a mapping that is independent of the classifier's instance. In other words, if `attr` is a static attribute and `p` is an instance of a classifier, the result of the mapping of `attr(p)` (or, in the UML notation, `p.attr`) at some point in run-time has the same result for every `p`. Therefore, the value of a static attribute can be referred to in two different ways:

- ❑ Specifying the classifier's instance and the attribute, `p.attr`, where the reference to the instance is actually irrelevant and does not need to be computed at run-time because the result is the same for each instance. Instead, it is used to determine, at compile time, the classifier to which the static attribute belongs.
- ❑ Without specifying the classifier's instance, but just referring to the attribute as a named feature of the classifier. For example, the attribute `count` can be referred to by its qualified name (for example, `Person::count`) or even by its unqualified name (`count`) within the namespace of its classifier.

The value of a static attribute is inherited in specializing classifiers.¹ For the mentioned example, if the class `Teacher` specializes the class `Person`, the attribute `Person::count` can be accessed over the objects of the class `Teacher`:

```
Teacher t = ...;
...t.count... // Refers to Person::count
```

In OOIS UML, static attributes are initialized in an undefined order, at the system's start-up time. The precise timing of the initialization (that is, the exact meaning of the "start-up time") is implementation-dependent. It may simply mean setting up the entire system and its object space. They are initialized even before any other instance of the classifier is created, or any application execution is activated. They also survive every execution of the application.

¹This is a semantic specialization of the OOIS UML profile. In standard UML, it is a semantic variation point, whether the values of static attributes are inherited or not.

Part III: Concepts

In general, static attributes are used in models when some information must be shared among all instances of the same classifier, or it belongs to the classifier, not to each particular instance. Some examples include the situations when the objects of a class should be counted, or some cumulative information should be stored for a class (for example, the sum or average value of a non-static attribute of all its objects, and so on). In effect, static attributes replace the notion of global static variables in traditional programming languages. Static attributes have the same purpose and very similar semantics as global static variables in traditional programming languages, but they have certain properties that make them different:

- ❑ Static attributes are logically “packed” in their classifiers, and, thus, clearly indicate that some information is important for and logically related to a certain classifier, and does not exist independently, as global variables do.
- ❑ Static attributes are named elements that belong to their classifiers as namespaces, and, thus, follow all naming rules (referred to by qualified or unqualified names).
- ❑ Static attributes (as opposed to global variables) can be encapsulated within classifiers because they (as with all other features of classifiers) have visibility, which may be public, protected, or private.

Section Summary

- ❑ A *static attribute* is an attribute of a classifier whose value is shared among all instances of that classifier. A static attribute defines a mapping of a classifier to the values of the attribute’s type that is independent of the classifier’s instances.
- ❑ Static attributes can be accessed over references to the classifier’s instances, or without references to instances.
- ❑ The values of static attributes are inherited in specializing classifiers.
- ❑ Static attributes are initialized in an implementation-dependent order at the start-up of the system.
- ❑ Static attributes are underlined in the textual UML notation.
- ❑ Static attributes are used when some information must be shared among all instances of the same classifier, or it belongs to the classifier, not to any individual instance.

Read-Only Attributes

Sometimes you may need an attribute to retain the same value over its entire lifetime, that is, to be immutable after the initialization of the owner instance. For example, the Social Security Number, as an attribute of the class `Person`, should be assigned a value at the moment of constructing an object of the class, and protected from later modifications (that is, from writing new values), during the lifetime of the object.

For such a purpose, UML introduces the notion of *read-only* attributes. An attribute of a classifier, static or non-static, can be specified to be read-only, which means that its value cannot be modified after its

initialization (for attributes of data types and static attributes of classes) or after the construction of its owner object (for non-static attributes of classes)².

In OOIS UML, if a non-static attribute of a class is read-only, then its value can be modified by an action only during the initialization of its owner object. Otherwise, if an action that modifies its value is activated after the owner object has completed its construction, an exception of type `WritingReadOnlyAttributeException` will be raised.

For example, consider the class `Person` with a read-only, non-static attribute `ssn` that stores the Social Security Number of a person:

```
+ ssn : Text {readOnly}
```

It can be assigned a value during the execution of the constructor for an object of this class, but it cannot be modified after the construction of the object is finished. In the OOIS UML native detail-level language, the method for the constructor `Person::constructor(s:Text[1])` may look like this:

```
... // Check the validity of the format of s and then do the assignment:  
ssn = s; // Correct when it is executed during construction
```

However, any such modification executed after the object has been constructed will raise an exception.

Checking whether an action can or cannot modify the value of a read-only non-static attribute of a class is performed exclusively at run-time, not statically. For example, such an action can be specified within the method of an operation that is not a constructor of a class. If that operation is invoked during the construction of the owner object, it can modify the read-only attribute. On the other hand, if the same operation is invoked after the owner object has been constructed (that is, after the invocation of the object's constructor from a Create Object action has been completed), it will fail.

For example, let's assume that the operation `Person::writeSSN(s:Text[1])` performs an action that stores the value of its argument to the read-only attribute `Person::ssn` like this:

```
... // Check the validity of the format of s and then do the assignment:  
ssn = s; // Not prevented at compile time
```

The method for the constructor `Person::constructor(s:Text[1])` looks like this:

```
writeSSN(s); // Correct when it is executed during construction
```

The invocation of `writeSSN` during the construction of an object of `Person` (that is, during the invocation of the constructor from the Create Object action) is correct. However, a stand-alone invocation of the same operation `writeSSN` or even the constructor (for an already existing object of `Person`) will result in an exception.

The same holds true for accessing a read-only non-static attribute of a class over the reflection mechanism.

²This is a semantic specialization of the OOIS UML profile. In the standard UML, the semantics of read-only structural features is not defined so strictly.

Part III: Concepts

On the other hand, in general, a static attribute of a classifier lives independently of any instance of that classifier. It is initialized during the start-up time of the system, even before any other instance of the classifier is created, or any application execution is activated. Its initialization is not bound to the initialization of any instance of the classifier, and has no direct correlation with the constructors of the classifier and their executions.

This is why static read-only attributes cannot be modified after their initialization. Every action that attempts to write such an attribute is executed within a method and is, thus, illegal because it is always executed after the start-up time of the system. This rule can be checked also statically. If such attributes are accessed dynamically through the reflection mechanism, the checking is always performed at run-time, and may raise the same exception.

As for non-static attributes of data types, it should be noted that a Write Attribute Action does not actually modify an attribute, but always produces a new data type instance with the new value of that attribute, as will be explained later in this chapter. For that reason, annotating an attribute of a data type as read-only may simply mean preventing any attempt from issuing such an action. This is why read-only attributes of data types behave as static attributes — they cannot be modified (that is, attempted to be written) after their initialization. This rule can be checked, even statically. If such attributes are accessed dynamically through the reflection mechanism, the checking is always performed at run-time, and may raise the same exception.

A read-only attribute is shown using the `readOnly` modifier as part of the notation for the attribute. A modifiable (non-read-only) attribute is shown using the modifier `unrestricted` as part of the notation for the attribute. This annotation may be suppressed, in which case it is not possible to determine from the diagram whether the attribute is read-only or modifiable. However, a modeling tool may allow suppression of this annotation only when the attribute is not read-only. In this case, it is possible to assume that the attribute is modifiable in all cases where the modifier `readOnly` is not shown.

Section Summary

- ❑ An attribute of a classifier can be specified as *read-only*, which means the following:
 - ❑ For non-static attributes of classes, the read-only attribute cannot be modified after the creation of its owner object has been completed. Any action that modifies its value and is executed after the initialization of its owner object has been completed will raise an exception at run-time.
 - ❑ For static attributes of classes and all attributes of data types, the read-only attribute cannot be modified after its initialization.
- ❑ Read-only attributes are shown using the `readOnly` modifier.

Frozen Attributes³

Sometimes it is too early to “freeze” the value of an attribute as soon as the constructor of the owner object is completed (as read-only non-static attributes of classes work). Instead, it is necessary to freeze it at a later moment, so that the value should not be modified any more from that moment on. For example, the system may allow the user to set the value of an attribute of an already created object only once, and to preserve that value afterward, preventing its modifications by all means. Or, the system may provide only certain means of modifying the value of the attribute, while all other ways of modifications (including generic and application-specific ones) should be prohibited.

For that purpose, an attribute value of an object can be *frozen* by an explicit action:

```
anObject.anAttribute->freeze()
```

After that action, any other action that attempts to modify the value of the attribute of that object would raise an exception of type `WritingReadOnlyAttributeException`. This checking is always performed dynamically.

An attribute value can be unfrozen as follows:

```
anObject.anAttribute->unfreeze()
```

After that, the attribute value can be modified again.

Read-only non-static attributes of classes are actually those for which an implicit freeze action is performed at the corresponding moment (after the initialization of the owner object). The unfreeze action on read-only attributes is an error reported at either compile time or run-time.

Section Summary

- An attribute value of an object can be frozen by an explicit action.
- A frozen attribute value cannot be modified. Any action that tries to modify a frozen attribute value will raise an exception.
- An attribute value can be unfrozen.

Derived Attributes

Let’s revisit the example of the Easylearn school system from Chapter 5, whose partial model is shown again in Figure 9-2. In that system, it was necessary to implement the calculation of the revenue for each

³This language feature is specific for the OOIS UML profile and does not exist in standard UML.

Part III: Concepts

course. The revenue was derived from the course price, the number of its attendees, and the discounting policy of the school. The calculation of the revenue was performed in the method of the operation `Course::calcRevenue`. However, that method recomputed the revenue and returned it as its result at every invocation. That approach could be inefficient if the revenue of a course is retrieved many times, even though the computation is not trivial and may require a traversal of a significant part of the object space. On the other hand, there is no need for such recalculation until the object structure on which the result depends has changed, which may happen only rarely (for example, several times a year, when the course is restarted).

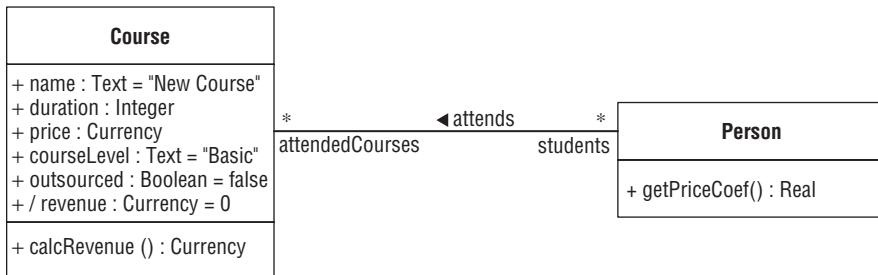


Figure 9-2: For the derived attribute `Course::revenue`, the value of this attribute can be computed from other information — the course price and the number of its attendees, in this case. The computation is performed by the operation `calcRevenue`, which updates the value of the attribute `revenue`.

An alternative is to store the computed value in a separate attribute, named `revenue`, and then to have that value ready for retrieval without recalculation. The method for the operation `Course::calcRevenue` now can be slightly modified so that it stores the result of its calculation in the attribute `revenue`. When users need to see the course revenue, it can be simply read from this attribute, without the need for complex recalculation. Of course, the application must now provide the proper updates of this attribute whenever the information upon which its value is based has changed, or again on an explicit request of the user. A portion of the modified model for this system is shown in Figure 9-2.

Figure 9-1 showed a similar situation with the static attribute `Person::count`, which was used to trace the number of existing instances of the class `Person`. Its value was updated by constructors and destructors, but could also have been computed by counting objects of the class when needed. Basically, the value of that attribute is also derived from other information.

To deal with situations like these, UML recognizes *derived attributes*. A derived attribute of a classifier is an attribute whose value can be computed from other sources of information. In the described examples, the attributes `Course::revenue` and `Person::count` are derived because their values are (or can be) computed from other sources of information. In UML, derived attributes are displayed with a slash in front of the name, as shown in Figure 9-2.

Derived attributes have no special implications on the run-time semantics of the model. When an attribute is annotated as derived, it has only an informative meaning to the reader or implementer of the model, and has no direct effect on the structure or behavior of the system. It simply indicates to the reader that the value of the attribute is in some way computed from other sources of information. It does not indicate *how* and *when* it is computed. Instead, it is up to the modeler to specify how this

value is related to other sources of information, and to which sources it is related, in terms of other modeling concepts. Therefore, the semantics of attributes also holds true for derived attributes — most notably, all actions that read or write the value of a derived attribute work for derived attributes as usual. Consequently, the notion of a derived attribute adds no run-time semantics to ordinary attributes in UML.

In practice, there are several approaches to implement the computation and maintenance of derived attributes. These approaches actually deal with answering the questions *when* and *how* the derived attribute is recomputed:

- ❑ **The modeling framework and the run-time environment do not provide any special support for maintenance of values of derived attributes.** In this case, modelers are responsible for providing the elements of the system (most probably methods) for keeping the value of a derived attribute up-to-date. Basically, this may encompass the following activities:
 - ❑ Identifying all elements of the model that represent the sources of information that may affect the value of the derived attribute.
 - ❑ Capturing all circumstances in which the values of the identified sources of information can be modified, and intercepting the actions that modify them. Most often, this results in the need to encapsulate the identified sources of information (for example, by declaring them as private or protected features), and wrapping the access to them into operations (for programmatic access) or commands (for interactive access). A simple alternative can be to let the user explicitly require recalculation of the derived attribute through a command.
 - ❑ Providing the method(s) that updates the value of the derived attribute when any source of information is changed.
 - ❑ Preventing unauthorized access to the derived attribute (that is, preventing the unrestricted writing of its value), so that the attribute can be changed only in a controlled way by the described methods. This may again result in declaring that attribute as private or protected, and wrapping the access to it into operations and commands, or freezing the attribute value outside the restricted modification methods.

In the early stages of development, especially during analysis, the relationship between the value of the derived attribute and the elements that provide the source information can be specified by constraints. However, these constraints cannot provide the automatic update of the derived attribute value, and most often should not have any effect at run-time. Therefore, such constraints should be replaced by the proper maintenance mechanisms during design, before the model is transformed into an implementation form.

As you can see, this approach may require much development effort and care. A simple alternative, applicable in many cases, is to let the user explicitly request the recalculation of the value of a derived attribute through a command (for example, by clicking a button that recalculates the value).

- ❑ **The modeling framework and the run-time environment provide some mechanism for implicit and automatic update of derived attribute values.** In effect, this mechanism represents an automatic equivalent for the previous approach:
 - ❑ The modeler can identify the sources of information (that is, the elements of the model) that affect the computation of the derived attribute, and then *explicitly* specify their relationship

Part III: Concepts

with the derived attribute by certain modeling concepts. Furthermore, the modeler identifies the *events* that serve as triggers for recalculating the derived attribute. The events can be expressed in terms of actions that modify the source information elements.

- ❑ The modeler specifies the method or expression for computation of the derived attribute value from the source information elements. The method or expression should be defined in a formal way, using one of the available techniques for specifying methods and expressions.
- ❑ The run-time environment tracks all events that change the source information elements and triggers the updating method on every such event to update the derived attribute value automatically.

As you can see, this approach is basically analogous to the mechanism of triggers and stored procedures in relational DBMSs, and, thus, can be implemented in a similar way. However, this approach needs a more precise definition of the concept of triggers, the relationship between the source information elements and the derived attribute, and the details of specifying the method or expression (most notably, the means for referencing the source information elements). For these reasons, this approach is still beyond the scope of standard UML and OOIS UML, but can be optionally supported by an implementation of the profile.

- ❑ **The value of the derived attribute can be recalculated whenever it is retrieved (for example, on each read action).** Therefore, the value of the derived attribute does not need to reside in any “permanent” computer memory, but can be simply recalculated as a temporary value. The way of computing its value (in terms of the source information elements) can again be specified in the model by a method or expression. The run-time environment then provides the necessary support to recalculate the value of the derived attribute at certain points in time, not explicitly triggered by modifications of the source information, but on every access to the derived attribute instead.

As you can see, this approach is basically analogous to the mechanism of views (queries) in relational DBMSs that can offer calculated fields in their record sets, and, thus, can be implemented in a similar way. However, this approach needs a more precise definition of the details of specifying the methods and expressions (most notably, the means for referencing the source information elements), as well as the implicit triggers that activate the recalculation. For these reasons, this approach is still beyond the scope of standard UML and OOIS UML, but can be optionally supported by a certain implementation of the profile.

In short, only the first approach is supported by the current version of OOIS UML, whereas standard UML allows all approaches, but does not specify the precise semantics or details of any. The other two approaches can be optionally supported by a modeling framework and run-time environment that implement the profile, but in that case, the implementation must resolve all open issues identified previously. Moreover, an implementation can even offer all three approaches, and the modeler can select the one that is most appropriate to a particular case.

It is obvious that all these approaches often represent a trade-off because their computational cost can be different. If it supports the second or the third approach, an implementation may also provide better controllability over the access to the derived attribute by permitting writing of its value only from the automatic updating methods implicitly invoked by triggers from the run-time environment.

In general, derived attributes are used when some information that can be derived from other information should be retrieved and is retrieved relatively often, but its computation is relatively costly. Of course, there is always a trade-off, because the maintenance of the derived attribute requires some additional modeling efforts and run-time overhead, as just described. Therefore, modelers must decide whether to introduce a derived attribute at all, and if they do that, to select the proper implementation approach from those given here.

If the source information is only rarely changed, and the derived information is often retrieved, then the derived attribute should be introduced and implemented using the first or the second approach, if available. On the other hand, if the situation is opposite, either the derived attribute should be implemented using the third approach, or the derived information should be provided by another means (for example, by an operation, as `Course::calcRevenue` did in Chapter 5).

Section Summary

- ❑ A *derived attribute* of a classifier is an attribute whose value can be computed from other sources of information.
- ❑ A derived attribute is an informative annotation that tells the reader of the model that the value of the attribute can be computed from other sources of information, but not how and when it is computed. Read and write actions work with derived attributes as usual.
- ❑ To implement the maintenance of a derived attribute, the developer must select one of the following approaches that deal with *when* and *how* the derived attribute is recomputed:
 - ❑ Identify other model elements that provide the source for the computation of the derived attribute and explicitly deal with capturing all circumstances when the source information changes. This is the only approach currently defined in OOIS UML.
 - ❑ Rely on an implementation-dependent mechanism provided by the framework that encompasses triggers and methods/expressions for implicit (automatic) updates of the derived attribute on any change of the source information elements.
 - ❑ Rely on an implementation-dependent mechanism provided by the environment that re-computes the derived attribute value whenever it is accessed.
- ❑ Introduction and implementation of a derived attribute is most often a trade-off. The decision is influenced by the frequency and cost of modifications of the source information, as well as the frequency and cost of retrieval of the derived information.

Redefinition of Attributes

In standard UML, features of classifiers (including attributes) can be *redefined* in specializing classifiers. A (directly or indirectly) specializing classifier can have a feature — for example, an attribute that redefines a feature (attribute) from a generalizing classifier. The redefining feature of the specializing classifier represents a specification that augments, constrains, or overrides the specification of the redefined feature. Consequently, for the instances of the specializing classifier, the specification provided by the redefining feature holds, along with or instead of (parts of) the specification of the redefined feature.

Redefinition in standard UML is a general mechanism that applies to many language elements — most notably, to features of classifiers. However, standard UML does not specify strict rules and precise meaning of redefinition in most cases. It just requires that the redefining element be compatible with the redefined element, while leaving the meaning of “being compatible” as a semantic variation point.

As for properties of classifiers (including attributes and association ends) a property of a specializing class may redefine another property of a generalizing class, with the effect of augmenting, constraining, or overriding the definition of the redefined property. This does not mean that the instances of the specializing class will have a separate slot as a run-time manifestation of the redefining property. They will have the same slot that is the run-time manifestation of the redefined property, but it will possibly have different characteristics. In other words, for the instances of the specializing class, the rules implied from the redefining property will hold for the slot that is the manifestation of the redefined property.

The redefining property is compatible with the redefined property if and only if the redefining property (as a typed multiplicity element) conforms to the redefined property, and if the redefining property is derived when the redefined property is derived. A derived property can redefine one that is not derived, however. The modeler must ensure that the constraints implied by the derivation are preserved when the property is updated. If a property has a specified default value and redefines another property with a specified default value, then the redefining property’s default is used in place of the redefined property’s default for the instances of the classifier that owns the redefining property. In standard UML, the name and visibility of the redefining property are not required to match those of the property it redefines.

Figure 9-3 shows an example of attribute redefinitions. In the specializing class `SquareSymbol`:

- ❑ The attribute `SquareSymbol::shape` redefines (the type of) the attribute `Symbol::shape`, meaning that the same slot of the attribute `shape` in objects of the class `SquareSymbol` may hold just the values of the type `Square`, which is a specialization of `Rectangle`.
- ❑ The attribute `SquareSymbol::height` redefines (the default value of) the attribute `Symbol::height`, meaning that the same slot `height` in objects of the class `SquareSymbol` will have the initial value 7 instead of the initial value 5 specified in the class `Symbol`.
- ❑ The derived attribute `SquareSymbol::width` redefines the attribute `Symbol::width`, which is not derived, meaning that the value of the attribute `width` in objects of the class `SquareSymbol` will be derived from other information, obviously from the attribute `height`.

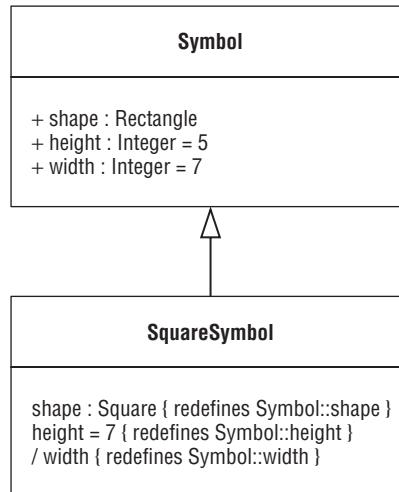


Figure 9-3: In this example of the **redefinition of attributes**, the attribute `SquareSymbol::shape` **redefines the type of the attribute** `Symbol::shape`. The attribute `SquareSymbol::height` **redefines the default value of the attribute** `Symbol::height`. The derived attribute `SquareSymbol::width` **redefines the attribute** `Symbol::width`, which is not derived.

In OOIS UML, only non-static attributes of classes can be redefined. Moreover, redefinition of attributes of classes is constrained to some extent compared with the standard UML, and the semantics of redefinition of particular aspects are more precisely defined as follows:

- ❑ The visibility and name of the attribute cannot be redefined. More precisely, the visibility and name of the redefining attribute must exactly match the visibility and name of the redefined attribute.⁴
- ❑ The redefining attribute may redefine the type of the redefined attribute, by specifying a type that conforms to the type of the redefined attribute. The semantics of such a redefinition are derived. It implies a constraint on the type of the attribute's value, implicitly defined in the context of the specializing class that owns the redefining attribute. For the given example in Figure 9-3 and the redefinition of the attribute `shape`, there is an implicit constraint in the specializing class `SquareSymbol` that constrains the type of the value of the inherited attribute `shape`.

⁴In standard UML, if a feature in the specializing classifier has the same name as a feature of the same kind from the generalizing classifier, then the former implicitly redefines the latter, even though they are not explicitly related so in the model.

Part III: Concepts

- ❑ The redefining attribute may redefine the multiplicity of the redefined attribute, by specifying a multiplicity that is included in the multiplicity of the redefined attribute. The semantics of such a redefinition are derived. It implies a constraint on the multiplicity of the attribute's value, implicitly defined in the context of the specializing class that owns the redefining attribute. As a special case, the redefining multiplicity can be 0..0 (provided that it is included in the redefined multiplicity), meaning that the slots in the objects of the specializing class cannot have values at all.
- ❑ The redefining attribute may redefine the default value of the redefined attribute. The objects of the specializing class will have the redefined initial value of their attribute instead of the default value from the generalizing class.
- ❑ The redefining attribute may be derived, although the redefined attribute is not derived. This means that for the objects of the specializing class, the value of the redefined attribute will be derived in some way from other sources of information. If the redefined attribute is derived, the redefining attribute must also be derived.

In OOIS UML, the redefinition of attributes of data types and static attributes of classes is not allowed.

An attribute of the specializing class implicitly redefines an attribute with the same name in the generalizing class. In the UML notation, the redefinitions can be made explicit with the use of a `{redefines ...}` string beside the textual representation of the attribute, as shown in Figure 9-3.

Section Summary

- ❑ In standard UML, a property of a specializing classifier can *redefine* a property of a generalizing classifier, providing a compatible specification that augments, constrains, or overrides the specification of the redefined property.
- ❑ The existence of the redefining property in the specializing class does not suggest the existence of a separate slot as its run-time manifestation in the instances of that class, but just augments, constrains, or overrides the characteristics of the slot implied from the redefined property and the values it can hold.
- ❑ In OOIS UML, only some aspects of non-static attributes of classes can be redefined:
 - ❑ The visibility and name of the redefining attribute must match those of the redefined attribute.
 - ❑ The redefining attribute may redefine the type of the redefined attribute by a type that conforms to the type of the redefined attribute. Such a redefinition implies a constraint on the type of the attribute's value, implicitly defined in the context of the specializing class that owns the redefining attribute.
 - ❑ The redefining attribute may redefine the multiplicity of the redefined attribute by specifying a multiplicity that is included in the

multiplicity of the redefined attribute. Such a redefinition implies a constraint on the multiplicity of the attribute's value, implicitly defined in the context of the specializing class that owns the redefining attribute.

- The redefining attribute may redefine the default value of the redefined attribute. The objects of the specializing class will have the redefined initial value of their attribute instead of the default value from the generalizing class.
- The redefining attribute may be derived, although the redefined attribute is not derived. This means that for the objects of the specializing class, the value of the redefined attribute will be derived in some way from other sources of information. If the redefined attribute is derived, the redefining attribute must also be derived.
- OOS UML does not support redefinition of attributes of data types and static attributes of classes.
- The notation for a redefining attribute is { `redefines redefined_element` }.

Actions on Attributes

Actions on attributes provide the means to read and write attribute values programmatically. Basically, the set of actions on attributes is analogous to the set of actions on variables. Therefore, methods can access the values of attributes in completely the same manner as the values of variables.

However, the execution of actions on attributes of classes may encompass much more processing behind the scenes, and has more complex semantics than the execution of actions on variables or on attributes of data types. The basic reason for this is that the values of variables have a local scope within the execution of an enclosing behavior (for example, a method), and are, thus, not accessed concurrently from different executions and do not survive the execution of the enclosing behavior. Similarly, each instance of data type, being a pure value that cannot be modified, is in some sense also local to the execution because there are no other references from other executions to the same physical instance of data type that may affect its value. On the other hand, objects reside in a common object space that is accessed concurrently from different executions. In general, the attribute values of objects persist across executions of the behavior enclosing the actions and can be accessed concurrently by these actions. In order to preserve the semantics of actions in the presence of such concurrent access, the execution of actions must incorporate some concurrency control mechanism, such as locking.

The background locking mechanism incorporated in actions on attributes of objects may use a relaxed policy that differentiates the access to an attribute that only reads its value and the one that modifies its value. Such a relaxed policy may improve concurrency by enabling several reading actions to access the attribute concurrently without collision. For that purpose, the actions on attributes are clearly divided into two categories — Read Attribute actions and Write Attribute actions.

The remainder of this chapter contains advanced reference material that can be skipped on first reading.

Part III: Concepts

In the following sections, each of the actions is presented in the following list:

- ❑ **Description** — A brief, informal description of the purpose and semantics of the action.
- ❑ **Declaration** — A formal declaration of the action, its parameters and result types, using the UML notation for operations. This declaration serves as a quick reminder of the action's syntax and meaning. The `throws` specification in some declarations lists the types of exceptions that can be raised in the action. Exceptions are explained later in this book.
- ❑ **Static rules** — The set of constraints for the action. These rules are checked at compile time, and have no impact on the run-time semantics of the action. The rules usually define the constraints on the multiplicity, ordering, and uniqueness properties of the attribute on which the action can be applied. If one of these rules is violated, the action cannot compile (that is, the model is ill-formed), and it cannot be executed. The modeling environment should report an error in such cases.

One general static rule for all actions that have parameters is that the actual argument of the action must conform to the formal parameter of the action (both being typed multiplicity elements).

- ❑ **Run-time semantics** — The description of the effects of the action at run-time, when the action is executed (that is, instantiated), along with possible exceptions.
- ❑ **Examples** — Some simple illustrative examples of usage of the action.

In these explanations, it is assumed that the attribute of an instance is referred to by an expression denoted with `attr`, and that the attribute is of type `T`, multiplicity `m`, and with the uniqueness and ordering properties shortly denoted with `p`.

Note that an action upon an attribute (more precisely, on a slot) is invoked using the arrow notation (`attr->action(...)`), instead of the dot notation (`attr.operation(...)`). This is because the dot operator is used when its left operand represents a single value, which is a reference to an instance of a classifier. The actions described here work on the slot itself, which can map to a collection of values at run-time. Each value is actually a reference to an instance of a data type. If the multiplicity of the attribute allows zero cardinality, its slot may contain no values. If the attribute is single-valued, its slot may contain at most one value. If it is multi-valued, its slot contains a collection of values.

Read Attribute Actions⁵

This section presents the set of Read Attribute actions available in OOIS UML. These actions do not affect the attribute value.

1. Read Size of Attribute

Description: Returns the number of values currently present in a slot.

Declaration:

```
attr->size() : Integer[1]
```

⁵This is advanced reference material that can be skipped on first reading.

Static rules: This action is applicable to every attribute.

Run-time semantics: The action returns (a reference to) an integer that represents the current cardinality of the slot (that is, the actual number of values in it). It is always a non-negative integer.

Example:

```
Integer size = aProduct.picture->size();
```

2. Read Attribute

Description: Reads the entire slot and results in a single value or a collection of values with the same type, multiplicity, ordering, and uniqueness as the attribute.

Declaration: There are two equivalent notations for the same action:

```
attr : T[m]{p}  
attr->read() : T[m]{p}
```

Static rules: This action is applicable to every attribute. Its result has the same type, multiplicity, ordering, and uniqueness as the attribute. If the attribute is single-valued, the action results in a single value, and the dot operator may be applied to the result to access a feature of the instance referenced by the value. Following is an example:

```
Course c = ...;  
Text txtRevenue = c.revenue.toText();
```

Run-time semantics: Reads the values of the slot. The action results in a value or a collection of values with the same size, contents, and ordering as the attribute.

Examples:

```
Course c = ...;  
Real r = c.revenue; // c.revenue denotes a Read Attribute action,  
// just as c.revenue->read()
```

3. Read Value from Attribute

Description: Reads a single value from a (possibly multi-valued) slot.

Declaration: For attributes whose multiplicity allows zero cardinality:

```
attr->val() : T[0..1]
```

For attributes whose multiplicity does not allow zero cardinality:

```
attr->val() : T[1]
```

Static rules: This action is applicable to every attribute. If the lower multiplicity bound of the attribute is greater than zero, the multiplicity of the result is 1..1. If the lower multiplicity bound of the attribute is zero, the multiplicity of the result is 0..1.

Part III: Concepts

Run-time semantics: If the cardinality of the slot is zero (the slot has no value), the result's cardinality is zero (it has no value). Otherwise:

- ❑ For a single-valued attribute, the action returns the single value of the slot. It is equivalent to the Read Attribute action.
- ❑ For a multi-valued attribute, the action returns one value of the slot. If the attribute is ordered, it returns the first value in the collection. If the attribute is unordered, it returns one value from the collection, but it is undefined which one.

The action does not affect the slot.

Example:

```
Picture pic = aProduct.picture->val();
```

4. Count Occurrences of Value in Attribute

Description: Returns the number of occurrences of the given value in a slot.

Declaration:

```
attr->count(val:T[0..1]) : Integer[1]
```

Static rules: This action is applicable to every attribute.

Run-time semantics: If the argument val has no value (that is, its cardinality is zero), the action returns zero. Otherwise, the action counts the occurrences of the given value val in the slot and returns that number as a non-negative integer. Consequently, the result is always less than or equal to 1 for single-valued or unique multi-valued attributes. The result is zero if the given value does not exist in the slot. The criterion for equality comparison of the values is value-based (that is, it relies on the isEqual operation of data types). The action does not affect the slot.

Examples:

```
Picture aPicture = ...;
Integer c = aProduct.picture->count(aPicture);
```

5. Read Value from Ordered Attribute

Description: Reads a single value from the given position of a slot of an ordered attribute. Variations of this action return the first and the last value from a slot of an ordered attribute.

Declaration: For attributes whose multiplicity allows zero cardinality:

```
attr->at(pos:Integer[0..1]) : T[0..1] throws MultiplicityViolationException
attr->first() : T[0..1]
attr->last() : T[0..1]
```

For attributes whose multiplicity does not allow zero cardinality:

```
attr->at(pos:Integer[0..1]) : T[0..1] throws MultiplicityViolationException
attr->first() : T[1]
attr->last() : T[1]
```

Static rules: The attribute must be ordered.

Run-time semantics: The action does not affect the slot.

- `at` — Returns the value at the position given with the argument `pos`. The positions are zero-based (the first value in the collection has the position 0).

If the position is not given (the cardinality of the argument `pos` is zero), the effect is the same as for the `first` variation of this action.

If the given position `pos` has a value that is not included in the multiplicity of the attribute, the exception `MultiplicityViolationException` is raised (exceptions will be explained later in this book). Precisely, if the given position is less than the lower bound of the multiplicity allows (in effect, it is negative), the exception is of type `LowerBoundViolationException`, which is a subtype of `MultiplicityViolationException`. Similarly, if the given position is greater than the upper bound of the multiplicity allows, the exception is of type `UpperBoundViolationException`, which is a subtype of `MultiplicityViolationException`.

If the given position `pos` has a value that is out of the range defined by the current cardinality of the slot (but is still included in the attribute's multiplicity), no value is returned (the cardinality of the result is zero).

- `first, last` — Return the first and the last values in the slot of the ordered attribute, respectively. If the slot is empty (its current cardinality is zero — that is, it has no values), the result's cardinality is zero (it has no value).

Examples:

```
Picture aPic = aProduct.picture->first();
Picture aPic = aProduct.picture->last();
Picture aPic = aProduct.picture->at(3);
```

Section Summary

- Read Size of Attribute:

```
attr->size() : Integer[1]
```

- Read Attribute:

```
attr : T[m]{p}
attr->read() : T[m]{p}
```

- Read Value from Attribute:

```
attr->val() : T[min(m.lower,1)..1]
```

- Count Occurrences of Value in Attribute:

```
attr->count(val:T[0..1]) : Integer[1]
```

Continued

- Read Value from Ordered Attribute:

```
attr->at(pos:Integer[0..1]) : T[0..1]
    throws MultiplicityViolationException
attr->first() : T[min(m.lower,1)..1]
attr->last() : T[min(m.lower,1)..1]
```

Write Attribute Actions⁶

This section examines the set of Write Attribute actions available in OOIS UML. If any of these actions is invoked on a read-only attribute after the owner object has completed its construction, or for a currently frozen slot, an exception of type `WritingReadOnlyAttributeException` is raised. If the attribute is derived, it is the responsibility of the modeler to take care of the proper update of the source information, if appropriate.

One subtle detail should be noted here. Namely, according to the UML philosophy for data types representing pure values that can never be modified, a Write Attribute action never changes the value of the slot of a data type instance it works upon. Instead, it creates and returns a *new* data type instance having the new value of the affected attribute and all the other attribute values equal.

For the sake of symmetry, every Write Attribute action on a non-static attribute returns a reference to a classifier instance. If the slot for which the action was invoked belongs to an object of class, a reference to that same object is returned by the action. If the slot belongs to a data value, a reference to the newly created data value is returned by the action.

For example, if we assume that `Date` is a data type having an accessible attribute named `day` of type `Integer`, and that the variable `today` refers to an instance of that type, the following action returns a new instance of `Date` having the value of `day` set to 3:

```
today.day->set(3)
```

This simply means that the variable `today` will still refer to the old data value with the old value of `day`. In order to do what is probably intended (that is, to set the `day` of `today` to 3), you must write the following:

```
today = today.day->set(3);
```

Although this may look cumbersome, it really follows the basic semantics of data types in UML.

For static attributes, these actions do not return any result. These actions work the same for static attributes of classes and of data types. They always affect the single slot of such an attribute.

In the specifications that follow, the type of the classifier owning the attribute is denoted with `c`.

⁶This is advanced reference material that can be skipped on first reading.

1. Clear Attribute

Description: Clears a slot by removing all values from it.

Declaration:

```
attr->clear() : C[0..1]
```

Static rules: The attribute's multiplicity must allow zero cardinality (that is, the lower bound of the multiplicity must be zero).

Run-time semantics: Removes all values from the slot. If the slot is already empty (without values), the action has no effect.

Example:

```
aProduct.picture->clear();
```

2. Remove Value from Attribute

Description: Removes all occurrences of the given value from a slot.

Declaration: For an attribute whose multiplicity allows zero cardinality:

```
attr->remove(val:T[0..1]) : C[0..1]
```

For an attribute whose multiplicity does not allow zero cardinality:

```
attr->remove(val:T[0..1]) : C[0..1] throws LowerBoundViolationException
```

Static rules: The multiplicity of the attribute must have different lower and upper bounds. In other words, the multiplicity of the attribute must allow at least two cardinalities. If the attribute's multiplicity allows zero cardinality, there is no exception specification.

Run-time semantics: If the argument has no value (that is, its cardinality is zero), the action does not affect the slot. Otherwise, the action searches for all occurrences of the given value in the slot. If the value does not exist in the slot, the action does not affect the slot. Otherwise, if the lower multiplicity bound would be violated by the removal of the values, an exception of type `LowerBoundViolationException` is raised and the slot remains unaffected. Otherwise, all occurrences of the value are removed from the slot. The comparison of the values is value-based.

Examples:

```
Picture aPicture = ...;  
aProduct.picture->remove(aPicture);
```

3. Remove One Occurrence of Value from Attribute

Description: Removes one occurrence of the given value from a slot.

Declaration: For an attribute whose multiplicity allows zero cardinality:

```
attr->removeOne(val:T[0..1]) : C[0..1]
```

Part III: Concepts

For an attribute whose multiplicity does not allow zero cardinality:

```
attr->removeOne(val:T[0..1]) : C[0..1] throws LowerBoundViolationException
```

Static rules: The multiplicity of the attribute must have different lower and upper bounds. In other words, the multiplicity of the attribute must allow at least two cardinalities. If the attribute's multiplicity allows zero cardinality, there is no exception specification.

Run-time semantics: If the argument has no value (that is, its cardinality is zero), the action does not affect the slot. Otherwise, the action searches for the given value in the slot. If the value does not exist in the slot, the action does not affect the slot. Otherwise, if the lower multiplicity bound would be violated by the removal of one value, an exception of type `LowerBoundViolationException` is raised and the slot remains unaffected. Otherwise, one occurrence of the value is removed from the slot. For ordered attributes, the first occurrence in order is removed. For unordered attributes, it is undefined which occurrence is removed. The comparison of the values is value-based. Consequently, for unique attributes, this action has the same effect as the Remove Value from Attribute action.

Example:

```
aProduct.picture->removeOne(aPicture);
```

4. Remove Value from Ordered Attribute

Description: Removes a single value from the given position in a slot of an ordered attribute. Variations of this action remove the first and the last value from a slot of an ordered attribute.

Declaration: For an attribute whose multiplicity allows zero cardinality:

```
attr->removeAt(pos:Integer[0..1]) : C[0..1] throws MultiplicityViolationException  
attr->removeFirst() : C[0..1]  
attr->removeLast() : C[0..1]
```

For an attribute whose multiplicity does not allow zero cardinality:

```
attr->removeAt(pos:Integer[0..1]) : C[0..1] throws MultiplicityViolationException  
attr->removeFirst() : C[0..1] throws LowerBoundViolationException  
attr->removeLast() : C[0..1] throws LowerBoundViolationException
```

Static rules: The attribute must be ordered. The multiplicity of the attribute must have different lower and upper bounds. In other words, the multiplicity of the attribute must allow at least two cardinalities.

Run-time semantics:

- ❑ `removeAt` — Tries to remove the value at the position given with the argument `pos`. The positions are zero-based (the first value in the collection has the position 0).
If the position is not given (the cardinality of the argument `pos` is zero), the effect is the same as for the `removeFirst` variation of this action.
If the given position `pos` has a value that is not included in the multiplicity of the attribute, the slot remains unaffected and the exception `MultiplicityViolationException` is raised. In particular, if the given position is less than the lower bound of the multiplicity allows (in effect, it is negative), the exception is of type `LowerBoundViolationException`, which is a subtype of

`MultiplicityViolationException`. Similarly, if the given position is greater than the upper bound of the multiplicity allows, the exception is of type `UpperBoundViolationException`, which is a subtype of `MultiplicityViolationException`.

If the given position `pos` has a value that is out of the range defined by the current cardinality of the slot (but is still included in the attribute's multiplicity), the slot remains unaffected.

Otherwise, if the removal of one value would violate the lower multiplicity bound of the attribute, an exception of type `LowerBoundViolationException` is raised and the slot remains unaffected.

Finally, if none of the previous occurred, the value at the position given by the argument `pos` is removed from the slot.

- ❑ `removeFirst`, `removeLast` — If the slot is empty (its current cardinality is zero — that is, it has no values), the slot remains empty. Otherwise, if the removal of one value from the slot would violate the lower multiplicity bound, an exception of type `LowerBoundViolationException` is raised and the slot remains unaffected. Otherwise, the first/last value from the ordered slot is removed.

Examples:

```
aProduct.picture->removeFirst();  
  
aProduct.picture->removeLast();  
  
aProduct.picture->removeAt(3);
```

5. Set (Assign) Attribute

Description: Assigns the value(s) of the result of an expression to a slot.

Declaration: There are two equivalent notations for the same action:

```
attr->set(expression) : C[0..1]  
attr = expression : C[0..1]
```

Static rules: The expression that is assigned to the slot must conform to the attribute (both being typed multiplicity elements).

If the attribute is single-valued, the expression can be the special symbol `null`.

Run-time semantics: First, all existing values of the slot are removed from the slot. Then, the values from the source expression, which is a collection of values conforming to the target attribute, are added to the target slot. If the attribute is ordered, the order of the values is preserved. If the result of the expression has no value (that is, its cardinality is zero), the target slot also becomes empty (the effect is equivalent to the Clear Attribute action).

In case the *expression* is the special symbol `null`, the effect of the action is the same as for the Clear Attribute action.

All this is done as an atomic action, without checking multiplicity and uniqueness constraints, because they are guaranteed by the static conformance rules.

Part III: Concepts

The same action is performed at the initialization of a slot.

Examples:

```
Course c = ...;
c.name = "Computers for Dummies";

c.name = null;

c.name->set("Computers for Dummies");
```

6. Add Value to Attribute

Description: Adds the given value to a slot.

Declaration: For an attribute with unlimited upper multiplicity bound:

```
attr->add(val:T[0..1]) : C[0..1]
```

For an attribute with a limited upper multiplicity bound:

```
attr->add(val:T[0..1]) : C[0..1] throws UpperBoundViolationException
```

Static rules: The multiplicity of the attribute must allow at least two cardinalities. If the attribute's multiplicity allows unlimited cardinality, there is no exception specification.

Run-time semantics: If the argument has no value (that is, its cardinality is zero), the action does not affect the slot. For a unique unordered attribute, if the given value already exists in the slot, the action does not affect the slot. For a unique ordered attribute, if the given value already exists in the slot, the action moves the value to the end of the slot. Otherwise (if the attribute is non-unique or the value does not exist in the slot), if the limited upper multiplicity bound would be violated by the addition of the value, an exception of type `UpperBoundViolationException` is raised and the slot remains unaffected. Otherwise, the action adds the value to the slot. For an ordered attribute, the value is added at the end. The comparison of the values is value-based.

Example:

```
aProduct.picture->add(aPicture);
```

7. Add Value to Ordered Attribute

Description: Adds a value to the given position in a slot of an ordered attribute. Variations of this action add the value at the first and the last position in a slot of an ordered attribute.

Declaration: For an attribute with unlimited upper multiplicity bound:

```
attr->addAt(val:T[0..1], pos:Integer[0..1]) : C[0..1]
    throws LowerBoundViolationException
attr->addFirst(val:T[0..1]) : C[0..1]
attr->addLast(val:T[0..1]) : C[0..1]
```

For an attribute with a limited upper multiplicity bound:

```

attr->addAt(val:T[0..1], pos:Integer[0..1]) : C[0..1]
    throws MultiplicityViolationException
attr->addFirst(val:T[0..1]) : C[0..1] throws UpperBoundViolationException
attr->addLast(val:T[0..1]) : C[0..1] throws UpperBoundViolationException

```

Static rules: The attribute must be ordered. Additionally, the multiplicity of the attribute must allow at least two cardinalities.

Run-time semantics:

- ❑ `addAt` — Tries to insert the value given in the argument `val` at the position given in the argument `pos`. The positions are zero-based (the first value in the collection has the position 0).
 - If the value is not given (that is, the cardinality of the argument `val` is zero), the action does not affect the slot.
 - If the position is not given (that is, the cardinality of the argument `pos` is zero), the effect is the same as for the `addLast` variation of this action.
 - If the given position `pos` has a value that is not included in the multiplicity of the attribute, the slot remains unaffected and the exception `MultiplicityViolationException` is raised. In particular, if the given position is less than the lower bound of the multiplicity allows (in effect, it is negative), the exception is of type `LowerBoundViolationException`, which is a subtype of `MultiplicityViolationException`. Similarly, if the given position is greater than the upper bound of the multiplicity allows, the exception is of type `UpperBoundViolationException`, which is a subtype of `MultiplicityViolationException`.
 - If the given position `pos` has a value that is out of the range defined by the current cardinality of the slot (but is still included in the attribute's multiplicity), the effect is the same as for the `addLast` variation of this action.
 - If the attribute is unique and the given value already exists in the slot, the action moves the value to the given position in the slot.
 - Otherwise, if the addition of one value would violate the limited upper multiplicity bound of the attribute, an exception of type `UpperBoundViolationException` is raised and the slot remains unaffected.
 - Finally, if none of the previous occurred, the given value is inserted at the position given by the argument `pos`. The rest of the values are shifted one place toward the end of the collection.
- ❑ `addFirst, addLast` — If the given value is empty (its cardinality is zero — that is, it has no values), the slot remains unaffected. If the attribute is unique and the given value already exists in the slot, the value is moved to the first/last position in the slot. Otherwise, if the addition of one value to the slot would violate the limited upper multiplicity bound, an exception of type `UpperBoundViolationException` is raised and the slot remains unaffected. Otherwise, the given value is inserted into the first/last position in the slot of the ordered attribute.

Examples:

```

aProduct.picture->clear();
aProduct.picture>addLast(aPic2);
aProduct.picture>addFirst(aPic1);
aProduct.picture>addAt(aPic3,15);

```

Part III: Concepts

The slot picture of the object referred to by the variable `aProduct` will ultimately have the following values in the given order: `aPic1, aPic2, aPic3`. Note that the action `addAt(aPic3, 15)` has the same effect as would the action `addLast(aPic3)` because the cardinality of the slot is less than the given position 15.

The Symbol null

The special symbol `null` represents absence of value in a single-valued result of an action on an attribute. Consider the following expression:

```
attr==null
```

This is equivalent to the following expression:

```
attr->size()==0
```

Similarly, consider the following expression:

```
attr!=null
```

This is equivalent to the following expression:

```
attr->size()!=0
```

The same symbol can be used to check the presence or absence of a value in a result of other actions on attributes. Following is an example:

```
if (aProduct.picture->first()==null) ...
```

Section Summary

❑ Clear Attribute:

```
attr->clear() : C[0..1]
```

❑ Remove Value from Attribute:

```
attr->remove(val:T[0..1]) : C[0..1]
throws LowerBoundViolationException
```

❑ Remove One Occurrence of Value from Attribute:

```
attr->removeOne(val:T[0..1]) : C[0..1]
throws LowerBoundViolationException
```

❑ Remove Value from Ordered Attribute:

```
attr->removeAt(pos:Integer[0..1]) : C[0..1]
throws MultiplicityViolationException
attr->removeFirst() : C[0..1]
throws LowerBoundViolationException
```

- ```

attr->removeLast() : C[0..1]
throws LowerBoundViolationException

 Set (Assign) Attribute:
attr->set(expression) : C[0..1]
attr = expression : C[0..1]

 Add Value to Attribute:
attr->add(val:T[0..1]) : C[0..1]
throws UpperBoundViolationException

 Add Value to Ordered Attribute:
attr->addAt(val:T[0..1], pos:Integer[0..1]) : C[0..1]
throws MultiplicityViolationException
attr->addFirst(val:T[0..1]) : C[0..1]
throws UpperBoundViolationException
attr->addLast(val:T[0..1]) : C[0..1]
throws UpperBoundViolationException

 The special symbol null represents absence of value in a single-valued result
of an action on attribute:
if (attr->first()==null) ...

```

## ***Freezing and Unfreezing Attributes<sup>7</sup>***

### **1. Freeze Attribute**

*Description:* Freezes the attribute value. After that, it cannot be modified.

*Declaration:*

```
attr->freeze()
```

*Static rules:* This action is applicable to every attribute.

*Run-time semantics:* Freezes the attribute. After that, every Write Attribute action on that attribute will raise an exception, unless it is a non-static attribute of a data type. Freezing has no effect on non-static attributes of data types.

*Example:*

```
aProduct.picture->freeze();
```

---

<sup>7</sup>This is advanced reference material that can be skipped on first reading.

## Part III: Concepts

---

### 2. Unfreeze Attribute

*Description:* Unfreezes the attribute value. After that, it can be modified, unless it is read-only.

*Declaration:*

```
attr->unfreeze() throws WritingReadOnlyAttributeException
```

*Static rules:* This attribute cannot be read-only.

*Run-time semantics:* Unfreezes the attribute. After that, its value can be modified by Write Attribute Actions, unless it is read-only. This action raises an exception of type `WritingReadOnlyAttributeException` for read-only attributes.

*Example:*

```
aProduct.picture->unfreeze();
```

#### Section Summary

- Freeze Attribute:

```
attr->freeze()
```

- Unfreeze Attribute:

```
attr->unfreeze()
throws WritingReadOnlyAttributeException
```

## Iterations on Attributes<sup>8</sup>

Similar to variables, there is a construct in OOIS UML to iterate through values in a slot of an attribute:

```
attr->forEach(v, T) statement
```

The static rules for this construct are the following. `attr` represents a slot. The construct `forEach` is applicable to every attribute. The first identifier `v` within the parentheses is actually a declaration of a new variable introduced into the scope of the body of the `forEach` construct. It is used to refer to the current value in the iteration. The second identifier `T` within the parentheses is a type that must conform to the type of the attribute. The type can be omitted; in that case, the type of the attribute is assumed. `v` is implicitly declared as a variable of type `T` and multiplicity 1..1. `statement` is a statement in the OOIS UML native detail-level language; it can be a compound statement (a block) enclosed in curly brackets {}.

The run-time semantics of this construct are the following. For each value in the slot `attr` that is of type `T` (substitution is always assumed), the statement `statement` is executed. Therefore, the type `T` serves as

---

<sup>8</sup>This is advanced reference material that can be skipped on first reading.

a type filter for the values in the slot. Only the values of type  $T$  (which must be the same or a subtype of the attribute's type), including subtypes, are visited; the others are skipped. Within that statement,  $v$  refers to the value currently being visited. The iterations are performed sequentially. The next iteration is not started before the previous one is completed. For ordered attributes, the order of iteration is defined by the order of the values in the slot. For unordered attributes, the order is undefined. For an empty slot (with the cardinality zero), the construct has no effect.

For ordered attributes, there is another construct that visits the elements in the reverse order:

```
attr->forEachReverse(v,T) statement
```

This construct has all other static rules as the construct `forEach`, except that the attribute must be ordered. This construct has the same run-time semantics as the construct `forEach`, except that the values in the slot are visited in the reverse order.

### Section Summary

- ❑ An iteration through an attribute by filtering only elements of type  $T$  and visiting each value  $v$ :  

```
attr->forEach(v,T) statement
```
- ❑ An iteration through an ordered attribute in the reverse order:  

```
attr->forEachReverse(v,T) statement
```

## Access to Slots Through Reflection<sup>9</sup>

Properties of classifiers (including attributes) are manifested as slots in classifiers' instances. A slot of a classifier instance (including attribute value) can be identified dynamically through the reflection mechanism. Every instance of a classifier can respond to the operation `slot`, which accepts a string parameter that specifies the name of the property, and the result of which refers to the property's slot. For each of the attribute actions specified so far, there is a corresponding operation that can be invoked on that result. Consider the following example:

```
Course c = ...;
c.slot("name").set("Computers for Dummies");
```

The operation `slot` accepts an argument of type `Text` that identifies the property (as a named element) by its name. All namespace resolution rules are applied for the specified name. The property must be accessible from the place of invocation of this operation. Otherwise, the operation raises an exception of the type `InaccessibleNamedElementException`. If the property with the specified name does not exist at all, the operation raises an exception of the type `NonexistentNamedElementException`.

---

<sup>9</sup>This is advanced reference material that can be skipped on first reading.

## Part III: Concepts

---

The result of this operation is of the type `Slot`, whose instances identify slots of classifier instances. This OOIS UML built-in classifier has operations for all described attribute actions, having the same signature (the name and the types of arguments and result).

### Section Summary

- ❑ Slots of a classifier instance can be identified dynamically through the reflection mechanism. Every instance of a classifier can respond to the operation `slot`, which accepts a string parameter that specifies the name of the property, and the result of which is of type `Slot` and refers to the property's slot of that instance.

## ***Implementation in Other Detail-Level Languages<sup>10</sup>***

When a method is implemented in a detail-level language other than the OOIS UML native one, the actions on attributes are specified using the OOIS UML API for the language. In that API, the slot is accessed using the standard notation for accessing data fields of objects in that language, while the action is specified as the operation on (the object referred to by) that field, with the appropriate arguments. The rules for implementing parameters of actions in terms of the host detail-level language concepts may be the same as for other operations.

For example, consider the following code in the OOIS UML native detail-level language:

```
Product aProduct = ...;
Picture aPicture = ...;
aProduct.picture->add(aPicture);
```

This may be written in C++ as follows:

```
Product* aProduct = ...;
Picture* aPicture = ...;
aProduct->picture->add(aPicture);
```

Or, it may be written in Java or C# as follows:

```
Product aProduct = ...;
Picture aPicture = ...;
aProduct.picture.add(aPicture);
```

As you can see, the slot in the host programming language is implemented as a data field of the object, and is accessed using the native notation of the language (-> in C++ or . in Java and C#). Actually, it is an instance of a class implemented in that host language as part of the API. The class has operations that correspond to all attribute actions (as the operation `add`, in this case). Because the parameter of this action

---

<sup>10</sup>This is advanced reference material that can be skipped on first reading.

is of multiplicity 0..1, a simple pointer/reference of the host language is passed as the argument of that operation.

This similarly holds true for all other actions, except for the pure Read Attribute and Assign (Set) Attribute actions. Namely, because of the background processing that these actions on attributes of classes perform, a simple access to a data field of an object in the host language is not sufficient. Consider the following:

```
...aProduct.picture...
```

For reading the slot `picture` of the object referred to by the variable `aProduct`, you should explicitly write the following in C++:

```
...aProduct->picture->read()...
```

You could also write the following in Java and C#:

```
...aProduct.picture.read()...
```

Similarly, the assignment notation should not be used to write a value to the slot:

```
aProduct.picture = ...;
```

Instead, the explicit `set` operation should be used:

```
aProduct.picture.set(...);
```

The same approach can be used for actions on attributes of classes and of data types. However, because attributes of data types have the semantics very similar to that of variables (in terms of their scope and usage), an implementation of the profile may opt to implement attributes of data types in the same manner as variables. This means, for example, that a single-valued attribute of a data type can be implemented as a simple reference, while a multi-valued attribute of a data type can be implemented as a collection. Both of them are accessed as pure data fields of the enclosing data type instance in the host language.

### Section Summary

- ❑ The OOIS UML API for the host detail-level programming language can provide the abstraction of a slot, with the operations that implement the actions on attributes.
- ❑ Exceptions to this rule are the Read Attribute and Assign (Set) Attribute actions. They should be specified using the explicit notation.
- ❑ An implementation of the profile may opt to implement attributes of data types as data fields of objects using the same approach as for variables.



# 10

## Associations

*Associations* are structural relationships between classes that describe sets of links between objects. Associations can have two or more ends. The associations with two ends (that is, binary associations) are most frequently used in practice and are discussed in this chapter first. Many modeling aspects and run-time effects exist only for the binary associations. In addition, associations can have more than two ends, when they are called N-ary associations. Such associations are also examined in this chapter. Finally, the concept of association class, which represents a model element that is a class and an association at the same time, is described in this chapter.

### Binary Associations

This section examines the details of binary associations.

#### ***Binary Associations and Links***

Association is a structural relationship that describes a set of *links*, whereby a link connects objects. Association is a semantic relationship between classes that involves connections among their instances. Associations can be used to model a number of different kinds of relationships between the entities represented with objects, such as, for example, pairings, correspondences/mappings, communication paths, and so on.

Although UML supports associations with more than two ends, *binary associations* are by far the most common. A binary association has just two ends, each of which is connected to a class. Every link of a binary association, thus, has two ends, each of which is connected to an object of the corresponding class. More formally, a link of a binary association is a pair of values, each of which refers to an object of the corresponding class. For example, a link of the association `assignment` shown in Figure 10-1a is a pair of values (`d, p`), whereby `d` refers to an object of the class `Department`, and `p` refers to an object of the class `Person`.

## Part III: Concepts

---

The classes at association ends play certain *roles*, meaning that the orientation of the links is relevant. The role is specified as the name of the association end. The same class may play different roles in different associations. For example, the class `Person` in Figure 10-1a plays the role of an employee in the association `assignment` and the role of a manager in the association `management`. Consequently, a given object of the class `Person` can be linked by two links of the two associations, playing the role of an employee in one and of a manager in the other.

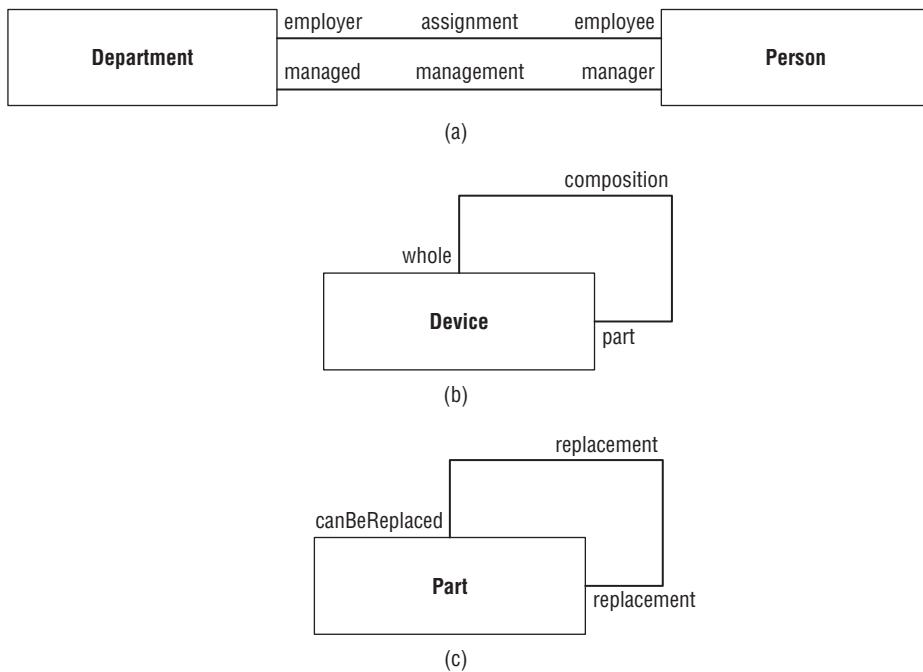


Figure 10-1: Binary associations and roles. (a) The same class can play multiple roles in different associations, or (b, c) even in the same association.

Furthermore, an association may have the same class at both of its ends, as shown in Figure 10-1b. However, the roles of the association ends clearly specify the roles of the objects at the ends of a link (which implies that the orientation of the link is relevant). According to the model in Figure 10-1b, the object of the class `Device` at one end of a link plays the role of a whole, whereas the object of the same class at the opposite end plays the role of a part, in the relationship that conceptualizes composition of devices. This similarly holds true for the model in Figure 10-1c. Simply said, the set of ends of an association (as model elements related to the association) is ordered in a formal sense, although this is usually of no particular importance.

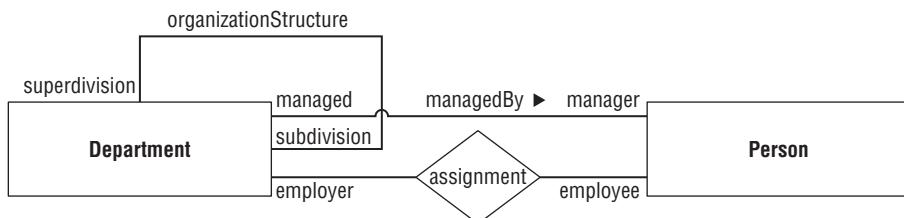
As you can see, a link of a binary association is inherently “oriented,” meaning that the “position” of the linked objects is relevant and asymmetric. Mathematically, as already stated, a link of a binary association is an ordered pair of values that refer to objects  $(x, y)$ , and not a set of values  $\{x, y\}$ , in which the position (or order) of the values would be irrelevant ( $\{x, y\} = \{y, x\}$ , while  $(x, y) \neq (y, x)$ ). For example, the association `replacement` in Figure 10-1c conceptualizes the fact that a part can be replaced by another

part. This means that a link  $(x, y)$  (whereby the part  $x$  plays the role `canBeReplaced` and  $y$  plays the role `replacement`) represents the fact that the part  $x$  can be replaced with the part  $y$ , but not vice versa.

However, in many cases where the roles of the objects participating in a link are unambiguous, the ordering of objects in a link (as a pair) is irrelevant in practice. For example, it is equivalent to say that “a person is employed by a company” and that “the company employs the person,” although the “positions” of the linked objects is inherently determined.

In UML, an association is a classifier, because it classifies links into groups. Consequently, association is also a packageable element, meaning that it is owned by a package. Ultimately, association is a named element. According to the UML style guidelines, the name of an association should start with a lowercase character.

In UML diagrams, an association may be drawn as a diamond with a solid line for each association end connecting the diamond to the class at its end, as shown for the association `assignment` in Figure 10-2. However, a binary association is usually depicted as a solid line connecting two classes, or a single class to itself (the two ends are distinct). A line may consist of two or more joined segments that individually have no semantic significance, but serve only for graphical convenience.



**Figure 10-2: Notation for associations**

The association’s name can be shown as text near or within the association diamond symbol, or near the midpoint of the solid line representing a binary association, but not so near to an end as to be confused with the end’s name. When a binary association is shown as a solid line, an optional filled triangular arrowhead next to or in place of the name of the association, and pointing along the line to one end, indicates that end to be the second in the ordered pair of the ends of the association (and also in the links as ordered pairs). The arrowhead also indicates the direction of reading and interpreting the name of the association. It suggests that the association is associating the end away from the direction of the arrowhead with the end to which the arrowhead is pointing, with the meaning given by the association name. For the association `managedBy` in Figure 10-2, the interpretation is “`Department` is managed by `Person`”. An association end is represented as the connection between the line of an association and the symbol of the connected class. A string with the name of the association end (that is, the role) may be placed near the end of the line. It can also be suppressed.

When two line segments of association symbols cross in a diagram, the crossing may optionally be shown with a small semicircular jog to indicate that the lines do not intersect, as in electrical circuit diagrams. Figure 10-2 shows an example. However, such cases should be avoided, because the diagrams should be kept simple and clear. Lines may be drawn using various styles, including orthogonal segments, oblique segments, and curved segments. The particular set of line styles is a user or tool choice, and has no semantic significance.

### Section Summary

- ❑ A *binary association* is a relationship that has two ends, each of which is connected to a class.
- ❑ A binary association describes a collection of links, which are pairs of values that refer to objects of the related classes. Links are instances of associations.
- ❑ The classes at association ends play certain *roles*.
- ❑ An association is a classifier, a packageable element, and a named element.
- ❑ In UML diagrams, an association may be drawn as a diamond with a solid line for each association end connecting the diamond to the class at its end. A binary association is usually depicted simply as a solid line connecting the classes.

## Association Ends and Properties

Put formally, an end of a binary association is an element of a model that specifies a mapping for each object of the class at the opposite end to a (possibly empty or single-valued) collection of objects of the class at that end. In the model shown in Figure 10-3, for example, the association end `employee` defines a mapping for each object  $d$  of `Department` to a collection of objects of `Person` — those that are currently linked to  $d$  by the links of the association `assignment`. Obviously, that mapping deals with the object space and is dynamic, meaning that it changes over time. At a certain moment in time, the mapping `employee` for an object  $d$  of `Department` results in one collection. At another moment in time, it may result in another collection, because the actions on the object space have changed that space in the meantime — links have been created or deleted. The actions that work with links actually affect that mapping.

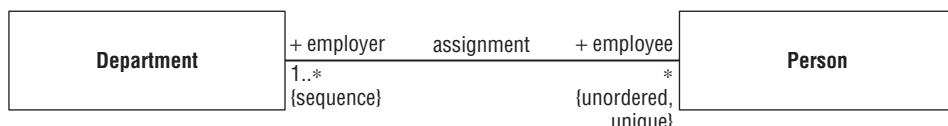


Figure 10-3: Notation for association ends

An association end is a typed multiplicity element whose type is the class at that end. For example, in the model shown in Figure 10-3, the type of the association end `employee` is `Person`, while the type of the association end `employer` is `Department`. The type of an association end constrains the elements of the collection that is the result of the mapping specified by the association end. The type of every association end must be accessible from the namespace that owns the association.

As a multiplicity element, an association end has other characteristics: multiplicity, uniqueness, and ordering.<sup>1</sup> The multiplicity and uniqueness of an association end constrain the cardinality and

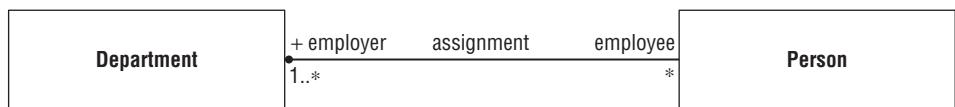
<sup>1</sup>It should be noted that ordering of one particular association end has nothing to do with the internal ordering of the set of ends of one association within the model. The former means that the collections of linked objects have relevant ordering, whereas the latter is an internal matter of mathematical interpretation of the fact that each end in the association has a different role.

uniqueness of the collection that is the result of the mapping specified by the end. These constraints are implicitly preserved or checked, and enforced by every action that affects that collection. According to the combination of uniqueness and ordering, the collection can represent a set (unique, unordered), an ordered set (unique, ordered), a bag (non-unique, unordered), or a sequence (non-unique, ordered). By default, an association end in OOIS UML has a multiplicity \* (unlimited), is unordered, and unique. In other words, the resulting collection is a set by default.

Graphically, an association end is represented by the point where the line of the association and the symbol of the connected class meet (see Figure 10-3). A string with the name of the association end may be placed near the end of the line. The name is suppressible. Various other notations can be placed near the end of the line. In particular, these are the multiplicity and a string with the end's characteristics enclosed in curly braces, including the following:<sup>2</sup>

- {bag} to show that the end is non-unique and unordered, and thus represents a bag.
- {sequence} or {seq} to show that the end is non-unique and ordered, and thus represents a sequence.

An end of a binary association, as a model element, can *belong to* (that is, be owned by) either the association itself, or the class connected at the opposite end of the same association. The ownership of an association end is a relationship between the association end and the association or the class at the opposite end. For example, the association end `employer` in Figure 10-4 belongs to the class `Person`, whereas the association end `employee` belongs to the association `assignment`. Graphically, the ownership of an association end by the class at the opposite end is indicated by a small filled circle at that end, called a *dot*. The dot is drawn integral to the line, at the point where it touches the symbol of the class, but does not overlap the symbol of the class.



**Figure 10-4:** In this example of association end ownership, the end `employer` is owned by the class `Person`, whereas the end `employee` is owned by the association.

The ownership of the association end has certain semantic implications. If the association end belongs to the association, but not to the class at the opposite end, then the association end is not a property of that class. Consequently, the objects of that class do not have the corresponding slot for that end that holds the collection of linked objects. For the example shown in Figure 10-4, the association end `employee` belongs to the association `assignment` and not to the class `Department`, so the class `Department` does not have the property `employee`, and its objects do not have the corresponding slot. Therefore, the dot notation `aDepartment.employee` to refer to a slot cannot be used in the textual notation, and the actions that work on properties cannot refer to the slot `employee` of an object.

If the association end belongs to the opposite class, then it is also a property (as a feature) of that class. Consequently, the class's objects will have the corresponding slot. For the same example given

<sup>2</sup>These are somewhat obsolete notations that are deprecated in OOIS UML. They are mentioned here for the sake of compatibility with older versions of UML. Explicit modifiers `ordered/unordered` and `unique/nonunique` should be used instead.

## Part III: Concepts

---

in Figure 10-4, the association end `employer` belongs to the class `Person` and not to the association `assignment`, so the class `Person` does have the property `employer`, and its objects do have the corresponding slot. Therefore, the notation `aPerson.employer` can be used in the textual notation, and the actions that work on properties can refer to the slot `employer` of an object. Additionally, the visibility of the association end is the visibility of the property, with implications as for any other feature of the class.

If the association end belongs to a class (and is, therefore, a property of that class), any property strings that apply to an attribute can be shown near the end of the line. For example, the visibility sign (+, -, or #) can be shown in front of the end's name, as shown in Figure 10-4.

Standard UML does not mandate the use of explicit association end ownership notation (the dot). It leaves it to the tool or the modeler to choose whether or not to use it at all. If it is selected, however, it must be used consistently throughout the entire model. The dot can be omitted only for ends that are owned by associations. In this way, in the models where the notation is used, the absence of the dot on certain ends does not leave the ownership of those ends ambiguous, as it is the case in the models where the notation is not used.

In OOIS UML, by default, an association end belongs to the opposite class and not to the association.<sup>3</sup> For that reason, the dot notation is not used here for association ends, except in special cases when this default is to be overridden. In such cases, the dot notation will be used for the end that belongs to the class, whereas the missing dot at the opposite end would emphasize the exception, as shown in Figure 10-4. Such an approach does not allow you to distinguish the associations whose ends are both owned by the classes (the default) from the associations where both ends are owned by the association. The latter is, however, very rare in practice.

In addition, if an association end belongs to the opposite class, the class at that association end must be accessible from the namespace owning the opposite class.<sup>4</sup> This is because the association end is a property owned by the opposite class, and is a typed element, whose type is the class at the considered association end. For the example in Figure 10-4, the association end `employer` belongs to the class `Person`, so the class `Department` must be accessible from the package owning the class `Person`. This is because the class `Person` owns the property `employer` of type `Department`.

On the other hand, if the association end belongs to the association, there is no such restriction, because the association end is not a property of a class. In that case, the class at the opposite end is actually independent of the class at the considered association end, meaning that the existence of the association end does not affect the class (because the end does not contribute a feature to the class). For the example in Figure 10-4, the association end `employee` belongs to the association `assignment`, so the class `Person` does not need to be accessible from the package owning the class `Department` (unless the very association is owned by that class), and the class `Department` is independent of (and unaffected by) the existence of the association end.

This fact usually serves as a motivation for declaring an association end to belong to the association, and not to the opposite class, as it does by default. If the class at the opposite end does not need the property as a manifestation of the association end, and should not be dependent on the class at the considered association end, then the association end is declared to belong to the association. This is usually the case

---

<sup>3</sup>In standard UML, there is no default ownership.

<sup>4</sup>These constraints on the model are specific to the OOIS UML profile. They do not exist in standard UML.

when the class belongs to a generic package, is designed for reuse in many other contexts, and should not depend on the classes and associations that introduce association ends connected to it. In other words, it is desired that the class does not need to be “recompiled” because it adopts new features every time it is reused and associated to another specific class. As you can see, the motivation for using it is purely practical and implementation-oriented.

## Section Summary

- ❑ An end of a binary association is an element of a model that specifies a mapping for each object of the class at the opposite end to a (possibly empty or single-valued) collection of objects of the class at that end.
- ❑ An *association end* is a typed multiplicity element whose type is the class at that end.
- ❑ An association end, as a model element, can be owned by the association itself, in which case the class at the opposite end does not have a property for that end.
- ❑ An association end can be (and is by default) owned by the class at the opposite end, in which case that class does have a property for that end.

## Semantics of Binary Associations and Association Ends<sup>5</sup>

The descriptions of association and association end given thus far mostly refer to the modeling rules. Let's now consider the precise run-time semantics, especially in correlation with the notion of association ends and their uniqueness and ordering. Figure 10-5 shows a sample model that will be used for the definitions and descriptions that follow. Let's assume that the association end *px* is the first, and the association end *py* is the second in the order of ends of the association *assxy* (remember that association ends are ordered within an association and that the small filled triangle in diagrams points to the last end in the order). In these discussions, it is irrelevant whether an association end belongs to the opposite class or to the association — the only difference is whether the opposite class does or does not have the corresponding property. The basic semantics of association and its ends are independent of this feature.



Figure 10-5: Sample model for describing the semantics of associations and association ends

An association describes links, whereby a link is an ordered pair whose values refer to objects. For the considered sample model in Figure 10-5, the association *assxy* describes links in the form  $(x, y)$ , whereby

<sup>5</sup>The definition of standard UML in [UML2] does not define the semantics of associations and association ends in a clear and unambiguous way. The definitions given here were first proposed in the article/paper [Milicev, 2007] and are specific to OOIS UML.

## Part III: Concepts

---

$x$  is an object of  $X$  and  $y$  is an object of  $Y$ . The collection of the existing links of an association at an arbitrary point in run-time will be referred to as the *association extent* at that point in time. The association extent is not a set (that is, it allows duplicates of pairs). Additionally, it is never ordered. Consequently, the collection of links of an association at some point in time is a bag.

For example, if the subscripts of  $x$  and  $y$  indicate different objects of  $X$  and  $Y$ , respectively, at a certain point  $t_1$  in run-time, the extent of the association  $\text{assxy}$  may be the following (the ordering of the pairs in the collection is not relevant):

```
assxyt1 = [(x1, y1), (x1, y2), (x1, y3), (x2, y1), (x2, y3), (x2, y1), (x3, y2)]
```

Just for the purpose of reference, the extent will be denoted with  $\text{assxy}_{t_i}$ , with the precise meaning “the collection of live links of the association  $\text{assxy}$  at the time  $t_i$ .” The elements of collections (that is, bags) will be enclosed in square brackets [].

The Create Link action executed for this association basically adds a new pair to this collection. For example, if a Create Link action with the parameters  $(x_1, y_2)$  is executed twice for this association  $\text{assxy}_{t_1}$ , the collection becomes (the newly added elements are underlined for better readability):

```
assxyt2 = [(x1, y1), (x1, y2), (x1, y3), (x2, y1), (x2, y3), (x2, y1), (x3, y2),
 (x1, y2), (x1, y2)]
```

An association end defines a mapping for each object of the class at the opposite end to a collection of objects of the class at that end. If the association end is non-unique, the collection represents the corresponding values from all those and only those pairs from the association extent that have the given object as the corresponding coordinate. For the given example, assuming that  $py$  is non-unique, for an object  $x$  of  $X$ , the mapping  $py(x)$  is derived from the association extent to represent the collection of the second coordinates  $y$  of all those and only those pairs from the extent that have  $x$  as the first coordinate. This similarly holds for non-unique  $px$ . Therefore, at the time  $t_1$ , the mappings are as follows (the ordering of collection elements is irrelevant):

```
py(x1)t1 = [y1, y2, y3]
py(x2)t1 = [y1, y3, y1]
py(x3)t1 = [y2]
px(y1)t1 = [x1, x2, x2]
px(y2)t1 = [x1, x3]
px(y3)t1 = [x1, x2]
```

Similarly, at the time  $t_2$ , the mappings are as follows (the ordering of collection elements is irrelevant):

```
py(x1)t2 = [y1, y2, y3, y2, y2]
py(x2)t2 = [y1, y3, y1]
py(x3)t2 = [y2]
px(y1)t2 = [x1, x2, x2]
px(y2)t2 = [x1, x3, x1, x1]
px(y3)t2 = [x1, x2]
```

Note that if an association end belongs to the opposite class, then the class owns that property and the dot notation can be used to refer to the slot. However, it represents the same collection defined by

the association end and, thus, has the same semantics. For example, if  $\text{py}$  belongs to  $x$ , then the slot  $x.\text{py}$  at a certain time represents the same collection defined by the mapping  $\text{py}(x)$  at that time:

$$\begin{aligned} x_1.\text{py}_{t1} &= \text{py}(x_1)_{t1} = [y_1, y_2, y_3] \\ x_2.\text{py}_{t1} &= \text{py}(x_2)_{t1} = [y_1, y_3, y_1] \\ x_3.\text{py}_{t1} &= \text{py}(x_3)_{t1} = [y_2] \end{aligned}$$

The Destroy Link action specifies a pair of values to be removed from the association. For each of the non-unique association ends of the association, you can specify whether all values from the collection defined by that end should be removed. Let's use the symbol  $\forall$  as a prefix of a value in a pair to indicate that all pairs having that value at that coordinate should be removed. If this symbol is missing, only one such pair should be removed. For example:

- $\text{DestroyLink}(\text{assxy}, (x_1, y_2))$  means “remove one pair  $(x_1, y_2)$  from the extent of  $\text{assxy}$ .”
- $\text{DestroyLink}(\text{assxy}, (\forall x_1, y_2))$  means “remove those and only those pairs  $(x_1, y_2)$  from the extent of  $\text{assxy}$ , such that  $\text{px}(y_2)$  does not contain  $x_1$  anymore.” It is easy to see that this means “remove all pairs  $(x_1, y_2)$  from the extent of  $\text{assxy}$ .”
- $\text{DestroyLink}(\text{assxy}, (x_1, \forall y_2))$  means “remove those and only those pairs  $(x_1, y_2)$  from the extent of  $\text{assxy}$ , such that  $\text{py}(x_1)$  does not contain  $y_2$  anymore.” It is easy to see that this means “remove all pairs  $(x_1, y_2)$  from the extent of  $\text{assxy}$ .” Consequently,  $\text{DestroyLink}(\text{assxy}, (\forall x_1, y_2))$ ,  $\text{DestroyLink}(\text{assxy}, (x_1, \forall y_2))$ , and  $\text{DestroyLink}(\text{assxy}, (\forall x_1, \forall y_2))$  are all equivalent and can be denoted with  $\text{DestroyLink}(\text{assxy}, \forall(x_1, y_2))$ .

If an association end is tagged as unique, then for each object of the opposite class, it defines a sub-collection of the entire collection of coordinates, such that the duplicate values are reduced to a single one. In other words, it is a projection of the entire collection to a set. The elements of a set will be enclosed in curly braces {} in this discussion. For the considered example, if  $\text{py}$  is unique, then:

$$\begin{aligned} \text{py}(x_1)_{t1} &= \{y_1, y_2, y_3\} \\ \text{py}(x_2)_{t1} &= \{y_1, y_3\} \\ \text{py}(x_3)_{t1} &= \{y_2\} \\ \text{py}(x_1)_{t2} &= \{y_1, y_2, y_3\} \\ \text{py}(x_2)_{t2} &= \{y_1, y_3\} \\ \text{py}(x_3)_{t2} &= \{y_2\} \end{aligned}$$

Similarly, if  $\text{px}$  is unique, then:

$$\begin{aligned} \text{px}(y_1)_{t1} &= \{x_1, x_2\} \\ \text{px}(y_2)_{t1} &= \{x_1, x_3\} \\ \text{px}(y_3)_{t1} &= \{x_1, x_2\} \\ \text{px}(y_1)_{t2} &= \{x_1, x_2\} \\ \text{px}(y_2)_{t2} &= \{x_1, x_3\} \\ \text{px}(y_3)_{t2} &= \{x_1, x_2\} \end{aligned}$$

As you can see, uniqueness is not a characteristic of the association, but of its end(s). Uniqueness defines the way the collection of ordered pairs from the association extent is projected on its end for each object

## Part III: Concepts

---

of the opposite class. If the end is non-unique, the projection is one-to-one to the original and is a bag. Otherwise, the projection defines a sub-collection that is a set.

If an association end is unique, the Destroy Link action always removes the given value from the collection of that end totally. For example, if  $\text{py}$  is unique, then  $\text{DestroyLink}(\text{assxy}, (x, y))$  means “remove those and only those pairs  $(x, y)$  from the extent of  $\text{assxy}$ , such that  $\text{py}(x)$  does not contain  $y$  anymore.” As already shown, this simply means “remove all pairs  $(x, y)$  from the extent of  $\text{assxy}$ ,” denoted with  $\text{DestroyLink}(\text{assxy}, \forall(x, y))$ .

It is interesting to note that association ends, as projections, are the only possibility to view the association extent, because there are actions that read the collections defined by the association ends, but not by the association per se. In other words, the association extent can be simply inferred from its projections, not explicitly retrieved. Simply, there is no action that would return the entire extent of  $\text{assxy}$ , but only the action(s) that would return the collections  $\text{py}(x)$  or  $\text{px}(y)$ .

As an interesting implication of such semantics, the extent of an association with both its ends specified as unique actually behaves like a set. Really, since both ends are unique, adding a new pair  $(x, y)$  that already exists in the extent does not affect the collections of any of its ends (that is, does not affect the projections). The Destroy Link action, as described, always has the semantics of deleting all pairs with the given values  $(x, y)$ . Because the collections of the ends (as projections of the association’s extent) are the only available views, exactly the same projections would be obtained by treating the association extent as a set of pairs (without duplicates) instead of as a bag (allowing duplicates) with the described semantics of actions.

For example, if the extent of  $\text{assxy}$  is at a certain moment in time  $t_1$ :

$$\text{assxy}_{t_1} = [(x_1, y_1), (x_1, y_2), (x_1, y_3), (x_2, y_1), (x_2, y_3), (x_3, y_2)]$$

and  $\text{CreateLink}(\text{assxy}, (x_1, y_2))$  is executed, then the association extent becomes the following, if it is treated as a bag:

$$\text{assxy}_{t_2} = [(x_1, y_1), (x_1, y_2), (x_1, y_3), (x_2, y_1), (x_2, y_3), (x_3, y_2), (x_1, y_2)]$$

Or, it remains the same, if it is treated as a set:

$$\text{assxy}_{t_2} = \{(x_1, y_1), (x_1, y_2), (x_1, y_3), (x_2, y_1), (x_2, y_3), (x_3, y_2)\}$$

The  $\text{DestroyLink}(\text{assxy}, (x_1, y_2))$  action has the same effect on both of these and produces the following:

$$\text{assxy}_{t_3} = [(x_1, y_1), (x_1, y_3), (x_2, y_1), (x_2, y_3), (x_3, y_2)]$$

As you can see, if (and only if) both ends of a binary association are unique, then the association’s extent behaves like a set, meaning that it does not exhibit duplicate pairs. It is important to emphasize that this conclusion does not have any implications on the outwardly visible behavior and interpretation of associations and their ends, but is just important for an implementation. An implementation of the OOIS UML profile may rely on this conclusion to make the storage and manipulation with extents of associations with both unique ends (which are most frequent in practice) more efficient than of other associations.

As another example, let's suppose that the association end  $\text{py}$  is unique, while the end  $\text{px}$  is non-unique. Let's say that the extent of  $\text{assxy}$  at a certain time  $t$  is as follows:

```

$$\text{assxy}_t = [(x_1, y_1), (x_1, y_2), (x_1, y_3), (x_2, y_1), (x_2, y_3), (x_2, y_1), (x_3, y_2), (x_1, y_2), (x_1, y_2)]$$

```

Then the association ends define the following projections:

```

$$\begin{aligned} \text{py}(x_1)_t &= \{y_1, y_2, y_3\} \\ \text{py}(x_2)_t &= \{y_1, y_3\} \\ \text{py}(x_3)_t &= \{y_2\} \\ \text{px}(y_1)_t &= [x_1, x_2, x_2] \\ \text{px}(y_2)_t &= [x_1, x_3, x_1, x_1] \\ \text{px}(y_3)_t &= [x_1, x_2] \end{aligned}$$

```

The multiplicity of an association end is an implicit constraint attached to that end, which constrains the cardinality of (that is, the number of elements in) every collection specified by that end. Therefore, the multiplicity is not a constraint on the very association extent, but on the cardinality of its projections only. The constraint is checked and enforced by every action that affects the projection. For example, consider the association  $\text{assxy}$  at the time  $t$ :

```

$$\text{assxy}_t = [(x_1, y_1), (x_1, y_2), (x_1, y_3), (x_2, y_1), (x_2, y_3), (x_2, y_1), (x_3, y_2), (x_1, y_2), (x_1, y_2)]$$

```

If the association end  $\text{py}$  is non-unique, then the cardinalities of collections (denoted here with  $\text{card}$ ) are as follows:

```

$$\begin{aligned} \text{card}(\text{py}(x_1))_t &= 5 \\ \text{card}(\text{py}(x_2))_t &= 3 \\ \text{card}(\text{py}(x_3))_t &= 1 \end{aligned}$$

```

If the association end  $\text{py}$  is unique, then the cardinalities are as follows:

```

$$\begin{aligned} \text{card}(\text{py}(x_1))_t &= 3 \\ \text{card}(\text{py}(x_2))_t &= 2 \\ \text{card}(\text{py}(x_3))_t &= 1 \end{aligned}$$

```

Ordering is yet another characteristic of an association end, and not of an association extent itself. Ordering of the association end  $\text{py}$  means that for each  $x$  of  $\text{x}$ ,  $\text{py}(x)$  is an ordered collection. More formally,  $\text{py}(x)$  is a mapping from a range of successive non-negative integer numbers  $0..n-1$  to a set of (some) objects of  $\text{y}$ , where  $n = \text{card}(\text{py}(x))$ . The integer assigned to a collection element will be referred to as its *index*. From now on, if an association end is ordered, the elements of its collection will be written in that order, and thus the indices are implied from the position of the element in the sequence. The question now is how ordering works with Create Link and Destroy Link actions that affect the entire association extent.

Let's first consider a non-unique ordered association end  $\text{py}$ . For example, let the extent of  $\text{assxy}$  at the time  $t_1$  be as follows:

```

$$\text{assxy}_{t_1} = [(x_1, y_1), (x_1, y_2), (x_1, y_3), (x_2, y_1), (x_2, y_3), (x_2, y_1), (x_3, y_2), (x_1, y_2), (x_1, y_2)]$$

```

## Part III: Concepts

---

Let the non-unique ordered collections of  $\text{py}$  be as follows (in order):

```
py(x1)t1 = [y2, y3, y2, y1, y2]
py(x2)t1 = [y1, y1, y3]
py(x3)t1 = [y2]
```

The non-unique ordered collections of  $\text{px}$  may be, for example, as follows (in order):

```
px(y1)t1 = [x2, x2, x1]
px(y2)t1 = [x3, x1, x1, x1]
px(y3)t1 = [x1, x2]
```

It is clear that ordering is not a characteristic of the association extent by any means. The indices of the two values from a pair are independent, as well as the indices of the values having different opposite coordinates (for example, indices of  $x$ 's for two different  $y$ 's). Therefore, an association extent cannot be treated as ordered, even when both its ends are ordered. It is now the question of how the indices (that is, the ordering) of the elements in the projections are determined. The answer is — Create Link and Destroy Link actions that affect the association extent determine the ordering.

If an association end is ordered, the Create Link action must specify the index that defines the position to which the value will be inserted in the corresponding projection. When the pair is added to the association extent, the indices of the values from the corresponding projections are adjusted accordingly, so that the new value is inserted at the given position.

For the last given example, when both ends  $\text{px}$  and  $\text{py}$  are ordered, a Create Link action must specify indices for both coordinates. For the last given  $\text{assxy}_{t1}$ , after the execution of the action `CreateLink(assxy, (x3[0], y3[1]))`, the association extent will look like this:

```
assxyt2 = [(x1, y1), (x1, y2), (x1, y3), (x2, y1), (x2, y3), (x2, y1), (x3, y2), (x1, y2),
(x1, y2), (x3, y3)]
```

The ordered projections then become the following:

```
py(x1)t2 = [y2, y3, y2, y1, y2]
py(x2)t2 = [y1, y1, y3]
py(x3)t2 = [y2, y3]
px(y1)t2 = [x2, x2, x1]
px(y2)t2 = [x3, x1, x1, x1]
px(y3)t2 = [x3, x1, x2]
```

Therefore, the command `CreateLink(assxy, (x3[0], y3[1]))` means “add the pair  $(x_3, y_3)$  to the association extent of  $\text{assxy}$ ,” and has the effect that  $y_3$  is added to  $\text{py}(x_3)$  at the position 1 (that is, at the end), and  $x_3$  is added to  $\text{px}(y_3)$  at the position 0 (that is, inserted at the beginning), whereas  $x_1$  and  $x_2$  are moved one place forward in  $\text{px}(y_3)$ .

Ordered unique association ends have a bit more complex interpretation. Let  $\text{py}$  be unique and ordered. For example, the association extent  $\text{assxy}$  at a certain moment  $t_1$  may look like this ( $\text{px}$  is still non-unique and ordered):

```
assxyt1 = [(x1, y1), (x1, y2), (x1, y3), (x2, y1), (x2, y3), (x2, y1), (x3, y2), (x1, y2)]
```

The projections may look like this (in order):

```

py(x1)t1 = [y3, y1, y2]
py(x2)t1 = [y1, y3]
py(x3)t1 = [y2]
px(y1)t1 = [x2, x2, x1]
px(y2)t1 = [x3, x1, x1]
px(y3)t1 = [x1, x2]

```

The Create Link action must specify the index of the value that will be inserted in the ordered association end. Additionally, if that value already exists in the unique collection, the value is moved to the given index. For the last example of `assxyH`, after the execution of the action `CreateLink(assxy, (x1[0], y2[0]))`, the association extent becomes the following:

```

assxyt2 = [(x1, y1), (x1, y2), (x1, y3), (x2, y1), (x2, y3), (x2, y1), (x3, y2),
 (x1, y2), (x1, y2)]

```

The projections become the following (in order):

```

py(x1)t2 = [y2, y3, y1]
py(x2)t2 = [y1, y3]
py(x3)t2 = [y2]
px(y1)t2 = [x2, x2, x1]
px(y2)t2 = [x1, x3, x1, x1]
px(y3)t2 = [x1, x2]

```

The Destroy Link action for an association with an ordered unique association end can either specify the index of the value or the value itself — either of these uniquely identifies the value to be removed. Let's keep assuming that `py` is unique and ordered and `px` is non-unique and ordered. Then it follows:

- `DestroyLink(assxy, (x[i], y))` can be interpreted as “remove  $x[i]$  from  $px(y)$  and  $y$  from  $py(x)$ .” Therefore, it removes from the association extent all the pairs  $(x, y)$ , and adjusts the rest of the indices in  $py(x)$  and  $px(y)$  accordingly. Note that `DestroyLink(assxy, (?[i], y))` has the same effect as `DestroyLink(assxy, (x[i], y))`, because  $x$  is uniquely identified as the value at the position  $i$  in  $px(y)$ .
- `DestroyLink(assxy, (x[i], ?[j]))` has the same effect as `DestroyLink(assxy, (x[i], y))` because  $y$  can be uniquely identified as the value at the position  $j$  in  $py(x)$ . It can thus be interpreted as “remove the value from the position  $j$  from  $px(y)$ .”

### Section Summary

- The *extent* of a binary association is an unordered collection (that is, a bag) of links, whereby a link is an ordered pair of values that refer to objects.
- An association end `py` at the class `Y` defines a mapping `py(x)` for each object `x` of the class `X` at the opposite end, which is a collection of all those (and only those) values `y` from the pairs  $(x, y)$  that exist in the association extent.

*Continued*

- ❑ If an association end  $\text{py}$  at the class  $\text{Y}$  is unique, then for each  $x$  of  $\text{X}$ ,  $\text{py}(x)$  is the unique collection, projected from the collection of all those (and only those) values  $y$  from the pairs  $(x, y)$  that exist in the association extent so that duplicates are removed (that is, projection to a set).
- ❑ If an association end  $\text{py}$  at the class  $\text{Y}$  is ordered, then for each  $x$  of  $\text{X}$ ,  $\text{py}(x)$  is an ordered collection of those (and only those) values  $y$  from the pairs  $(x, y)$ .
- ❑ If both association ends are unique, then the association extent behaves like a set.
- ❑ Multiplicity of an association end  $\text{py}$  is a constraint on the cardinality of the collection  $\text{py}(x)$  for every object  $x$  of the class  $\text{X}$  at the opposite end.

## **Special Characteristics of Association Ends**

This section examines special characteristics of association ends.

### **Navigability**

The Read Links action is specified for the given association end  $\text{py}$  (see Figure 10-5) and returns  $\text{py}(x)$  — the collection of objects of  $\text{Y}$  connected with the given object  $x$  of  $\text{X}$  at the (slot of the) association end  $\text{py}$ . It is said that this action *navigates* from the object  $x$  over the (slot of the) association end  $\text{py}$ . By default, such navigation is allowed, and the association end is said to be *navigable*. Navigability of an association end means that objects participating in links at that end can be accessed efficiently from instances participating in links at the other end of the association.

Every association end that belongs to the class at the opposite end and, thus, represents a property of that class is always navigable.

On the other hand, an association end that belongs to the association is navigable by default, but can also be declared as non-navigable. In that case, the Read Links action is not allowed for that association end.<sup>6</sup> If the association end for that action is specified statically, the model is incorrect and cannot be compiled. If the association end for that action is specified dynamically, over the reflection mechanism, the action will raise an exception.

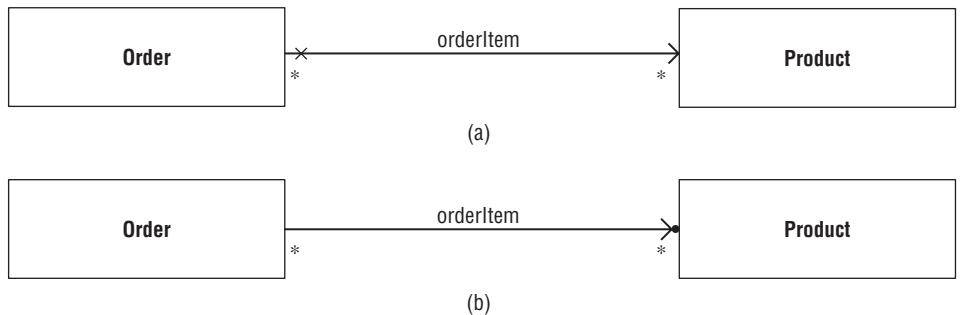
The modeling tool must declare an association end owned by its association as navigable or not. In the UML visual notation, an open arrowhead on the end of an association indicates that the end is navigable, and a small  $\times$  on the end of an association indicates that the end is not navigable (see Figure 10-6a). These symbols can be all shown, when navigability or its absence are made completely explicit in diagrams (see Figure 10-6a).

Alternatively, arrows can be suppressed for associations with navigability in both directions and shown on navigable ends for associations navigable in only one direction (see Figure 10-6b), while the instances of  $x$  are always suppressed. In this case, a bi-directionally navigable association cannot be distinguished

---

<sup>6</sup>In standard UML, a non-navigable association end means that the access to the linked objects is either impossible or inefficient. The precise mechanism by which such access is achieved is implementation-specific. In OOIS UML, the interpretation is restricted to impossible access.

from an association that is not navigable in any direction. However, the latter case is practically useless, although semantically allowable.



**Figure 10-6: Optional notations for navigable and non-navigable association ends.** An open arrowhead indicates that the end is navigable and a small  $\times$  indicates that the end is not navigable. One can navigate from an object of `Order` to the collection of linked products, but not in the opposite direction. (a) The style when all arrows and instances of  $\times$  are shown and navigability or its absence are made explicit. (b) The style when arrows are shown for navigable ends and instances of  $\times$  are suppressed. When arrows do not exist, the association is either navigable in both directions or not navigable in any (very uncommon in practice).

Finally, navigability symbols can be suppressed at all places, when it cannot be concluded from the diagram whether an end is navigable or not. The approach that is used is a matter of style because all are allowed in the UML notation, but only one style should be consistently used throughout one single model.

The second style mentioned here and used in Figure 10-6b will be used throughout this book. Therefore, if a binary association has an open arrowhead at one end, it is evident from the diagram that its opposite end is not navigable, and, thus, belongs to the association. Consequently, it will not result in a property of the opposite class.

For the example in Figure 10-6b, the class `Product` will not have a property for the association end at the class `Order` because that end is not navigable and belongs to the association. The class `Property` will thus be independent of the class `Order`. The navigable end at the class `Product` belongs to the class `Order`, and is, thus, its property.

In general, an association end is denoted as non-navigable when the navigability is simply not needed or should be prohibited.

The former situation occurs often in conjunction with derived association ends that will be described in a later section. In such cases, an implementation can benefit from the information from the model that the navigation in a certain direction is not needed, so it can optimize the internal representation of the structure for such limited access. Of course, this is a matter of optimization and does not affect the semantics.

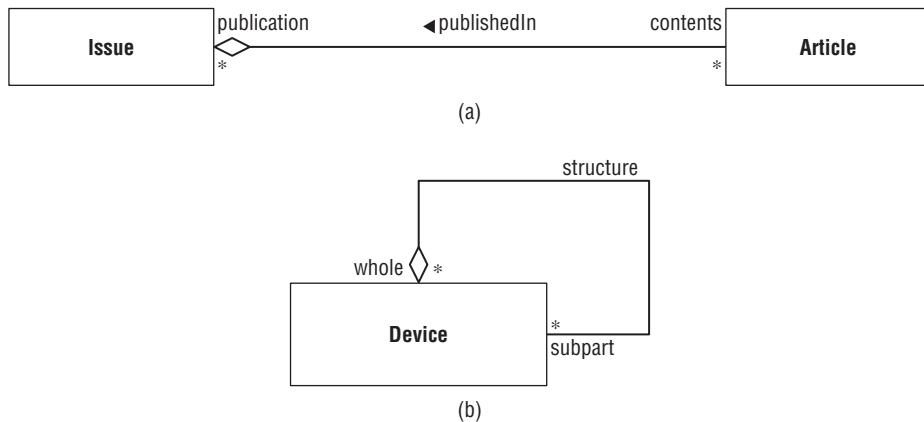
The latter case may occur if the navigation should be prohibited because of specific requirements from the problem domain or because of implementation reasons, although this does not happen too often. For example, the navigability in one direction might be extremely inefficient or difficult to implement, so it must be prohibited.

### Section Summary

- ❑ By default, each association end is *navigable*, which means that it is possible to *navigate* over the slot of that association end from the given object of the class at the opposite end — to retrieve the collection of objects linked to the given object over the slot of that association end.
- ❑ The Read Links action is allowed only for navigable association ends.
- ❑ An association end that belongs to the opposite class is always navigable.
- ❑ An association end that belongs to the association is navigable by default, but can also be declared as non-navigable. In that case, it is not possible to navigate over that end — the Read Links action for that association end is not allowed.

### Aggregations and Compositions

Some binary associations conceptualize the relationships between abstractions in which one of the abstractions represents a *whole*, while the other represents a *part* of the whole. For example, in Figure 10-7a, an Article is a part of the Issue in which it is published. Similarly, the model in Figure 10-7b shows that Devices can be structured into hierarchies, in which one Device, as a whole, can be decomposed into other Devices as its subparts. Such associations are called *aggregations* in UML.



**Figure 10-7: Examples of aggregation.** (a) An Article is a part of an Issue in which it is published. An Article can be re-published in several Issues. (b) A Device can consist of other Devices as its subparts. A Device can be a subpart of many Devices.

In UML, one end of a binary association can be declared as *aggregate* to represent the whole end of an aggregation. Graphically, such an end is adorned with a hollow diamond (see Figure 10-7), noticeably smaller than the diamond that (optionally) represents the association itself.

Aggregation is just a special case of a binary association in UML. An aggregation is a binary association with one (and only one) end declared as aggregate. An aggregation adds no semantics to the basic

semantics of associations — there are no specific run-time implications of aggregations. In particular, there is no propagated deletion or any other kind of existence dependency between the whole and the part, or any constraint in the ownership over the part. As shown in Figure 10-7, the multiplicity of the whole end of an aggregation can be arbitrary. For example, an Article can be re-published in an arbitrary number of Issues, or a Device can be a subpart of many other Devices. An aggregation is, thus, a purely conceptual thing that indicates the whole-part relationship between the associated classes to the reader of the model, but has no specific semantic implications.

A special kind of aggregation is a *composition*. It is an aggregation with the aggregate end declared as *composite*. In diagrams, a composite end is adorned with a filled diamond (see Figure 10-8). A composition is a strong aggregation that has not only the conceptual significance of a whole-part relationship, but also has two very precise semantic implications in addition to the basic semantics of associations:

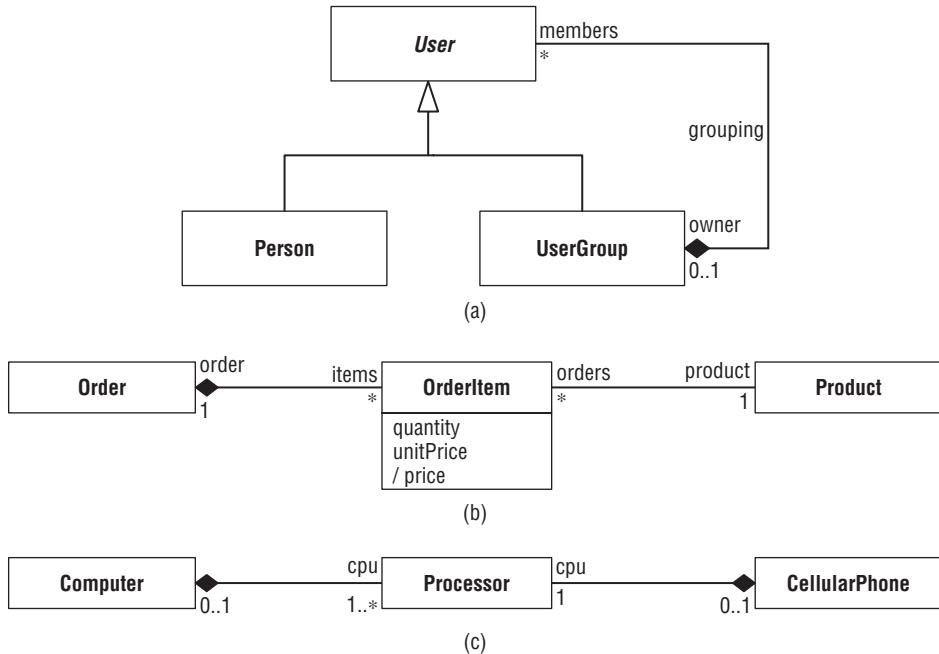
- ❑ There is implicit propagated destruction from the whole to the part. In other words, when an object of the class at the whole end is destroyed, the destruction is propagated to all linked objects of the class at the part end.
- ❑ An object of the class at the part end cannot be part of more than one whole of any composition in which that class plays the role of part at the same time. Consequently, the upper multiplicity bound of a composite end is always 1. The lower multiplicity bound at the composite end can be either 0 or 1.

The first of these rules defines that a composite end has implicitly the propagated destruction characteristic. The second rule is an implicit constraint attached to all composite ends of compositions in which a certain class plays the role of part. Consequently, compositions have the semantics derived from other constructs of OOIS UML.

Figure 10-8 shows several examples of compositions. Figure 10-8a depicts the conceptual model of the user management module of an information system. The model recognizes the concept of a *User*, as an abstract generalization of a concrete *Person* that uses the system, and a *User Group* that may have an arbitrary number of Users (either Persons or other sub-groups) as its members. The composite end *owner* of the association grouping indicates that member Users are “parts” of their owner User Group, and that when a User Group is destroyed, all its members are destroyed, too. Of course, this destruction is further propagated to the sub-groups of this group, recursively.

Additionally, a User cannot be part of more than one User Group at a time. The multiplicity constraint 0..1 at the *owner* end indicates the same. The lower multiplicity bound of *owner* end is set to 0 because the topmost User in the hierarchy has no owners. However, the composite end *owner* introduces a constraint that is stronger than the one specified by its multiplicity. A User that is a member of a User Group cannot be linked with any other object that plays the role of a whole in any other composite association that may exist in the same model.

Figure 10-8b shows a fragment of the model for an order processing application of a trading company. The three key concepts — *Order*, *Product*, and *Order Item* — are related with two associations. *Order Item* represents a line in a placed *Order* (that is, it is a part of its *Order*), and defines the ordered *Product*, its quantity, unit price, and the derived total price. The composite end *order* indicates that an *Order Item* is destroyed when an *Order* is removed from the system, but it does not affect the corresponding *Product*.



**Figure 10-8: Examples of composition.** (a) A User Group is a composition of Users, which can be Persons or other User Groups (as sub-groups). (b) An Order is a composition of Order Items, each of which defines the quantity and price of one ordered Product. (c) A Processor can be embedded as a CPU either in a Computer or in a Cellular Phone, but not in both at the same time.

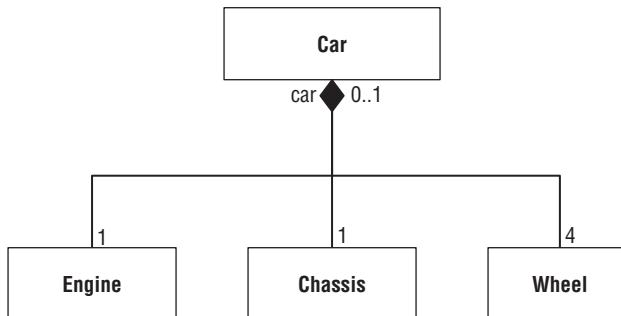
Additionally, its multiplicity constraint 1 of the same end implies that an Order Item cannot exist without being linked to its owner Order. Because the semantics of compositions demands that an Order Item cannot be a part of more than one composite at a time, it follows that an Order Item is always a part of exactly one Order at any time. The other association is not a composition and has no propagated destruction, but its multiplicity constraint 1 at the product end implies that an Order Item must be linked to a Product during its creation. Consequently, an Order Item must be linked to both an Order and a Product upon its creation. However, because the destruction of a Product is not propagated to the linked Order Item, and because the multiplicity constraint at the product end does not allow zero cardinality, a Product cannot be deleted from the system if it has Orders that refer to that Product (over Order Items).

Note that the intentions of the modeler in the last two examples could have been met even without using compositions, but simply by using propagated destructions in concert with multiplicity constraints at the same ends. Figure 10-8c shows an example in which the composite ends carry more information than simple propagated deletion with multiplicity constraints at the same end. This simple model indicates that a Processor can be a part of either a Computer or of a Cellular Phone, but not of both at the same time. Because a Processor cannot be a part of both, the multiplicity constraints at the ends at the classes Computer and Cellular Phone must allow zero cardinality. However, if only propagated destruction and multiplicity constraints had been applied to those ends, the model would not have forbidden the same Processor to be linked to both a Computer and a Cellular Phone at the same time. The composite ends forbid that to happen.

A composition does not imply any policy on when and how parts are created within a whole. If, however, the multiplicity constraint at the part end of a composition has the lower multiplicity bound greater than

0 (as in Figure 10-8c), the constructor(s) of the class must ensure the proper creation of the parts. Additionally, the composition does not imply when exactly the parts are destroyed. They can be destroyed before the whole is destroyed by explicit actions. What is guaranteed is that the parts cannot survive the destruction of their whole because they are implicitly destroyed when the whole is destroyed. Finally, if the multiplicity constraints allow that, an object of a class that plays the role of a part in compositions can exist independently of their whole. For example, the model in Figure 10-8c does not prevent a Processor from existing independently of any Computer or Cellular Phone. If this is not wanted, it should be prevented by explicit constraints in the model.

If several aggregations or compositions have the same class playing the role of the whole, and if all other characteristics of the aggregate/composite ends of these aggregations/compositions are the same, the associations may be drawn as a tree by merging the aggregate/composite ends into a single segment, as shown in Figure 10-9. Any adornments on that single segment apply to all of the aggregate/composite ends.



**Figure 10-9: Presentational option for aggregations or compositions having a common aggregate/composite end**

### Section Summary

- ❑ An *aggregation* is a special kind of a binary association in which one (and only one) end is declared as *aggregate*.
- ❑ Aggregation has a pure conceptual significance to the reader of the model, and has no additional run-time semantics. It conceptualizes the whole-part relationship between the associated classes.
- ❑ A *composition* is a kind of aggregation with the aggregate end declared as *composite*. It has the following derived semantics:
  - ❑ There is implicit propagated destruction from the whole to the part end.
  - ❑ An object of the class at the part end cannot be part of more than one whole of any composition in which that class plays the role of a part. The upper multiplicity bound of a composite end is always 1.

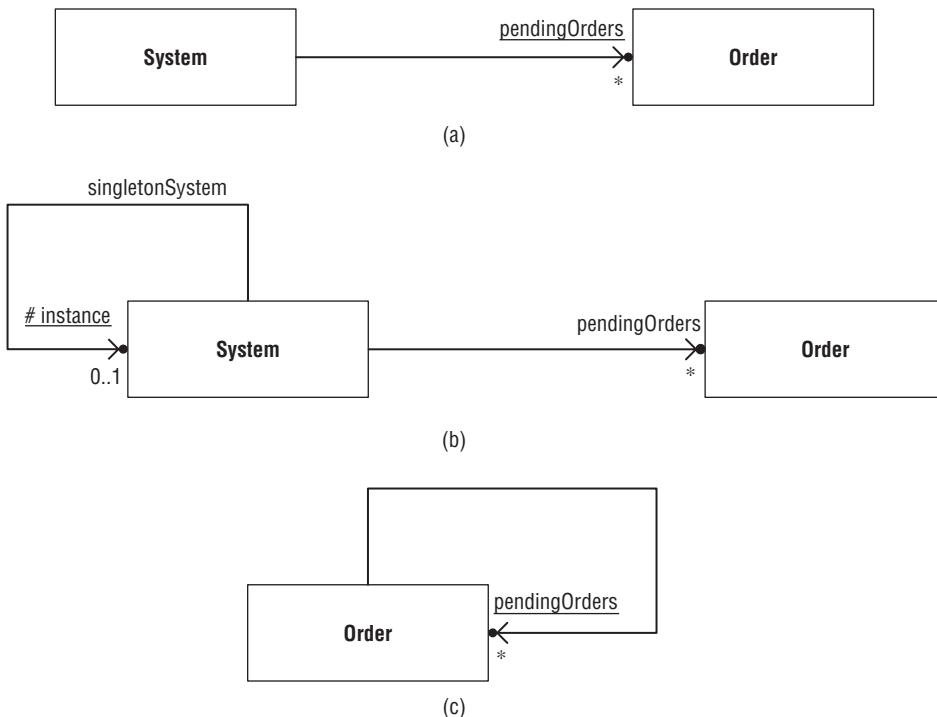
## Part III: Concepts

### Static Association Ends

As with static attributes, sometimes you may want to have an association end that results in a slot that belongs to the class itself, not to every instance of that class. Such an association end is declared as *static*. In effect, the motivation for introducing a static association end may be the same as for introducing a static attribute, except that in OOIS UML, static attributes (as all other attributes) may be of data types only. In the cases where you need a reference to an object or a collection of objects of one class that is shared among all objects of another class, you may use a static association.

In effect, the slot that is the manifestation of a static association end is shared among all objects of the class that owns that association end, and is not proprietary to any individual object. At run-time, it results in the same collection of linked objects of the class at that association end for every object of the class that owns that end.

For example, let's consider an order processing system in which a list of pending Orders must be kept. When a new Order is placed, it should be added to the list of pending Orders. When an Order is fulfilled, it is removed from the list. Figure 10-10 shows three different ways to support this requirement.



**Figure 10-10: Examples of static association ends. Three different ways to model the same requirement that the system should keep a list of pending orders: (a) the list of pending orders is a static property of the class System; (b) every instance of the class System has its own list of pending orders, but there is (at most) one instance of the class System that represents the entire system (Singleton design pattern); (c) the list of pending orders is a static property of the very class Order.**

In the model in Figure 10-10a, the class `System` models the entire order processing system, while its static property `pendingOrders` represents the list of pending Orders. It is a static association end toward the class `Order`. When a customer places a new Order, the Order should be added to the list by the following code:

```
System::pendingOrders->add(anOrder);
```

A processed Order is removed from the list by the following:

```
System::pendingOrders->remove(anOrder);
```

In the model in Figure 10-10b, every instance of the class `System` has its own non-static property `pendingOrders` that represents the list of pending Orders of that system. However, the unique instance of the system is represented by a protected static property `instance` of the class `System`. In that way, the responsibility to create and keep the single instance of the class is given to the class as suggested by the *Singleton* design pattern [Gamma, 1995]. Now, a new Order is added to the list by the following code:

```
System::instance.pendingOrders->add(anOrder);
```

A processed Order is removed from the list by the following code:

```
System::instance.pendingOrders->remove(anOrder);
```

Finally, in the model in Figure 10-10c, bookkeeping of the list of pending Orders is the responsibility of the class `Order`. The list is modeled by the static property `pendingOrders` of that class. When a customer places a new Order, it should be added to the list by the following code:

```
Order::pendingOrders->add(anOrder);
```

A processed Order is removed from the list by the following:

```
Order::pendingOrders->remove(anOrder);
```

In general, a static association end `py` at the class `Y` maps the class `X` at the opposite end, but not every object of that class, to a collection of objects of `Y`. In other words, `py(x)` is the same collection for every object `x` of `X`. Consequently, an association having a static end has significantly different run-time semantics than an association without static ends because the extent of the former does not represent a collection of ordered pairs  $(x, y)$ . Instead, a static association end has the run-time semantics more similar to those of a static attribute or of a variable, with the difference in OOIS UML that the type of the association end is always a class, never a data type.

Because of this interpretation, static association ends require many constraints in modeling.<sup>7</sup> First, a binary association may have at most one static end. Obviously, an association with both static ends would make no sense (it would carry no more information than a single Boolean). Second, a static association end always belongs to the opposite class, never to the association, because it is intended to represent a property of the class. A static association end is, thus, always navigable, while the opposite end is never

---

<sup>7</sup>These constraints on the model are specific for the OOIS UML profile. They do not exist in standard UML.

## Part III: Concepts

---

navigable, because it does not represent any mapping to collections of objects. Consequently, the name, multiplicity, ordering, and uniqueness of the end opposite to a static association end have no semantic significance, although they may be specified in the model. Finally, any end of an association with a static end cannot have propagated destruction, and cannot be aggregate or composite.

As with a static attribute, a static association end can be accessed as follows:

- ❑ By specifying the object and the property, `p.prop`, but where the reference to the object is actually irrelevant and does not need to be computed at run-time because the result is the same for each object. Instead, it is used to determine, at compile time, the class to which the static property belongs.
- ❑ Without specifying the object, but just referring to the property as a named feature of the class. For example, the attribute `pendingOrders` can be referred to by its qualified name (for example, `Order::pendingOrders`) or even by its unqualified name (`pendingOrders`) within the namespace of its class.

The collection to which a static association end maps is inherited in specializing classes. Static association ends cannot be initialized, but represent empty collections initially. It is the responsibility of the application to provide explicit actions that maintain the values of static association ends. Consequently, the lower multiplicity bound of a static association end must be zero.<sup>8</sup>

All actions that work with a binary association having one static end affect only the collection defined by that end. Essentially, their effect is exactly the same as the effect of the corresponding actions that work on variables of the same type as the static association end. In that way, static association ends work just like (persistent and global) variables.

Static properties are underlined in the UML notation, as shown in Figure 10-10. Optionally, only the name of the property can be underlined.

Static association ends in UML have the same purpose and are generally used in the same circumstances as static attributes. The only difference is that their type and their owner are classes, not data types. Therefore, the motivations for using static association ends are similar to those discussed for static attributes — when some information must be shared among all instances of the same class, or it belongs to the very class, not to each particular instance.

### Section Summary

- ❑ A *static association end* is a (static) property of a class, whose value is shared among all objects of that class. A static association end defines a collection of objects of the class at that end.
- ❑ Static properties can be accessed over references to objects, or without references to objects.

---

<sup>8</sup>These semantic rules are specific for the OOIS UML profile. They do not exist in standard UML.

- ❑ A binary association can have at most one static end. A static association end belongs to the class at the opposite end. The association end opposite to a static end is never navigable. Any end of an association with a static end cannot have propagated destruction and cannot be aggregate or composite.
- ❑ The values of static properties are inherited in specialized classes.
- ❑ Static association ends cannot be implicitly initialized.
- ❑ Static properties are underlined in the UML notation.
- ❑ Static properties are used when some information needs to be shared among all objects of the same class, or it belongs to the class, not to each particular object.

### **Read-Only Association Ends**

Sometimes, an object should be linked to other objects at its creation, and these links should not be modified after the object's initialization has completed. For example, in an order processing system, a Shipment of products should be assigned to its originating Order at its creation, and should not be reassigned to another Order afterward. For that purpose, similar to attributes, a navigable association end can be specified as *read-only*, as shown in Figure 10-11a.

A read-only non-static association end specifies that a link cannot be created or deleted for an object of the class at the opposite end after that object has been initialized. In other words, objects participating in the association across from a non-read-only end can have links created or deleted as long as the object across from the read-only end is still being initialized. For the mentioned example in Figure 10-11a, an Order can be linked to or unlinked from a Shipment only while the Shipment is being initialized.

The read-only specification for an association end restricts the applicability of all actions that attempt to modify the mappings designated by that end. For the example in Figure 10-11a, the read-only association end `order` designates a mapping from any object of the class `Shipment` to the collection of objects of the class `Order`. If an action would modify that mapping for an object of the class `Shipment` whose initialization is over, its execution will raise an exception of type `WritingReadOnlyAssociationEndException`.

The meaning of "initialization" of an object is as described for read-only attributes. That is, an object is being initialized as long as the Create Object action is being executed, including the execution of its constructor and nested operations invocations.<sup>9</sup> Therefore, the applicability of an action that wants to modify a read-only association end is checked exclusively at run-time. As already described for read-only attributes, an operation other than constructor can issue an action that modifies a read-only association end. During the activity of this operation within the activity of the constructor, such an action is allowed. But if the same operation is invoked after the object has completed its initialization, the same action would raise an exception.

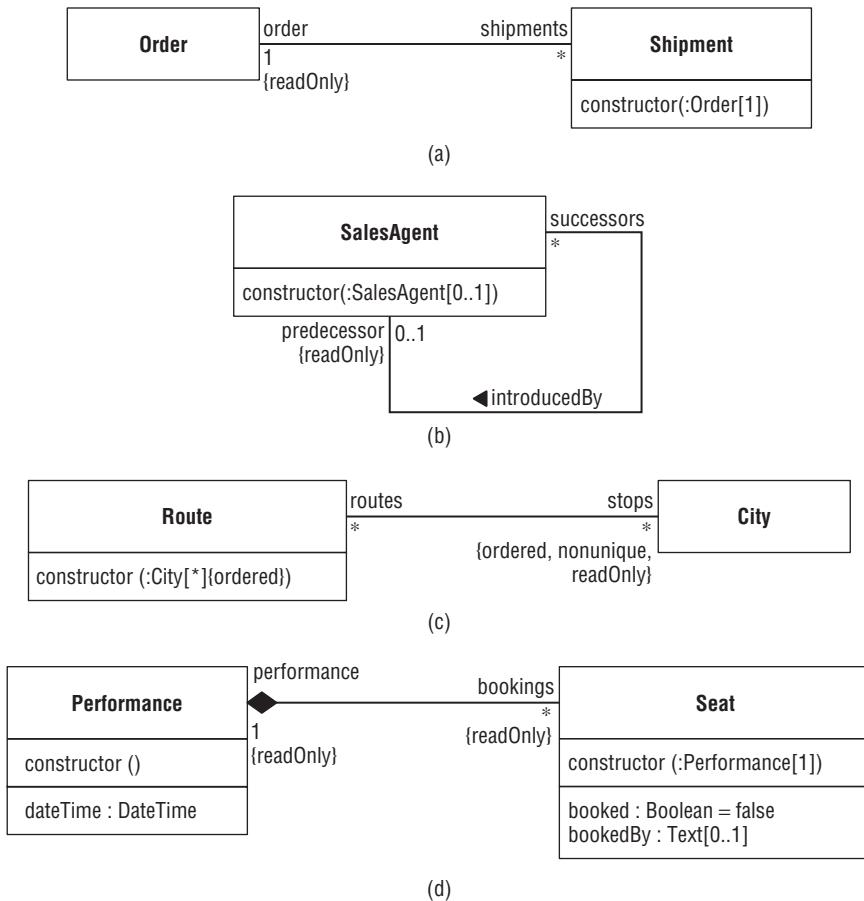
---

<sup>9</sup>This is a semantic specialization of the OOS UML profile. In standard UML, the semantics of "initialization" are not so strictly defined.

## Part III: Concepts

---

Of course, for the example shown in Figure 10-11a, the multiplicity constraint 1 at the end `order` would prevent the creation of new links or deletions of existing links for a Shipment in many (but not all) cases. However, of course, there are circumstances in which the multiplicity constraint cannot do that at all. To illustrate this, let's consider another example shown in Figure 10-11b. The figure shows a fragment of the model for a multilevel marketing system, whereby a Sales Agent can introduce other Sales Agents into the multilevel system, in a “pyramidal” manner. The relationship between agents should be stored in the system and is modeled by the `introducedBy` association.



**Figure 10-11: Examples of read-only association ends.** (a) In an order processing system, a Shipment must be assigned to an Order at its creation and cannot be reassigned to another Order afterward. (b) In a multilevel marketing system, a Sales Agent can be introduced to the system by another Sales Agent (its predecessor) and cannot be reassigned to another predecessor afterward. (c) A sightseeing Route that visits a series of Cities is defined at its creation and cannot be modified afterward. (d) In a booking system of a theater, for each Performance, a collection of Seats is created by the Performance’s constructor. These two objects are linked during their creation and cannot be unlinked afterwards.

Once created as a “successor” of the Sales Agent that introduced him or her to the system, a new Sales Agent cannot be reassigned to another senior agent. This is why the association end predecessor is restricted as read-only. Of course, the topmost Sales Agents that have been initially introduced to the system have no predecessors. This is why the multiplicity at that end allows zero cardinality. A correct implementation of the creation of a Sales Agent as a successor of an existing Sales Agent is through the constructor `constructor(pred:SalesAgent[0..1])` of the class `SalesAgent` that accepts a reference to the predecessor and creates the link with that object:

```
this.predecessor->set(pred);
```

On the other hand, a completely independent method cannot create a new object of `SalesAgent` and then link it to its predecessor, because at that time, the object has already been initialized:

```
SalesAgent anAgent = new SalesAgent;
anAgent.predecessor->set(aSeniorAgent); // Exception!
```

Note that the multiplicity constraint 0..1 at the end predecessor does not cause an exception in this case, but the read-only restriction does.

The same holds for this code:

```
aSeniorAgent.successors->add(new SalesAgent); // Exception!
```

An example with completely unrestricted multiplicities at both association ends is shown in Figure 10-11c. Here, a sightseeing Route is defined by a sequence of Cities it visits. This sequence is ordered and non-unique because a Route can revisit the same City several times, and is of arbitrary size. However, the intention is that the sequence is not modified once a Route is defined (initialized). Therefore, the constructor `constructor(cities:City[*]{ordered})` of `Route` accepts the sequence of Cities and creates the links with them:

```
cities->forEach(c) this.stops->add(c);
```

Of course, the opposite ends in the listed examples are not read-only, which means that objects of the class `Order`, the predecessor Sales Agent, or objects of the class `City` do not have to be in the process of initialization in order to be linked. They can be linked to or unlinked from the opposite objects at any time, provided that other constraints (such as multiplicities) are satisfied.

Consequently, if both ends are read-only, links can be created or deleted only if all participating objects are being initialized. The example in Figure 10-11d illustrates such a case. In a booking system for a theater, for each Performance (scheduled on a certain date and at a certain time), a collection of objects of `Seat` (one for each seat in the theater) is created by the Performance’s constructor. These two objects are linked during their creation and cannot be unlinked afterward. Note that the links between a Performance and its bookings must be created within the constructor of the class `Seat`, as follows:

```
// For all seats in the theater, do:
new Seat(this);
```

The method for `Seat::constructor(p:Performance[1])` then looks like this:

```
this.performance->add(p);
```

## Part III: Concepts

---

Note that the creation of the link between two objects cannot be postponed for the constructor of `Performance` because, at that time, the initialization of a `Seat` has already completed. If the method of the constructor of `Performance` looked like the following, it would raise an exception:

```
// For all seats in the theater, do:
bookings->add(new Seat); // Exception!
```

The read-only restriction does not affect implicit deletion of links when an object at the end across from the read-only end is being destroyed. For example, when a `Performance` is being destroyed, with that destruction being propagated to all its bookings, the read-only restrictions do not prevent the destruction of these objects and the links between them.

Only non-static navigable association ends can be read-only in OOIS UML.

A read-only association end is shown using the `readOnly` modifier as part of the string within curly braces drawn near the end. This annotation may be suppressed, in which case it is not possible to determine from the diagram whether the association end is read-only or modifiable. However, a modeling tool may allow suppression of this annotation only when the association end is not read-only. In this case, it is possible to assume that the association end is modifiable in all cases where the modifier `readOnly` is not shown. The latter approach is used throughout this book.

### Section Summary

- ❑ A read-only non-static association end specifies that a link cannot be created or deleted for an object of the class at the opposite end after that object has been initialized. The read-only specifier for a non-static association end restricts the applicability of all actions that attempt to modify the mappings designated by that end. This rule is checked at run-time and raises an exception if the checking fails.
- ❑ The read-only restriction does not affect implicit deletion of links when an object at the end across from the read-only end is being destroyed.
- ❑ Only non-static navigable association ends can be read-only.
- ❑ A read-only association end is shown using the `readOnly` modifier as part of the string within curly braces.

## Frozen Association Ends<sup>10</sup>

Similar to attributes, an object's slot that is a manifestation of an association end can be *frozen* by an explicit action:

```
anObject.aProperty->freeze()
```

---

<sup>10</sup>This feature is specific for the OOIS UML profile.

After that action, any other action that attempts to modify the mapping designated by a frozen slot of that object would raise an exception of type `WritingReadOnlyAssociationEndException`. This checking is always performed dynamically.

An object's slot that is a manifestation of an association end can be unfrozen by the following:

```
anObject.aProperty->unfreeze()
```

After that, the slot can be modified again.

Read-only association ends are actually those for which an implicit freeze action is performed immediately after the initialization of the owner object. The unfreeze action on read-only association ends is an error reported at either compile time or run-time.

### Section Summary

- An object's slot that is a manifestation of an association end can be frozen by an explicit action.
- The mapping designated by a frozen property cannot be modified by actions. Any action that tries to modify a frozen slot will raise an exception.
- An object's slot that is a manifestation of an association end can be unfrozen.

## Derived Associations and Association Ends

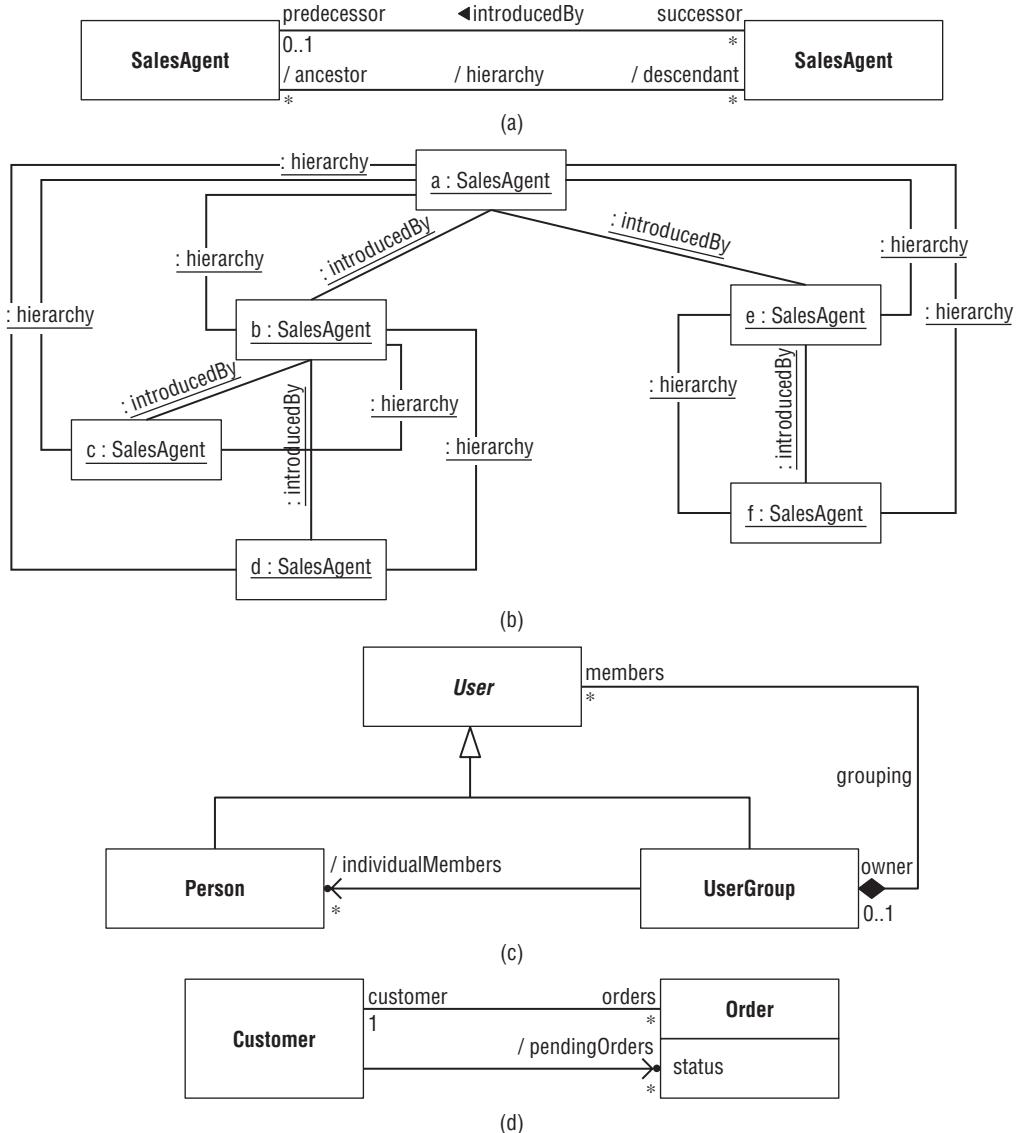
For the sake of efficiency and convenience of access, it is sometimes necessary to have the information that has a form of a collection of objects related to another object explicitly stored in an association, although it can be derived from other sources of information.

Let's revisit the example of a multilevel marketing system, whose partial model is shown in Figure 10-12a. In that system, a Sales Agent can introduce other Sales Agents into the multilevel system, in a "pyramidal" manner. The relationship between agents should be stored in the system and is modeled by the `introducedBy` association.

This relationship is used to calculate the agent's commission. The commission is taken from all the agent's accomplished sales, but also from the accomplished sales of all its direct or indirect successors. An algorithm that would calculate the commission for a Sales Agent by traversing the entire sub-tree, rooted at that agent, over the links of the association `introducedBy`, may require visiting a large number of objects and may pose a heavy retrieval workload to the persistent storage server, because the objects would be fetched one by one in a recursive manner.

To overcome this problem, a good idea is to introduce another association whose links would store the information about all direct and indirect ancestor-descendant relationships. This association is named `hierarchy`, and is shown in Figure 10-12a. Figure 10-12b shows a diagram for a sample object structure according to this model. A Sales Agent is linked with its direct successors by the links of the association `introducedBy`, but also to all its direct and indirect descendants by the links of the association `hierarchy`. Obviously, the links of the latter can be derived from the links of the former by a transitive closure.

## Part III: Concepts



**Figure 10-12: Examples of derived associations and association ends.** (a) In a multilevel marketing system, a Sales Agent can be introduced to the system by another Sales Agent (its predecessor). For the purpose of efficient calculation of commissions for all direct and indirect ancestors of an agent, the derived association `hierarchy` is introduced. (b) An object diagram showing a sample object structure for the model in (a), whereby the links of the derived association `hierarchy` connect Sales Agents with all their direct and indirect ancestors. (c) A User Group has the property `members`, which results to all of its individual members and sub-groups, and a derived property `individualMembers` that provides a direct access to its individual members only. (d) The collection of pending Orders placed by a certain Customer can be derived from the collection of all Orders of that Customer and their statuses.

On the one hand, the introduction of the derived association hierarchy makes the algorithm of calculating the agent's commission straightforward and efficient, because it should visit only the objects directly linked over the property `descendant`. On the other hand, the derived association introduces additional maintenance overhead: the links of this association must be updated on every creation or deletion of a link of the association `introducedBy`.

The trade-off is resolved by estimating the frequency and need for the two operations — calculating an agent's commission and modifying the links of `introducedBy` and `hierarchy` associations. The former is expected to be frequent. After all, this is one of the main tasks of the system. The latter is done infrequently — generally, only when a new Sales Agent is introduced to the system.

As described in the earlier section on read-only association ends, the end `predecessor` can be designated as read-only, so that an agent cannot be reassigned to another predecessor after his or her introduction to the system. Additionally, the first operation is performed regularly and for all agents in the system. The second is done occasionally, just for one new agent.

Another example is shown in Figure 10-12c. A User Group has the property `members`, which results in all of its individual members and sub-groups, and a derived property `individualMembers` that provides a direct access to its individual members only. The unidirectional association with a derived navigable association end `individualMembers` is introduced to provide an efficient and convenient access to the subset of members of a User Group who are Persons. The association is unidirectional because the navigability in the opposite direction is simply not needed as it would be completely redundant — a Person can reach the User Group of which he or she is a member through the inherited property `owner`.

The association end `individualMembers` designates a collection of objects that can be derived from the association end `members` by filtering those that are of type `Person`. The design decision to introduce this derived association may be again motivated by convenience and/or efficiency of retrieval. Of course, the frequency and need for a direct access to the individual members of a User Group must justify the overhead of maintaining the consistency between the source and the derived information.

A final example is shown in Figure 10-12d. Here, the collection of pending Orders placed by a certain Customer can be derived from the collection of all Orders placed by that Customer and their status. The motivation is again the need and frequency of direct retrieval of pending Orders of a Customer, which can be reached simply by referring to the property `pendingOrders` of an object of the class `Customer`.

In short, a *derived association* is an association whose links can be computed from other sources of information. Similarly, a *derived association end* designates the mappings that can be computed from other sources of information. In the UML notation, the derived associations and association ends are displayed with the slash sign in front of their names, as shown in Figure 10-12.

As with derived attributes, derived associations and association ends have no special impact on the run-time semantics of the model. When an element is annotated as derived, it has just an informative meaning to the reader or implementer of the model, and has no direct effect on the structure or behavior of the system. It simply indicates to the reader that the run-time content of the element is in some way computed from other sources of information. It does not indicate *how* and *when* it is computed.

Instead, it is up to the modeler to specify how this content is related to other sources of information and to which sources it is related, in terms of other modeling concepts. Therefore, the semantics of associations and association ends also hold for derived associations and association ends. Most notably, all actions that create, delete, or read links work for derived associations and association ends as usual. Consequently, the notion of a derived element has no additional formal semantics in UML.

## Part III: Concepts

---

In practice, the approaches to implement the computation and maintenance of derived elements are the same as described for derived attributes. In short, only the approach that does not assume any support from the environment and leaves everything to the modeler is assumed in the current version of OOS UML. Standard UML allows all approaches, but does not specify the precise semantics of any.

In general, derived associations and association ends are used when some information that can be derived from other information should be retrieved, is retrieved relatively often, but its computation is relatively costly. Of course, there is always a trade-off, because the maintenance of the derived elements requires some additional modeling efforts and run-time overhead, as described earlier. Therefore, the modeler has to decide whether to introduce a derived element at all, and if he/she does that, to select the proper implementation approach.

### Section Summary

- ❑ A *derived association* is an association whose links can be computed from other sources of information.
- ❑ A *derived association end* designates the mappings that can be computed from other sources of information.
- ❑ A derived element is an informative annotation that tells the reader of the model that the run-time content of the element can be computed from other sources of information, but not how and when it is computed. Read and write actions work with derived associations and association ends as usual.
- ❑ Introduction and implementation of a derived element is most often a trade-off. The decision is influenced by the frequency and cost of modifications of the source information, as well as the frequency and cost of retrieval of the derived information.

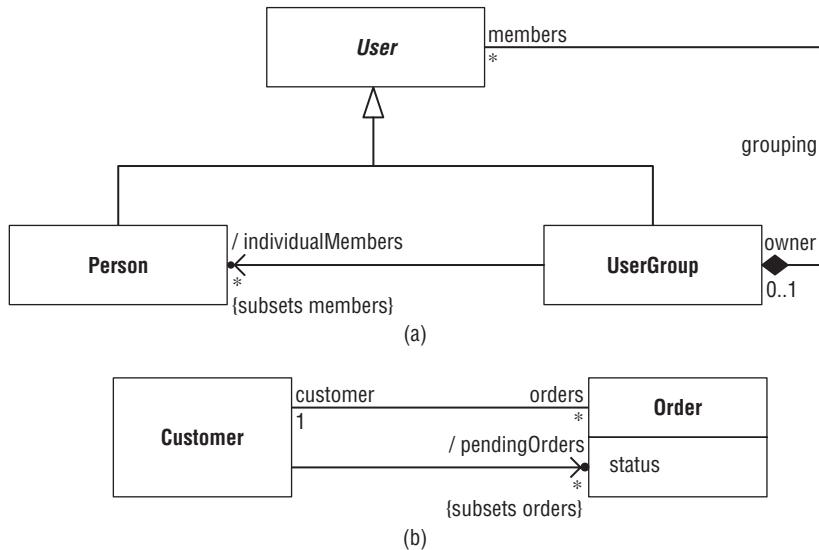
## Subsets and Unions

Let's reconsider the example of a user management system, whose model is shown again in Figure 10-13a. A User Group has the property `members`, which results in all of its individual members and sub-groups, and a derived property `individualMembers` that provides a direct access to its individual members only. However, it is evident that for each object of `UserGroup`, the property `individualMembers` designates a collection of objects of `Person` that must be a sub-collection of the collection designated by the property `members` of the same User Group. In other words, there must be a constraint that relates these two association ends and conceptualizes the fact that the individual members of a User Group are always a subset of all members of that group.

Such a constraint is introduced by *subsetting* association ends in UML. As shown in Figure 10-13a, the association end `individualMembers` is specified to *subset* the association end `members`. This is depicted in diagrams by the keyword `subsets` within the curly braces close to the subsetting association end.

Another example is the already considered order processing system, whose model is shown again in Figure 10-13b. Here, the collection of pending Orders placed by a certain Customer must be a

sub-collection of all Orders placed by that Customer. Therefore, the association end pendingOrders subsets the end orders.



**Figure 10-13: Examples of subsetting association ends. (a) The collection of individual members of a User Group is always a sub-collection of all members of that group. (b) The collection of pending Orders placed by a Customer is a sub-collection of all Orders placed by that Customer.**

In general, an association end can be specified to subset one or more other association ends. Note that although both presented examples in Figure 10-13 have a subsetting association end that is also a derived end, this is pure coincidence. A subsetting association end can (but need not) be derived. However, the following modeling rules apply to subsetting association ends:

- ❑ The subsetting association end (for example, the association end pendingOrders or individualMembers in Figure 10-13), as a typed multiplicity element, must conform to the subsettied association end (for example, the association end orders or members in Figure 10-13, respectively), being also a typed multiplicity element.<sup>11</sup>
- ❑ The class across from the subsetting association end (for example, the class Customer across from the association end pendingOrders, or the class UserGroup across from the association end individualMembers in Figure 10-13) must be the same or a subclass of the class across from the subsettied association end (for example, the class Customer across from the association end orders, or the class UserGroup across from the association end members, respectively).
- ❑ A navigable association end can be subsettied only by a navigable association end.

<sup>11</sup>Standard UML requires only that the type and multiplicity of the subsetting property conform to the type and multiplicity of the subsettied property, respectively. OOIS UML requires the conformance of uniqueness and ordering, too.

## Part III: Concepts

---

Subsetting has the following run-time semantics.<sup>12</sup> Every *subsets* relationship between the subsetting and the subsetted association ends is considered as an implicit constraint attached to the two association ends. The constraint is, therefore, implicitly checked as with any other constraint, whenever a modification of the contents of any of the involved elements may affect the validity of the constraint.

The meaning of the constraint is the following. For every object of the class across from the subsetting association end (for example, of the class *Customer* across from the association end *pendingOrders*, or of the class *UserGroup* across from the association end *individualMembers* in Figure 10-13), the collection *c1* of objects designated by the subsetting end (for example, the association end *pendingOrders* or *individualMembers*, respectively) must be a “sub-collection” of the collection *c2* designated by the subsetted association end (for example, the association end *orders* or *members*, respectively) for the same object. The collection *c1* is a “sub-collection” of the collection *c2* if and only if every element of *c1* appears in *c2* at least the number of times it appears in the sub-collection *c1*. Ordering of the elements is not taken into consideration. In other words, subsetting has no additional semantics for ordered association ends.

As you can see, when the subsetted association end is unique, because the subsetting end must also be unique, being a sub-collection means being a subset in the mathematical sense. On the other hand, if the subsetted end is non-unique and the subsetting end is unique, then *c1* is a sub-collection of *c2* if and only if *c2* contains every element from the set *c1* at least once.

Sometimes the collection designated by the subsetted association end represents a strict union of the collections designated by all its subsetting association ends. For example, the collection of Orders placed by a certain Customer can be derived from the collections of Orders of all possible statuses (initiated, pending, and closed), as shown in Figure 10-14a. Thus, it is a strict union of these subsetting collections, meaning that it contains all the elements from the subsetting collections and nothing else.

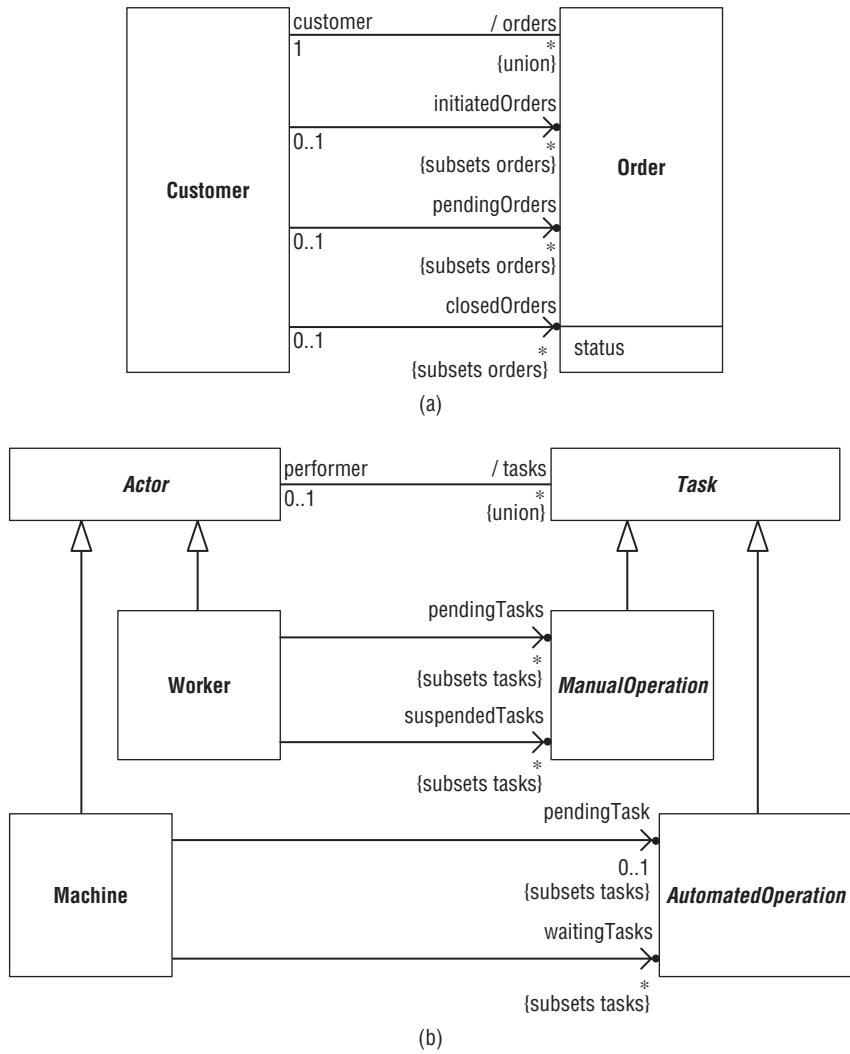
Another example appears in Figure 10-14b, which shows a fragment of the model of a workflow management system. In that system, an Actor is an abstract generalization of all kinds of performers that can accomplish some Tasks. An Actor can be a Worker (that is, a human), or a Machine, while a Task can be a Manual Operation or an Automated Operation of a certain kind. Obviously, a Worker can perform only Manual Operations, while a Machine can perform only Automated Operations.

The set of Tasks assigned to an Actor is accessed through the property *tasks* of the class *Actor*. For a Worker, the set of assigned Tasks is divided into the subsets of pending and suspended Tasks. It is assumed that a Worker can work on an arbitrary number of pending Tasks at a time. On the other hand, a Machine can have, at most, one pending Task (for any domain-specific reason whatsoever), while the other assigned Tasks are waiting for processing on the Machine.

Consequently, the properties *pendingTasks* and *suspendedTasks* of the class *Worker*, as well as the properties *pendingTask* and *waitingTasks* of the class *Machine* subset the property *tasks* of the class *Actor*. On the other hand, for each Worker, the collection of all assigned Tasks, designated by the inherited property *tasks*, is always a strict union of pending and suspended Tasks. Similarly, for each Machine, the collection of all assigned Tasks, designated by the inherited property *tasks*, is always a strict union of the pending and waiting Tasks. This means that the collection designated by the property *tasks* does not contain any other Task that is not contained in one of its sub-collections.

---

<sup>12</sup>This is a specialization by the OOIS UML profile. Standard UML specifies the semantics of subsetting informally and incompletely.



**Figure 10-14: Examples of derived unions.** (a) The collection of all Orders placed by a certain Customer is derived as a union of all collections that subset it. (b) In the model for a workflow management system, the collection of all Tasks assigned to a certain Actor in a workflow is derived as a union of all collections that subset it. These sub-collections are defined specifically for the classes derived from Actor and Task: for an Actor that is a Worker, the set of assigned Tasks is derived as a union of pending Tasks and suspended Tasks, while for an Actor that is a Machine, the set of assigned Tasks is derived as a union of the pending Task and waiting Tasks.

In general, an association end can be designated as a *union*. In diagrams, it is represented with the keyword `union` written inside curly braces near the association end. An association end that is a union is always derived, which is why it is called a *derived union*. For every object of the opposite class, the

## Part III: Concepts

---

collection designated by that end is a “strict union” of all collections designated by all association ends that subset it, for the same object. Being a “strict union” means the following:

- ❑ If the union end is unique, its collection contains all those and only those elements that are contained by the subsetting collections (which are also unique).
- ❑ If the union end is non-unique, its collection contains all those and only those elements that are contained by the subsetting collections (which can be unique or non-unique), and each of them exactly the number of times that is the sum of the number of times it appears in the subsetting collections.
- ❑ Ordering of the elements in the collections is irrelevant.

Effectively, a derived union simply strengthens the constraint that is implied from subsetting. The union must contain all the subsetting collections, and must not contain anything else.

Note that subsets and unions in OOIS UML require checking constraints at run-time that may induce significant computational overhead. Consequently, the modeler must be aware of that overhead and use these concepts carefully.

### Section Summary

- ❑ An association end can be specified as a *subset* of one or more association ends. The subsetting association end, as a typed element, must conform to the sub-setted association end, and the class across from the subsetting end must be the same or a subclass of the class across from the subsetted end.
- ❑ Every subsetting relationship between association ends is an implicit constraint attached to the two association ends. For every object of the class across from the subsetting association end, the collection of objects designated by the sub-setting end must be a sub-collection of the collection designated by the subsetted association end for the same object. Ordering of the elements in the collections is irrelevant.
- ❑ An association end can be specified as a *derived union*. In that case, for every object of the opposite class, the collection designated by that end is a strict union of the collections designated by all association ends that subset it for the same object.

## Acyclic Associations<sup>13</sup>

Some associations allow the objects on both sides of a link to be of the same class. For example, the object playing the role of the subdivision and the object playing the role of the super-division in a link of the association hierarchy in Figure 10-15a are both of the class Department. Similarly, the objects at both sides of a link of the association grouping in Figure 10-15b can be of the class UserGroup because a User

---

<sup>13</sup>This concept is specific to the OOIS UML profile and does not exist in standard UML.

Group can be a member (that is, a subgroup) of a User Group. Such associations are called *reflective*.<sup>14</sup> In general, a reflective association is an association whose links can connect objects of the same class. These are associations that relate a class to itself, as in Figure 10-15a, or to a (direct or indirect) generalization of that class, as in Figure 10-15b.

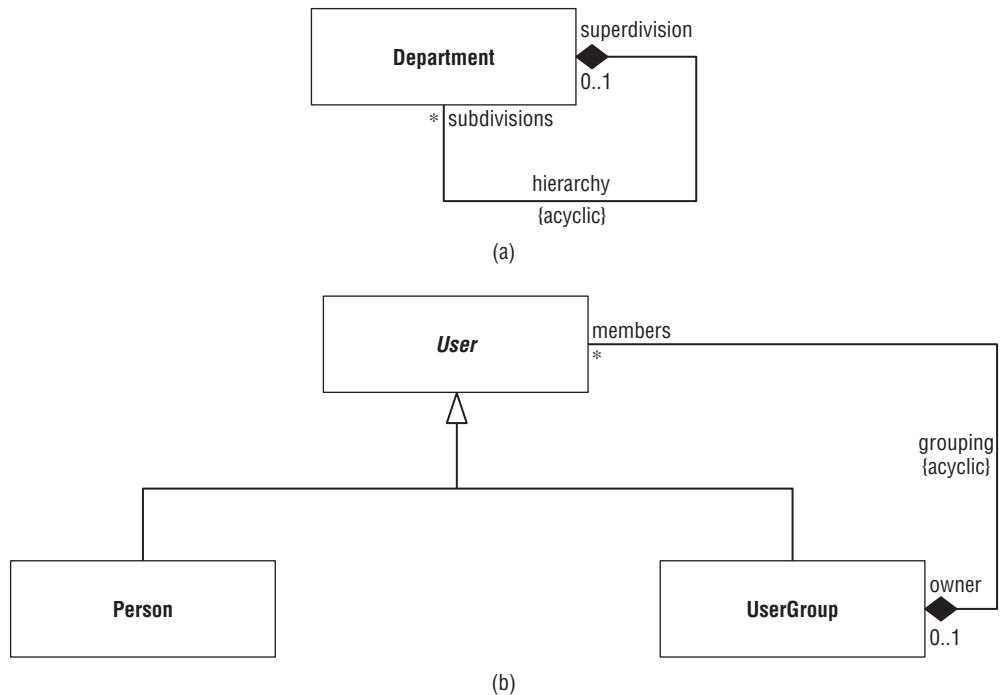


Figure 10-15: Examples of acyclic reflective associations. (a) A Department cannot be a direct or indirect subdivision of itself. (b) A User Group cannot be a direct or indirect member of itself.

The default semantics of associations in UML do not prevent a link of a reflective association from connecting an object to itself. For example, the default semantics of association would allow a Department to be a subdivision of itself, or a User Group to be a member of itself. However, in practice, reflective associations are often used to model hierarchies of objects, such as hierarchies of Departments in Figure 10-15a or of User Groups in Figure 10-15b. In such hierarchies, it is unacceptable that an object becomes a descendant of itself, not only directly, but also indirectly. For the mentioned examples, a Department must not be a direct or indirect subdivision of itself, or a User Group must not be a direct or indirect member of itself.

In general, such conceptual constraint means that a directed graph consisting of the objects of the related class(es) and the links of the reflective association does not have cycles, but is a tree. (The direction of the edges of the graph is implied from the inherent orientation of the association, that is, by the roles of its ends. For example, the direction of the links of the association hierarchy in Figure 10-15b is to be taken

<sup>14</sup>It should be noted that the term “reflective” in this context has nothing to do with the reflection mechanism that provides the access to the information from the model during execution.

## Part III: Concepts

---

either from the super-division toward the subdivision, or the other way around, but always the same.) A cycle would assume a link connecting an object to itself (a direct cycle), or a directed path of links starting from an object and ending at the same object (an indirect cycle).

To prevent circular link paths for a reflective association, the association can be annotated as *acyclic*. This is depicted in diagrams by the keyword `acyclic` written within curly braces near the midpoint of the line representing the association. This characteristic has the semantics of an implicit constraint that is attached to the association. As with any other constraint, it is implicitly checked on every action that may affect its validity. In this case, it is checked on every creation of a link of the reflective association. The constraint prevents cycles, as already described. If applied to a non-reflective association, this specification has no semantic significance (it has no effects at run-time).

Note that the checking of such a constraint at run-time may induce significant computational overhead because the run-time engine must traverse a potentially huge object subgraph for identifying (absence of) circular link paths. Consequently, the modeler must be aware of that overhead and use this concept carefully.

### Section Summary

- ❑ A *reflective* association is an association whose links can connect objects of the same class. These are associations that relate a class to itself, or to a (direct or indirect) generalization of that class.
- ❑ A reflective association can be annotated as *acyclic*, imposing an implicit constraint on the association.
- ❑ The constraint means that a directed graph consisting of the objects of the class(es) related by the reflective acyclic association and the links of the association does not have cycles, but is a tree. A cycle would assume a link connecting an object with itself (a direct cycle), or a directed path of links starting from an object and ending at the same object (an indirect cycle).

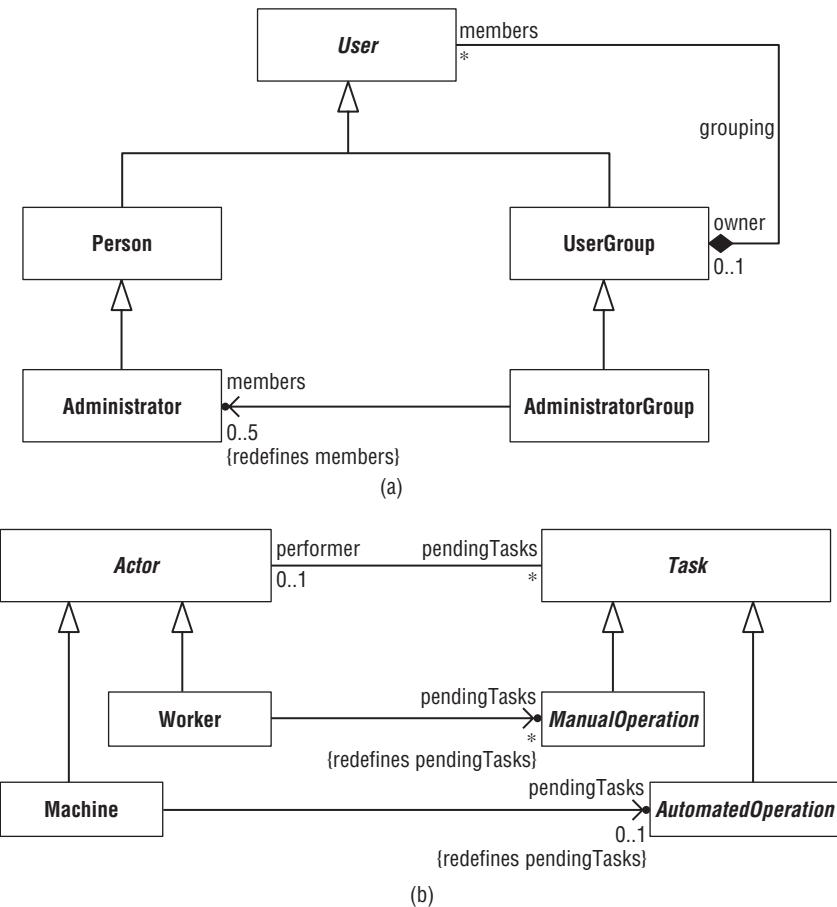
## Redefinition of Association Ends

As already described for redefining attributes, in standard UML, features of classifiers (including properties that are association ends) can be *redefined* in specializing classifiers. A (directly or indirectly) specializing classifier can have a feature — for example, a property, that redefines a feature (property) from a generalizing classifier. The redefining feature of the specializing classifier represents a specification that augments, constrains, or overrides the specification of the redefined feature. Consequently, for the instances of the specializing classifier, the specification provided by the redefining feature holds, along with or instead of (parts of) the specification of the redefined feature.

As for properties of classes (including attributes and association ends), a property of a specializing class may redefine another property of a generalizing class, with the effect of augmenting, constraining, or overriding the definition of the redefined property. This does not mean that the instances of the specializing class will have a separate slot as a run-time manifestation of the redefined property. They will just have the same slot as a run-time manifestation of the redefined property, but with possibly

different characteristics. In other words, for the instances of the specializing class, the rules implied from the redefining property will hold for the slot that is the run-time manifestation of the redefined property.

Figure 10-16 provides two examples of association end redefinitions. In Figure 10-16a, the association grouping specifies that a User Group, in general, can have an arbitrary number of Users as its members. However, an Administrator Group, which is a special kind of User Group, can have only Administrators as its members, and at most five of them. Of course, an Administrator is also a kind of User. Note that an Administrator (being also a Person and a User) can still be a member of a general User Group because it has not redefined the property owner.



**Figure 10-16: Redefinition of association ends.** (a) While a general User Group can have an arbitrary number of Users as its members, an Administrator Group can have at most five Administrators as its members. (b) In general, an Actor can have an arbitrary number of pending Tasks. However, a Worker can have only Manual Operations as its pending Tasks, while a Machine can have at most one Automated Operation as its pending Task.

## Part III: Concepts

---

Similarly, in Figure 10-16b, an Actor, in general, can have an arbitrary number of pending Tasks. However, a Worker, as a kind of Actor, can have only Manual Operations as its pending Tasks, while a Machine can have at most one Automated Operation as its pending Task. It is interesting to note that, as long as the class `ManualOperation` does not redefine the property `performer`, it can be assigned as a pending Task of an Actor of a different kind than Worker (or Machine, since Machine constrains its pending Tasks to Automated Operations only).

In OOIS UML, only non-static association ends owned by classes can be redefined. Moreover, redefinition of association ends is constrained to some extent in comparison with standard UML, and the semantics of redefinition of particular aspects is more precisely defined as follows:

- ❑ The visibility and name of the association end cannot be redefined. More precisely, the visibility and name of the redefining association end must exactly match the visibility and name of the redefined association end.<sup>15</sup>
- ❑ The redefining association end may redefine the type of the redefined association end, by being attached to a class that is the same or (normally) a subclass of the class at the redefined association end. The semantics of such redefinition is derived. It implies a constraint on the type of the linked objects, implicitly defined in the context of the specializing class across from the redefining association end. For the example given in Figure 10-16a and the redefinition of the association end `members`, there is an implicit constraint in the specializing class `AdministratorGroup` that constrains the type of the objects linked to that property.
- ❑ The redefining association end may redefine the multiplicity of the redefined association end, by specifying a multiplicity that is included in the multiplicity of the redefined association end. The semantics of such a redefinition is derived. It implies a constraint on the multiplicity of the association end, implicitly defined in the context of the specializing class across from the redefining association end. As a special case, the redefining multiplicity can be `0..0` (provided that it is included in the redefined multiplicity), meaning that the slots in the objects of the specializing class cannot have such links at all.
- ❑ The redefining association end may be derived, although the redefined association end is not derived. This means that for the objects of the specializing class, the collection of the redefined association end will be derived in some way from other sources of information. If the redefined association end is derived, the redefining association end must also be derived.

In OOIS UML, the redefinition of static association ends and association ends owned by associations is not allowed. If an end of an association is a redefining end, the opposite end of that association can be either a redefining end, or a non-navigable end belonging to the association. An association having a redefining end has no semantic implications itself; only its ends do.

In the UML notation, the redefinitions can be made explicit with the use of a `{redefines ...}` string near the association end, as shown in Figure 10-16.

---

<sup>15</sup>In standard UML, if a feature in the specializing classifier has the same name as a feature of the same kind from the generalizing classifier, then the former implicitly redefines the latter, even though they are not explicitly related in the model.

### Section Summary

- ❑ In UML, a property of a specializing classifier can *redefine* a property of a generalizing classifier, providing a compatible specification that augments, constrains, or overrides the specification of the redefined property.
- ❑ The existence of the redefining property in the specializing class does not imply the existence of a separate slot as its run-time manifestation in the instances of that class, but just augments, constrains, or overrides the characteristics of the slot implied from the redefined property and the values it can hold.
- ❑ In OOIS UML, only some aspects of non-static association ends owned by classes can be redefined:
  - ❑ The visibility and name of the redefining association end must match those of the redefined association end.
  - ❑ The redefining association end may redefine the type of the redefined association end by a class that is the same or a subclass of the class at the redefined association end. Such a redefinition implies a constraint on the type of the linked objects, implicitly defined in the context of the specializing class across from the redefining association end.
  - ❑ The redefining association end may redefine the multiplicity of the redefined association end by specifying a multiplicity that is included in the multiplicity of the redefined association end. Such a redefinition implies a constraint implicitly defined in the context of the specializing class across from the redefining association end.
  - ❑ The redefining association end may be derived, although the redefined association end is not derived. This means that for the objects of the specializing class, the collection of linked objects at the redefined association end will be derived in some way from other sources of information. If the redefined association end is derived, the redefining association end must also be derived.
- ❑ OOIS UML does not support redefinition of static association ends and association ends owned by associations.
- ❑ The notation for redefining association end is {*redefines* ... }

## Actions on Binary Associations

This section examines actions on binary associations.

## Part III: Concepts

---

### Creation of Links

Instances of associations (that is, links) are created by the Create Link action. For example, in the OOIS UML native detail-level language, the following statement creates a new link of the association `attends` that links the objects referred to by the variables `aPerson` and `aCourse`:

```
new attends(aPerson, aCourse);
```

So, the input parameters of the action Create Link are the association whose link is to be created and the objects to be linked. The association must be accessible from the place of the action specification. The action does not have output parameters.

The Create Link action (as with most other OOIS UML actions) can be invoked in two different ways: by a statically modeled action and dynamically through the reflection mechanism.

In the first case, the action is specified using a detail-level language, whereas the association whose link is to be created is statically specified and checked at compile time. In other words, if any error exists in the specification of the action (such as the association does not exist or is inaccessible), it is reported at compile time. No run-time exceptions will be raised because of these reasons. In the OOIS UML native detail-level language, this kind of specification is supported by the operator `new`. Its basic form is as follows:

```
new associationName (obj1, obj2)
```

If the roles of the two objects provided by the arguments are unambiguously defined (for example, if the association is not reflective), the two arguments can be provided in any order. For example, if the variable `aPerson` is of type `Person` and the variable `aCourse` is of type `Course`, the object referred to by `aPerson` will play the role of the student, and the object referred to by `aCourse` will play the role of the attended course in the newly created link, in both of the following cases:

```
new attends(aPerson, aCourse);
new attends(aCourse, aPerson);
```

However, if the association is reflective and both provided arguments could play any of the roles, the order of the arguments corresponds to the order of the association ends in the UML model. For example, if the association `organization` relates the class `Department` with itself, whereby the association end `superdivision` is the first, and the end `subdivision` is the second in the model, then the following action will create a link in which `dept1` will play the role of `superdivision` and `dept2` of `subdivision`:

```
new organization(dept1, dept2);
```

If an end of the association is ordered, the index (position) of the object at that end can be provided by a separate argument, immediately following the object:

```
new associationName (obj1, index1, obj2, index2)
```

The index should be a non-negative integer value. The index can be omitted even though the corresponding association end is ordered. In that case, or when it has no value (that is, it has zero cardinality), the object at that end is placed as the last in the corresponding collection.

In conventional OO programming languages, the same effect can be achieved using the same notation with new, or by the means of static operations of classes provided implicitly by the implementation because the semantics of the built-in operator new of these languages may not be adequate. For example, to create a new link of the association attends, you would write the following in C++:

```
attends::createLink(aPerson, aCourse);
```

Or, similarly, you would write the following in Java and C#:

```
attends.createLink(aPerson, aCourse);
```

The second means of invoking the same action is through the reflection mechanism of OOIS UML. In that case, the association whose link will be created is provided dynamically, as an input pin of the action. Therefore, all violations (such as when the association is inaccessible) are reported through run-time exceptions. For example, in the OOIS UML native detail-level language, this is specified as follows:

```
UML::Action::createLink("Easylearn::attends", aPerson, aCourse);
```

In this example, the reflection library, stored in the package named `UML` of the OOIS UML library (in particular, the class `Action` from this package) contains the (static) operations for all actions executable through reflection. The operation `createLink` from this class accepts a string with the name of the association whose link will be created (`attends` in this case), and the references to the two objects to be linked.

Another operation from the same `Action` class accepts (a reference to) an object of type `Association`, which represents an association from the model within the reflection data. For each association in the system's model, an instance of the class `Association` will exist in the environment. In order to access such an object, another mechanism can be used:

```
Association attendsAssoc =
 UML::Reflection::getAssociation("Easylearn::attends");
UML::Action::createLink(attendsAssoc, aPerson, aCourse);
```

In a conventional OO programming language, the accompanying OOIS UML library must support the reflection mechanism in a completely equivalent manner.

Regardless of the method of invocation, the following rules must be obeyed at the place of invocation of the action:

- ❑ The specified association must exist and must be accessible at the place of the action invocation (that is, within the namespace of the method in which the action occurs), according to the visibility rules for named elements.
- ❑ The objects provided as arguments must be of types that conform to the types of the corresponding association ends.

In the case of static invocation, a compilation error will be reported if any of these rules is violated. In the case of dynamic invocation through reflection, a run-time exception of an adequate type is raised if a rule is violated.

## Part III: Concepts

---

Regardless of the method of invocation, the action Create Link has the following effect at run-time:

1. For every ordered association end and the corresponding provided index (position):
  - ❑ If the given position (index) has a value that is not included in the multiplicity of the association end, the association extent remains unaffected and the exception `MultiplicityViolationException` is raised (exceptions are explained later in this book). In particular, if the given position is less than the lower bound of the multiplicity allows (in effect, it is negative), the exception is of type `LowerBoundViolationException`, which is a subtype of `MultiplicityViolationException`. Similarly, if the given position is greater than the upper bound of the multiplicity allows, the exception is of type `UpperBoundViolationException`, which is a subtype of `MultiplicityViolationException`.
  - ❑ If the given position (index) has a value that is out of the range defined by the current cardinality of the association end (but is still included in the end's multiplicity), the effect is the same as if the position were not given (that is, the object is added to the end of the corresponding collection).
2. If the creation of a link would violate the limited upper multiplicity bound of any association end, an exception of type `UpperBoundViolationException` is raised and the association extent remains unaffected.
3. A new link of the association is created and added to the association extent as described in the earlier section, "Semantics of Binary Associations and Association Ends."

Note that if this action is performed within a constrained action group, such as during the initialization of the object at any end, the inspection of the multiplicity constraints is deferred to the moment when the group is completed. After all, if an association end has its lower multiplicity bound greater than zero, it is necessary to allow the "transitional" period from the basic construction of the object, when the cardinality of its slot is still zero, up to the moment of the completed initialization, when the cardinality must obey the multiplicity constraint. The execution of the constructor of such an object is the right context for the actions that create the initial links until the cardinality of the slot satisfies its multiplicity. This is similarly true for the finalization of an object.

### Section Summary

- ❑ A link of an association is created by the Create Link action.
- ❑ This action can be invoked as follows:
  - ❑ Statically, `new associationName (obj1, index1, obj2, index2)`
  - ❑ Dynamically, through the reflection mechanism; for example,  
`UML::Actions::createLink("attends", aPerson, aCourse);`

## Destruction of Links

Links are destroyed with the Destroy Link action, which can be invoked in the OOIS UML native detail-level language in several different ways:

- ❑ Using the operator `destroy` and specifying the association statically by its name, as in the following example:

```
destroy attends(aPerson, aCourse);
```

- ❑ Using the operation `destroyLink` of the `UML::Action` class and specifying the association dynamically, through the reflection mechanism, as in the following example:

```
UML::Action::destroyLink("Easylearn::attends", aPerson, aCourse);
```

or:

```
Association attendsAssoc =
 UML::Reflection::getAssociation("Easylearn::attends");
UML::Action::destroyLink(attendsAssoc, aPerson, aCourse);
```

In other conventional OO programming languages, the first option would correspond to the notation in C++:

```
attends::destroyLink(aPerson, aCourse);
```

Or, similarly, the following would be used in Java and C#:

```
attends.destroyLink(aPerson, aCourse);
```

The second option may be available directly as presented or with minor syntactical differences.

The general forms of the three described ways of invocation of the Destroy Link action are as follows:

```
destroy associationName (obj1, index1, obj2, index2, deleteAll)
UML::Action::destroyLink(associationName, obj1, index1, obj2, index2, deleteAll)
UML::Action::destroyLink(association, obj1, index1, obj2, index2, deleteAll)
```

In the first form, the association is specified statically, as a named element, by its (possibly qualified) name. In the second and third forms, the association is specified dynamically, through the reflection mechanism. In the second form, the association is specified by its (possibly qualified) name given as a string argument, whereas in the third form it is specified by an instance of the classifier `Association` from the reflection package.

The same remark about the ordering of objects in the argument list applies to creation of links. If the roles of the objects can be unambiguously determined by their types, their ordering is irrelevant and can be arbitrary. If, however, the association is reflective and both objects can play both roles, the ordering of arguments corresponds to the ordering of association ends in the model.

## Part III: Concepts

---

The index after each of the objects can exist if the corresponding association end is ordered. The index should be a non-negative integer value. For an ordered association end, either the object or the index can be omitted, or the index can have no value (that is, have zero cardinality), with the effects described in the earlier section, “Semantics of Binary Associations and Association Ends.”

The argument `deleteAll` is of type `Boolean[0..1]` and can always be omitted. It has effect only if both association ends are non-unique; otherwise, it is ignored. If both association ends are non-unique and this argument has the value `true`, all the links connecting the given objects are removed from the association, and the indices, if provided, are ignored in that case. If this argument has the value `false` or has no value or is omitted, only the link or links designated by the other arguments are removed from the association.

Regardless of the method of invocation, the following rules must be obeyed at the place of invocation of the action:

- ❑ The specified association must exist and must be accessible at the place of the action invocation (that is, within the namespace of the method in which the action occurs), according to the visibility rules for named elements.
- ❑ The objects provided as arguments must be of types conform to the types of the corresponding association ends.

In the case of static invocation, a compilation error will be reported if any of these rules is violated. In the case of dynamic invocation through reflection, a run-time exception of an adequate type is raised if a rule is violated.

Regardless of the method of invocation, the action `Destroy Link` has the following effects at run-time, in this order:

- 1.** For every ordered association end and the corresponding provided index (position):
  - ❑ If the given index (position) has a value that is not included in the multiplicity of the association end, the association extent remains unaffected and an exception is raised as for other similar actions on ordered values.
  - ❑ If the given index (position) has a value that is out of the range defined by the current cardinality of the collection (but is still included in the association end’s multiplicity), the association extent remains unaffected.
- 2.** Otherwise, if the removal of the specified link(s) would violate the lower multiplicity bound of any of the association ends, an exception of type `LowerBoundViolationException` is raised and the association extent remains unaffected.
- 3.** Finally, if none of the previous occurred, the specified link or links are removed from the association extent, according to the semantics described in the earlier section, “Semantics of Binary Associations and Association Ends.” If such links do not exist in the association extent, the action has no effects.

Similar to the `Create Link` action, if this action is performed during the initialization or finalization of the object at any end, the inspection of the multiplicity constraints is deferred to the moment when the initialization/finalization process is completed, because all the actions performed during an object initialization or finalization are implicitly considered as a constrained action group.

### Section Summary

- ❑ Links are deleted with the Destroy Link action:

```
destroy associationName (obj1,
 index1, obj2, index2, deleteAll)
```

or

```
UML::Action::destroyLink(anAssociation, obj1,
 index1, obj2, index2, deleteAll)
```

### **Derived Actions on Properties**

When an association end belongs to the class at the opposite end, it is a property of that class and is manifested at run-time as a slot in its objects. Consequently, the set of actions that work on properties is the same for both kinds of properties — attributes and association ends. Basically, such an action works on a property and has the same interface, so that the modeler can treat attributes and association ends in the same way. If the property is an attribute, the effect of the action is as described in Chapter 9 in the section “Actions on Attributes.” If the property is an association end, the action has the run-time semantics as described in the section “Semantics of Binary Associations and Association Ends,” earlier in this chapter. However, the effect of the action on the collection designated by the association end is, in general, the same as if the property were an attribute. Of course, the action affects the association extent and both its ends.

In particular:

- ❑ The declaration of the action is the same.
- ❑ All static rules applicable to actions on attributes also hold for association ends as properties of their owner classes, because they are named elements and multiplicity typed elements.
- ❑ The property that is an association end can be accessed through the reflection mechanism in the same way as an attribute.
- ❑ The multiplicity constraints are checked at run-time not only on the slot addressed in the action, but also on the affected slots at the opposite ends of links.
- ❑ The comparison of the values in the collection designated by the addressed slot is based on object identity for association ends, and on the data value for attributes. The elements of the collections are referred to as values in both cases. There are always references to classes or data types.
- ❑ The concurrency control mechanism may lock not only the object owning the addressed slot, but also all objects linked over that slot, in order to preserve consistency.
- ❑ An action from the set of Read Property actions has the same result as the corresponding Read Attribute action, when the property is taken to designate a collection as described in the earlier section, “Semantics of Binary Associations and Association Ends.”

## Part III: Concepts

---

- ❑ An action from the set of Write Property actions basically affects the association extent as described in the earlier section, “Semantics of Binary Associations and Association Ends,” but has an indirect effect on the collection designated by the addressed property analogous to the corresponding Write Attribute action. In effect, these actions are just other forms of the actions Create Link and Destroy Link, described previously.
- ❑ The actions can be implemented in a conventional detail-level language in a similar way as for attributes.

The entire set of actions on association ends as properties in OOS UML is listed briefly in the summary of this section. For more detailed explanations of the static rules and dynamic semantics, refer to the section “Actions on Attributes” in Chapter 9.

### Section Summary

- ❑ The set of actions that work on properties is the same for both kinds of properties — attributes and association ends. The actions on properties that are association ends have the same interface as actions on attributes, but affect the association extent and both its ends. The effect on the collection designated by the addressed slot is analogous to the effect of the corresponding attribute action.
- ❑ Read Size of Property:  

```
prop->size() : Integer[1]
```
- ❑ Read Property:  

```
prop : T[m]{p}
prop->read() : T[m]{p}
```
- ❑ Read Value from Property:  

```
prop->val() : T[min(m.lower,1)..1]
```
- ❑ Count Occurrences of Value in Property:  

```
prop->count(val:T[0..1]) : Integer[1]
```
- ❑ Read Value from Ordered Property:  

```
prop->at(pos:Integer[0..1]) : T[0..1]
throws MultiplicityViolationException
prop->first() : T[min(m.lower,1)..1]
prop->last() : T[min(m.lower,1)..1]
```

- ❑ Clear Property:

```
prop->clear() : C[0..1]
```

- ❑ Remove Value from Property:

```
prop->remove(val:T[0..1]) : C[0..1]
throws LowerBoundViolationException
```

- ❑ Remove One Occurrence of Value from Property:

```
prop->removeOne(val:T[0..1]) : C[0..1]
throws LowerBoundViolationException
```

- ❑ Remove Value from Ordered Property:

```
prop->removeAt(pos:Integer[0..1]) : C[0..1]
throws MultiplicityViolationException
prop->removeFirst() : C[0..1]
throws LowerBoundViolationException
prop->removeLast() : C[0..1]
throws LowerBoundViolationException
```

- ❑ Set (Assign) Property:

```
prop->set(expression) : C[0..1]
prop = expression : C[0..1]
```

- ❑ Add Value to Property:

```
prop->add(val:T[0..1]) : C[0..1]
throws UpperBoundViolationException
```

- ❑ Add Value to Ordered Property:

```
prop->addAt(val:T[0..1], pos:Integer[0..1]) : C[0..1]
throws MultiplicityViolationException
prop->addFirst(val:T[0..1]) : C[0..1]
throws UpperBoundViolationException
prop->addLast(val:T[0..1]) : C[0..1]
throws UpperBoundViolationException
```

- ❑ The special symbol `null` represents absence of value in a single-valued result of an action on a property:

```
if (prop->first()==null) ...
```

*Continued*

- ❑ Freeze Property:

```
prop->freeze()
```

- ❑ Unfreeze Property:

```
prop->unfreeze()
```

- ❑ Iteration through the collection by filtering only elements of type T and visiting each value v:

```
prop->forEach(v,T) statement
```

- ❑ Iteration through an ordered collection in the reverse order:

```
prop->forEachReverse(v,T) statement
```

### Derived Actions on Association Ends

When an association end belongs to the association, the actions that work on properties cannot be used. However, there is a set of actions on association ends that are equivalent to the actions on properties. These actions are applicable to association ends that belong to classes, as well as to those that belong to associations.

For example, to read the collection designated by the association end `attendedCourses` for an object referred to by `aPerson`, you can write the following:

```
attends::attendedCourses->read(aPerson)
```

Similarly, to add a value to that collection (actually, to create a link), you can write the following:

```
attends::attendedCourses->add(aPerson, aCourse)
```

As you can see, the notation for the actions on association ends is similar to the notation of the corresponding actions on properties, except for the following:

- ❑ The left operand of the operator `->` specifies the association end only, not a slot of an object for that end (as with `object.property`).
- ❑ The first argument in the argument list of the action is always the object of the class at the opposite end. The rest of the arguments are the same. If this argument has no value (that is, has zero cardinality), the action has no effect.

All other aspects of the actions (such as the static rules and the run-time semantics) are exactly the same. The only additional static rule is that the referred association end must be accessible and navigable. (Recall that it is always navigable if it is owned by the class.)

### Section Summary

- ❑ For every Read or Write Property action, there is an analogous Read or Write Association End action, with a completely similar notation and the same semantics, except for the following:
  - ❑ The left operand of  $\rightarrow$  specifies the association end only, not a slot of an object for that end.
  - ❑ The first argument in the argument list of the action is always the object of the class at the opposite end. The rest of the arguments are the same.
- ❑ For any association end, belonging to the opposite class or to the association, an association end action can be invoked, provided that the end is navigable.

### **Specific Actions on Association Ends**

Direct and naive application of some of the described actions on association ends can sometimes lead to very time-consuming execution, which may be unacceptable in practice. This is especially the case when the collection of linked objects is very large. Consider an association that relates a city and its inhabitants — hundreds of thousands or even millions of objects can be linked to one single object representing a city. Although the basic set of actions on associations and their ends is complete, so that all necessary manipulations can be achieved with one action or a combination of several actions, the modeler must always be aware of the physical constraints of the system — the implementation may require considerable processing of persistent storage to perform the specified actions.

For this reason, OOIS UML provides several specific actions on association ends. Although their effect could be achieved by combining other existing actions, their implementation may be optimized and, thus, made more efficient. This is why the modeler should give them preference in the appropriate cases.

The first such action is the Reorder action for ordered association ends. It simply exchanges the positions (indices) of the two given values in the ordered collection designated by an association end. Its signature is as follows:

```
assocEnd->reorder(obj:X[0..1], index1:Integer[0..1], index2:Integer[0..1])
```

If the association end belongs to a class, it can be performed on the property:

```
prop->reorder(index1:Integer[0..1], index2:Integer[0..1])
```

It exchanges the values at the two given positions (or, put another way, exchanges the indices of the values at the two given positions). If either of the arguments has no value, the action has no effect.

## Part III: Concepts

---

If the association end is unique, any of the exchanging items can be specified either by its position or by its value:

```
assocEnd->reorder(obj:X[0..1], obj1:T[0..1], obj2:T[0..1])
prop->reorder(obj1:T[0..1], obj2:T[0..1])
```

The second such action is the Replace (or Relink) action. It destroys one or several links and creates a new link in a single atomic action. Its signature is as follows:

```
assocEnd->replace(obj:X[0..1], obj1:T[0..1], obj2:T[0..1])
prop->replace(obj1:T[0..1], obj2:T[0..1])
```

It has the compound semantics of two actions performed as a constrained group:

```
assocEnd->remove(obj:X[0..1], obj1:T[0..1]);
assocEnd->add(obj:X[0..1], obj2:T[0..1]);
```

There are variants of this action where `removeOne` is performed instead of `remove`, and/or `addAt` is performed instead of `add`.

When the collection designated by an association end is very large, it is most often not necessary to fetch the entire collection, but just its subset consisting of the objects satisfying some criterion. The read action with an additional argument, specifying the criterion, does this:

```
assocEnd->read(obj:X[0..1], criterion:Text[0..1])
prop->read(criterion:Text[0..1])
```

The criterion is a textual specification of a constraint specified in the scope of the object linked at that end. It represents a filter for the collection of returned objects. If it has no value, the action returns the entire collection. For example, this action returns the collection of inhabitants of the given city that are born in July 1968:

```
aCity.inhabitants->read("birthday.year=1968 and birthday.month=7");
```

### Section Summary

- Reorder:

```
assocEnd->reorder(obj:X[0..1], index1:Integer[0..1],
index2:Integer[0..1])
prop->reorder(index1:Integer[0..1], index2:Integer[0..1])
```

Replace (Relink):

```
assocEnd->replace(obj:X[0..1], obj1:T[0..1], obj2:T[0..1])
prop->replace(obj1:T[0..1], obj2:T[0..1])
```

Read with Filter:

```
assocEnd->read(obj:X[0..1], filter:Text[0..1])
prop->read(filter:Text[0..1])
```

## N-ary Associations

Both standard UML and the OOIS UML profile support a general case of associations with more than two ends — the so-called N-ary associations. Although much less frequently used in practice than binary associations, N-ary associations may sometimes offer a convenient modeling solution to a particular problem. Many aspects of N-ary associations are simple generalizations of those of binary associations. However, some of them require additional clarifications or definitions given in this section.

### **Notion of N-ary Association**

An association in general may relate more than two classes, or, more precisely, may have more than two ends. Such associations are referred to as *N-ary associations*.<sup>16</sup>

Consider, for example, the associations shown in Figure 10-17. In Figure 10-17a, the association `timetable` models scheduling of classes in a school system. It is a relationship whose link relates an object of `Course`; an object of `Room`, as the place of the class; and an object of `TimeSlot`, as the time of the class.

Figure 10-17b shows a part of the model for an information system of a team sport competition. Every Team participating in the competition has its members playing certain roles, such as a goalkeeper, a defender, a coach, a director, and so on.

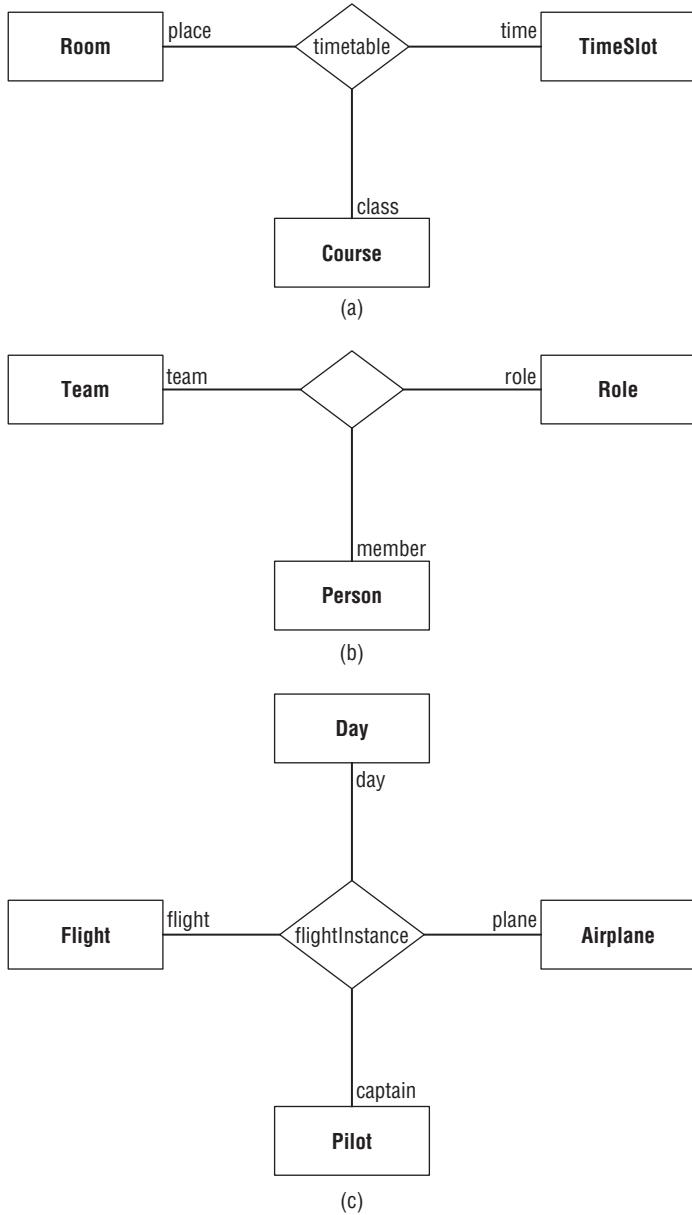
Finally, in Figure 10-17c, an instance of a flight (that is, the concrete incarnation of a flight on a certain day) is modeled with an association with four ends. The class `Flight` models the flight, which is defined by its code, departure place, arrival place, time, and so on. However, for every instantiation of that flight on a certain day, allocation of the aircraft and crew is performed. In particular, a flight instance relates the `Flight`, the `Day`, the `Airplane`, and the `Pilot` in charge.

---

<sup>16</sup>Strictly speaking, a binary association is just a special case of an N-ary association, where  $N$  is equal to two. However, for the sake of clarity, this discussion considers only the associations having more than two ends. Many general rules for N-ary associations also hold for binary associations as their specializations, however.

## Part III: Concepts

---



**Figure 10-17: Examples of N-ary associations.** (a) In an educational institution, a timetable schedules a Course in a certain Time Slot and in a certain Room. (b) In a sports competition, every participating Team has a set of members playing certain Roles (for example, a goalkeeper, a defender, a coach, and so on). (c) In an airline information system, an instance of a flight is defined by the Flight (flight number, departure place, destination, departure time, and arrival time), the date, the Airplane allocated to that flight, and the Pilot in charge.

As with a binary association, an N-ary association specifies a collection of links. However, a link of an association with N ends is an N-tuple, each component of which is an object identity. Pictorially, a link of an N-ary association is not just an edge having two ends attached to objects, but rather a “star” having N ends attached to objects. All N ends must be attached to objects on link creation, and cannot be detached partially — only the link as a whole can be destroyed. Moreover, when any of the objects at the ends is destroyed, the link itself is destroyed, too.

Consequently, an N-ary association extent is a collection of N-tuples with all components having a nonempty object identity. For the example in Figure 10-17a, at a certain moment in time, the extent of the association `timetable` may consist of the tuples rendered in the following table:

| <code>class:Course</code>      | <code>time:TimeSlot</code> | <code>place:Room</code> |
|--------------------------------|----------------------------|-------------------------|
| Operating Systems              | Mon 10am-12noon            | 56                      |
| Operating Systems              | Wed 10am-12noon            | 56                      |
| Object-Oriented Programming    | Mon 10am-12noon            | 311                     |
| Algorithms and Data Structures | Mon 12noon-2pm             | 56                      |
| Relational Databases           | Wed 2pm-4pm                | 312                     |

An N-ary association with more than two ends must always be rendered in diagrams using the diamond notation, as shown in Figure 10-17.

### Section Summary

- ❑ An association may have two or more ends. An association with more than two ends is referred to as an N-ary association.
- ❑ Binary associations are just special cases of N-ary associations with N=2.

## Semantics of N-ary Associations and Association Ends

The generalization of the semantics of binary associations to N-ary associations is rather straightforward, when the following concept is introduced. The *context* of a given association end is the collection of other ends of the same association. This collection is ordered, as the ends of an association are always ordered in the model. For the example given in Figure 10-17a, the context of the association end `time` is the collection of the ends `class` and `place`, while the context of the end `place` is the collection of the ends `class` and `time`. In certain instances, this term will be used to refer to the collection of classes at these other ends. For example, the context of the association end `time` may also refer to the collection of the classes `Course` and `Room`, while the context of the end `place` may refer to the collection of the classes `Course` and `TimeSlot`.

Just as the extent of a binary association represents an unordered collection (possibly having duplicates) of ordered pairs, the extent of an N-ary association represents an unordered collection (possibly having

## Part III: Concepts

---

duplicates) of ordered N-tuples, where each component of an N-tuple represents the (never empty) identity of an object of the class at the corresponding end.

The semantics of an association end (more precisely, the effect of a Read Links action for the association end) is the following. Let's associate an object to each component in the context of that end. These components select a sub-collection of all links in the association extent. The selected sub-collection of links from the association extent identifies a collection of objects at the considered "open" end to read. More formally, an association end  $p$  of an association with  $N$  ends maps a tuple with  $N-1$  objects of the context of that end to a collection of objects of the class at that open end.

Let's reconsider the example in Figure 10-17a and the association `timetable` consisting of the tuples rendered in the following table:

| <code>class:Course</code>      | <code>time:TimeSlot</code> | <code>place:Room</code> |
|--------------------------------|----------------------------|-------------------------|
| Operating Systems              | Mon 10am-12noon            | 56                      |
| Operating Systems              | Wed 10am-12noon            | 56                      |
| Object-Oriented Programming    | Mon 10am-12noon            | 311                     |
| Algorithms and Data Structures | Mon 12noon-2pm             | 56                      |
| Relational Databases           | Wed 2pm-4pm                | 312                     |

Consider the association end `place` as the open end to read. The Read Links action specifies an object for all of the other ends (`class` and `time`) and returns the collection of the objects at the open end. For example, the result of a Read Links action for the tuple (Operating Systems, Mon 10am-12noon) is the collection consisting of a single object representing the room 56, whereas the same action for the tuple (Operating Systems, Wed 2pm-4pm) results in an empty collection because there is no course on Operating Systems at 2 p.m. on Wednesday. If `class` is the open end, the same action for the tuple (Wed 2pm-4pm, 312) would return a collection consisting of the object Relational Databases.

If the considered end is ordered, the designated collection will be ordered. If the end is unique, the collection is a set obtained by collapsing the duplicates from the selected tuples; otherwise, it may contain duplicate elements.

The effect of Create Link and Destroy Link actions is completely analogous to binary associations, except that they specify an N-tuple of object identities instead of pairs.

### Section Summary

- ❑ The term *context* of the given association end refers to the collection of other ends of the same association or the classes at those ends.
- ❑ The extent of an N-ary association represents a collection (possibly having duplicates) of N-tuples, where each component of an N-tuple represents the identity of an object of the class at the corresponding end.

- The semantics of an association end (or Read Links action for the given end) is the following. Associate an object to each component in the context of that end. These components select a sub-collection of all links in the association extent. The selected sub-collection of links identifies a collection of objects at the considered “open” end to read.
- If the considered open end is ordered, the designated collection will be ordered. If the end is unique, the collection is a set obtained by collapsing the duplicate values from the selected tuples; otherwise, it may contain duplicate elements.
- The effect of Create Link and Destroy Link actions is completely analogous to binary associations, except that they specify an N-tuple of object identities instead of pairs.

## Multiplicity of N-ary Association Ends

Because an end of an association with N ends designates a collection of objects of the class at that end for *each* tuple of N-1 existing objects of the classes at the other ends, the multiplicity at the considered end constrains the cardinality of that collection. In particular, you should notice that the multiplicity of an end constrains the cardinality of the collection of objects for each and every possible combination of N-1 existing objects at the other ends.

Consider the examples taken from Figure 10-17, shown again in Figure 10-18 with specified multiplicities of all ends.

For the example in Figure 10-18a, the multiplicity constraints can be deduced and interpreted in the following way:

- The end place** — For each combination of a Course and a Time Slot, there may exist at most one Room in which that Course is scheduled at that time. But it may also not exist, if that Course is not scheduled at that time. Therefore, the multiplicity is 0..1.
- The end class** — For each combination of a Room and a Time Slot, there may exist at most one Course scheduled at that time and place. But it also may not exist, if the Room is free at that time. Therefore, the multiplicity is 0..1.
- The end time** — For each combination of a Room and a Course, there may exist an arbitrary number of Time Slots in which that Course is scheduled in that Room, because the Course may take place many times in the same Room. But it may also not exist, if the given Course simply never takes place in that Room. Therefore, the multiplicity is 0..\*.

Similarly, for the example in Figure 10-18b, the multiplicity constraints can be deduced and interpreted in the following way:

- The end team** — For each combination of a Person and a Role, there may exist at most one Team in which the Person plays that Role. But it may also not exist, if that Person simply does not play that Role. Therefore, the multiplicity is 0..1.
- The end member** — For each combination of a Team and a Role, there must exist exactly one Person that plays that Role in the Team because the rules of the competition require that every Role be covered by a single Person in every Team. Therefore, the multiplicity is 1..1.

## Part III: Concepts

- **The end role** — For each combination of a Team and a Person, there may exist more than one (or arbitrarily many) Roles played by that Person in that Team (for example, the same Person can be the director and the coach of a team) because the rules of the competition allow this. But it may also not exist, if the given Person is simply not a member of that Team. Therefore, the multiplicity is  $0..*$ .

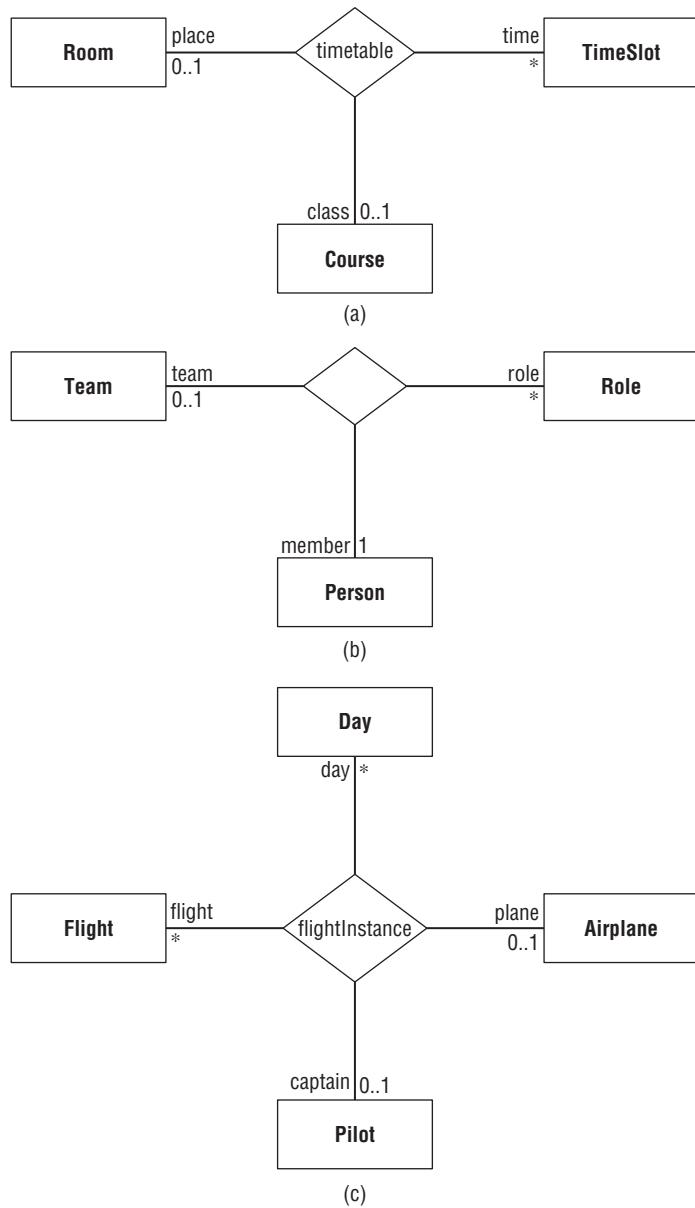


Figure 10-18: Multiplicity of N-ary associations

The same reasoning can be applied to the example in Figure 10-18c to deduce and interpret the multiplicity constraints. This is left to the reader as an exercise.

These illustrative examples indicate some general recommendations on how multiplicity constraints can be derived for an N-ary association end. To derive the lower multiplicity bound, consider an arbitrary combination of objects at the other ends and see whether an object at the considered end must exist for every such combination:

- ❑ The conclusion that it may not exist can be drawn from the fact that arbitrary combination of other objects simply does not exist, makes no sense, or is prohibited. In that case, the lower multiplicity bound of the considered end is zero. Examples are the ends `place` and `time` in Figure 10-18a.
- ❑ The conclusion that it may not exist can be drawn from the fact that it simply does not need to exist. In that case, the lower multiplicity bound of the considered end is again zero. An example is the end `class` in Figure 10-18a.
- ❑ If there must be at least one (or some number) of objects for every such combination, then the lower multiplicity bound is at least one (or some number). An example is the end `member` in Figure 10-18b.

To derive the upper multiplicity bound, consider an arbitrary combination of objects at the other ends and see whether more than one object at the considered end can exist for every such combination:

- ❑ The conclusion that there cannot be more than one such object can be drawn from the fact that the object is uniquely identified by (that is, functionally dependent on) the combination of objects at the other ends. In that case, the upper multiplicity bound is one. An example is the end `place` in Figure 10-18a.
- ❑ If there can be more than one such object, then the upper multiplicity bound is greater than one. Most often, it is simply unlimited (\*).

Note one very important and stringent constraint implied from a lower multiplicity bound greater than zero. For each and every combination of objects at the other ends, there must exist at least one (or more) objects related to them. For the example in Figure 10-18b, for each pair of an existing Team and an existing Role, there must be exactly one Person playing that Role in that Team. Moreover, it is not even possible to create a new object of the class at one of the other ends in an isolated action. Instead, that object must be created within a constrained group of actions that also create new links for all combinations of other existing objects at the other ends and that new object.

For the same example, if a new Team is to be created, new links must be immediately created for that Team and each existing Role. This similarly holds true if a new Role is to be created — a Person playing that role must be specified for every existing Team. Therefore, lower multiplicity bounds greater than one are used only when such restricted behavior is desired, and usually when the set of objects of one of the classes at the other ends is limited, known in advance, and rarely (or never) enhanced. Such is the class `Role` in this example, because it defines a small set of roles the members of a Team can play, while the multiplicity at the end `member` requires that every Role in every Team be covered by a Person (but not necessarily a different one).

In short, most often, the lower multiplicity bounds of N-ary association ends are equal to zero, and the upper multiplicity bounds are equal to either one or unlimited. In other words, the multiplicity bounds of N-ary association ends are most often `0..*` or `0..1`.

### Section Summary

- ❑ The multiplicity of an end of an association with N ends constrains the cardinality of the collection of objects of the class at that end for *each* possible combination (tuple) of N-1 existing objects of the classes at the other ends.

## **Specific Rules for N-ary Association Ends<sup>17</sup>**

Basically, N-ary associations are simple generalizations of binary associations. However, some of their aspects have specific restrictions or implications. In this section, it is assumed that the N-ary association has more than two ends.

An end of an N-ary association is always owned by the association and never by a class. Therefore, it is never a feature (property) of a class. Additionally, the end must be explicitly specified as navigable if the actions are to be performed on that end, because it is never implicitly navigable as an end belonging to a class. By default, every end is navigable.

An end of an N-ary association cannot have the adornment for propagated destruction, or be an aggregate end. Additionally, it cannot be static.<sup>18</sup>

An end of an N-ary association can be read-only, frozen, and unfrozen. Freezing or unfreezing an end of an association with N ends is performed for the specified tuple of N-1 objects at the other ends. If an end is frozen for some tuple of N-1 objects at the other ends, any action that would try to modify the collection of the objects at that end linked to the given tuple of N-1 of objects at the other ends will fail, unless the action is done while all of the N-1 objects at the other ends are being initialized. If the end is read-only, this holds true for any tuple of N-1 objects at the other ends.

An N-ary association or its end can be derived, having the same interpretation as for binary associations. An end of an N-ary association can be constrained to be a subset or a derived union, having the same meaning as for binary associations, except that the collection of objects at the considered end is determined by a tuple of N-1 objects at the other ends.

N-ary associations cannot be specified as acyclic.

The context *c1* of an end of an N-ary association *a1* conforms to the context *c2* of an end of another association *a2*, if and only if the two contexts have the same number of classes (that is, the associations have the same number of ends), and every element of *c1* conforms to the corresponding element of *c2* (that is, every class at an end of *a1* is the same or a subclass of the class at the corresponding end of *a2*). An end *ae1* of an N-ary association *a1* can redefine an end *ae2* of an association *a2* only if the context of *ae1* conforms to the context of *ae2*, and the type of *ae1* conforms to the type of *ae2*. The meaning of the

---

<sup>17</sup>This is advanced material that can be skipped on first reading.

<sup>18</sup>This is a restriction of OOIS UML. It does not exist in standard UML and standard UML does not specify the semantics of this concept.

redefinition is the same as for binary associations — it implies a set of implicit additional constraints applied to the redefined association end in the redefining context.

Note that some of the mentioned concepts that imply constraints on an association end for some or all combinations of objects at the other ends (such as read-only, freezing, subsetting, union, or redefinition) may be significantly more difficult to implement than for binary associations, and may cause excessive computational overhead at run-time, especially when the number of association ends is large. Therefore, the modeler must be especially careful with applying them for N-ary associations. Fortunately, N-ary associations are quite rarely used in practice, and these additional features of their ends are even more rarely needed.

### Section Summary

- ❑ An end of an N-ary association is always owned by the association and never by a class. Therefore, it is never manifested as a feature (property) of a class.
- ❑ An end of an N-ary association must be explicitly specified as navigable if the actions are to be performed on that end, because it is never implicitly navigable as an end belonging to a class.
- ❑ An end of an N-ary association cannot have the adornment for propagated destruction and cannot be aggregate or static.

## ***Actions on N-ary Associations***

Actions on N-ary associations are completely analogous to those on binary actions, except that they work on tuples of N objects (for Create Link and Destroy Link actions) or of N-1 objects (for actions on association ends). Because ends of N-ary associations are never properties of classes, only actions on association ends are available. All other static rules and dynamic semantics are the same.

For example, to create a link of the association `flightInstance` shown in Figure 10-18c, the following action is used:

```
new flightInstance(aFlight,aDay,aPilot,anAirplane);
```

Similarly, to get the Airplane scheduled for a certain Flight on a certain Day, of which the given Pilot is in charge, you could write the following:

```
Airplane ap = flightInstance::plane->read(aFlight,aDay,aPilot);
```

Note that actions on N-ary associations do not provide a way to pose a less determined query, such as to get the Airplane scheduled for a certain Flight on a certain Day, without specifying the Pilot. In general, all other N-1 objects must be specified to read the collection at the given open association end. This is actually the most severe restriction of an N-ary association, at least in the current version of UML. In order to allow such broader queries on related classes, the model must use other language concepts instead of an N-ary association. This issue is addressed in the next section.

### Section Summary

- ❑ Actions on N-ary associations are completely analogous to those on binary actions, except that they work on tuples of N objects (for Create Link and Destroy Link actions) or of N-1 objects (for actions on association ends).

## Conceptual Modeling Issues

Basically, an N-ary association is used to conceptualize the fact that N classes are structurally related by things (that is, links) that connect N-tuples of their objects. However, the same thing can be modeled in UML in a different way, where a new class is introduced in place of the N-ary association, and is related to the other classes by N binary associations. Let's analyze the semantic effects of selecting one or the other alternative conceptual model.

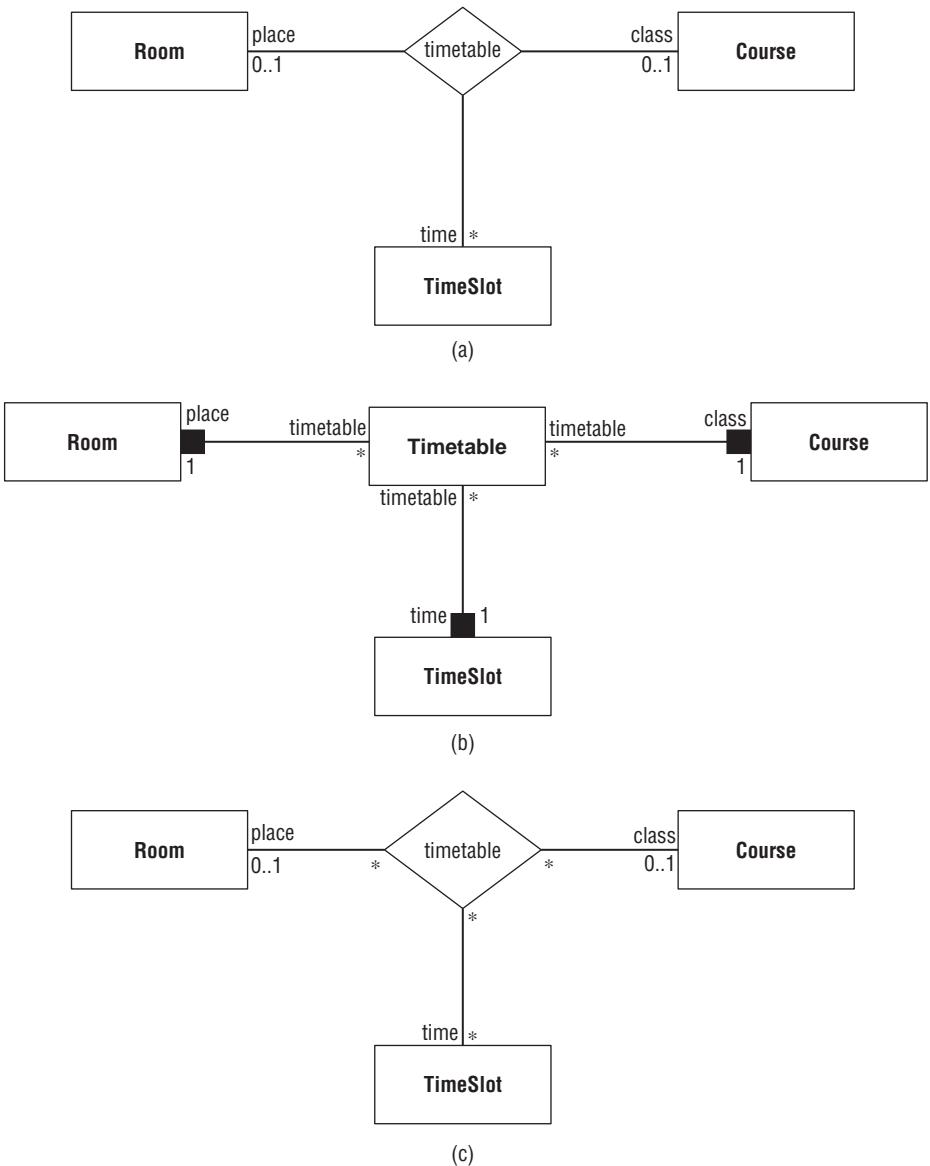
Let's reconsider the example of a school timetable system, whose model with a ternary (N=3) association `timetable` is shown again in Figure 10-19a. The same problem domain is modeled in an alternative way shown in Figure 10-19b, with the class `Timetable` and three binary associations attached to the three related classes. In the object spaces of these two models, an instance (that is, link) of the ternary association `timetable` connecting three objects of `Room`, `TimeSlot`, and `Course` corresponds to an instance (that is, object) of the class `Timetable` with three binary links toward the three objects of `Room`, `TimeSlot`, and `Course`. Here is a brief comparative analysis of the consequences of these two models — that is, of their semantic matches, as well as differences.

The inherent semantic property of the ternary association `timetable` in Figure 10-19a is that its instances (that is, links) have dependent lifetimes. A ternary link of `timetable` is implicitly destroyed whenever any of the three connected objects is destroyed. The same effect is achieved with propagated destruction at the three ends of binary associations in Figure 10-19b. Whenever any of the three objects is destroyed, the destruction is propagated to the linked objects of `Timetable`, and of course, the binary links are implicitly destroyed, too.

A `timetable` link in Figure 10-19a connects exactly three objects of `Room`, `Course`, and `TimeSlot`, and cannot have a “dangling” end. Almost the same is achieved with the multiplicities at the three ends of binary associations in Figure 10-19b set to exactly 1 — a `Timetable` object cannot exist without being attached to exactly three other `Room`, `Course`, and `TimeSlot` objects. (Note, however, that a `Timetable` object can be re-linked by a constrained group of actions to another object, while an N-ary link cannot be “reattached”.)

The first significant difference is in navigation and information retrieval. For an N-ary association, one can only navigate from a tuple of N-1 objects toward the collection of objects at the Nth open end. For example, with the model in Figure 10-19a, you can get the Time Slots allocated to a certain pair of `Course` and `Room` only with `timetable::time->read(aCourse, aRoom)`.

To do the same for the model in Figure 10-19b, you must specify a bit more complicated navigation or query. On the other hand, for the model in Figure 10-19b, you can easily get the collection of, for example, Time Slots in which the given Room is occupied (regardless of the scheduled Course), or the Time Slots in which the given Course is taught, regardless of the place. Such retrievals are not possible for the association in Figure 10-19a.



**Figure 10-19: N-ary association vs. a class with N simple binary associations. (a) A model of a timetable in an educational institution with an N-ary association. (b) A model where the N-ary association is transformed into a class with N binary associations. (c) The model with the N-ary association and specified internal multiplicities.**

In general, an N-ary association provides an easy and direct mapping from a tuple of N-1 objects to a collection of objects at the open end, while all other navigations and mappings are unsupported. On the other hand, if the N-ary association is replaced with an ordinary class and N binary associations, all mappings are possible, although with more indirect and complex navigation.

## Part III: Concepts

---

Of course, the actions that modify the object structure defined by the model in Figure 10-19a or the one in Figure 10-19b are different, but basically, the equivalent effects can be obtained in somewhat different ways.

Another big difference between the two modeling approaches is in the multiplicity constraints. A multiplicity constraint of an N-ary association end (for example, the multiplicity 0..1 at the end `place` in Figure 10-19a) constrains the collections of that end as the open end for every tuple of N-1 objects at the other ends (for example, for all pairs of `Course` and `Time Slot`). This multiplicity is called *external* [Genova, 2002].

On the other hand, a multiplicity at an end attached to the `Timetable` class in Figure 10-19b (for example, the multiplicity \* at the end opposite to the class `Room`) constrains the collection of Timetables to which one object of the other class (`Room`, in this case) can be linked. In the context of the corresponding N-ary association in Figure 10-19a, this would correspond to the number of tuples in which one object of the related class (`Room`, in this case) can appear. This kind of multiplicity is called *internal* [Genova, 2002] and can be specified close to the internal ends of the line segments of the N-ary association, as shown in Figure 10-19c.

Note that external and internal multiplicities are independent, as is clear from Figure 10-19c, and can both be used to specify the cardinality of the tuples in more precise ways. Actually, these are not the only possible multiplicity constraints because a general case of an N-ary association would allow N-1 multiplicities for each association end, one for each mapping of a sub-tuple with  $k$  elements to the rest of  $N-k$  elements of the N-tuple, where  $1 \leq k \leq N-1$ . However, external and internal multiplicities are most useful in practice and almost always sufficient.

Standard UML does not support internal multiplicities. OOS UML allows the specification of internal multiplicities, but just for informative purposes, without any formal semantics or run-time constraints.

Of course, there is one more significant difference. Objects of `Timetable` in Figure 10-19b have their identities, may embody state and behavior, may be accessed by queries, and so on. This is not possible for links of `timetable` in Figure 10-19a, unless the association is an association class (described in the next section).

In short, a model with a simple class with N binary associations (such as that shown in Figure 10-19b) provides better flexibility in a general case. An N-ary association (such as that shown in Figure 10-19a) is used only if it is important to enforce the external multiplicity instead of the internal one, and there is the need to provide only the mapping from a tuple of N-1 objects to a collection of related objects. Also note that the model with a class and binary associations is closer to the relational model, in which only tables with foreign keys referring to other records exist, and there is no separation between a class and an association. In other words, the transformation from the model in Figure 10-19a into the model in Figure 10-19b resembles the mapping of an entity-relationship model with a more abstract concept of N-ary association into a relational model with tables and foreign keys only. The latter is of a lower level of abstraction, but provides better flexibility.

Consequently, N-ary associations are very rarely used in practice; they are used predominantly in the early analysis activities, just to indicate a conceptual connection between N classes, when other semantic aspects are not yet considered, and are usually transformed into binary associations in later design phases.

### Section Summary

- ❑ A model with an N-ary association can be transformed into a model with a new class in place of the N-ary association and N binary associations attached to the other N classes.
- ❑ The effects of such a transformation are manifold. Both models can provide the same outwardly visible behavior in many aspects, but some remain significantly different — most notably, the effects of multiplicity constraints and the possibility of navigation and information retrieval.
- ❑ N-ary associations support only *external* multiplicity constraints, which specify the cardinality of the collections at an open end to which tuples of N-1 objects at the other ends map. They do not support *internal* multiplicity, which would constrain the number of tuples in which an object of the class at one end can participate. A class with N binary associations can support internal, but not external, constraints of the analogue N-ary association.
- ❑ N-ary associations support only the retrieval of the collection of the objects at the open association end to which a tuple of N-1 objects at the other ends maps. A class with N binary associations can support more complex information retrieval and all mappings.

## Association Classes

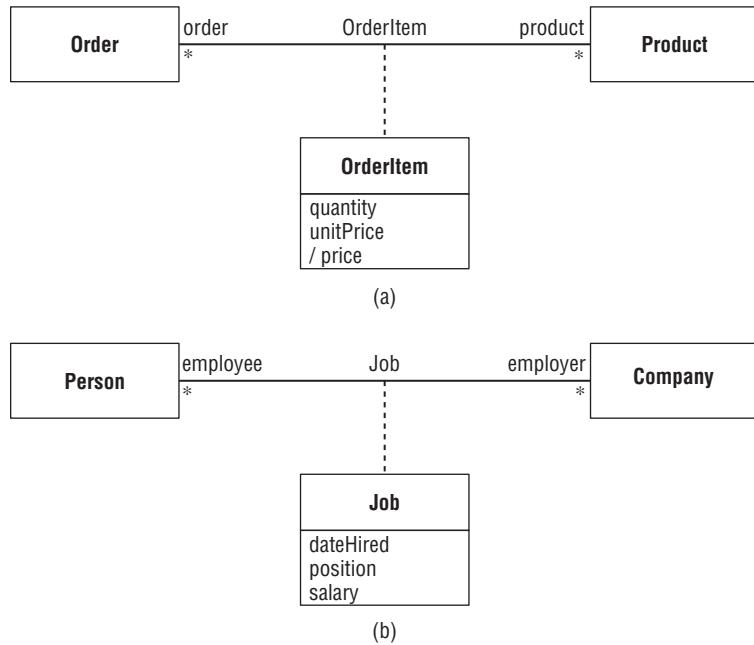
An *association class* is a model element that is a class and an association at the same time. It has all characteristics and semantics of both. It can be seen as an association that also has class characteristics, or as a class that also has association characteristics. It relates a set of classes by defining the collection of links connecting their objects, but also defines a set of features that belong to the relationship itself and not to any of the related classes. Its instances are objects and links at the same time. They have the characteristics of both. Although it basically inherits the rules and semantics of classes and associations, an association class raises several specific issues discussed in this section. Finally, opting between an association class, an N-ary association, or a simple class with binary associations in the conceptual model of a specific domain may often represent a trade-off, as discussed further at the end of this section.

### Notion of Association Class

Let's reconsider an order processing system in which an Order placed by a customer consists of a list of the ordered Products, along with the specification of the ordered quantities, unit prices, and total prices (which are derived as the products of unit prices and quantities). It is easy to conclude that the concepts of Order and Product will be modeled with classes, as shown in Figure 10-20a, but where should the information about the quantity of an ordered Product be placed? The ordered quantity of a Product in the list is not a property of the Order, at least because an Order can have an arbitrary number of Products. Neither is it a property of the Product, at least because the product can be an element of an arbitrary number of Orders. Instead, the information about the quantity of the ordered Product is a property of the connection of the Order and the Product.

## Part III: Concepts

---



**Figure 10-20: Examples of association classes.** (a) An excerpt from the conceptual model of an order processing system. (b) The conceptual model for recording the current employment of Persons in Companies.

For such purposes, UML recognizes the notion of *association class*. An association class is an association that is also a class, which thus means that the association has the class's features (properties and operations). It relates a set of classes by defining links that connect their objects, but also defines a set of features that belong to the relationship itself, and not to any of the related classes.

The instances of association classes are called *link objects*. They are links as well as objects, and have the semantics of both. They have identity, may have attribute values, and may be linked with other objects, but also behave as ordinary links. For example, a link object is implicitly destroyed whenever any of the objects it connects is destroyed.

For the example in Figure 10-20a, the association class OrderItem is an association relating the classes Order and Product in a many-to-many fashion. However, it is also a class, having the attributes such as quantity, unitPrice of the Product recorded at the time of placing the Order, and the derived attribute price. These attributes are features of the association itself, and not of any of the related classes. Therefore, every link between an Order and a Product carries the information about the quantity and price of that very Product ordered within that very Order.

Another example is shown in Figure 10-20b. It presents the conceptual model for recording the current employment of Persons in Companies. A Person can be currently employed in an arbitrary number of Companies. However, each of these connections has its own properties: the position, starting date, and the salary for the job. Note that these properties conceptually do not belong to any of the related classes, but to the relationship itself.

As a language concept, an association class inherits all language characteristics of associations and classes. However, these characteristics are inherited without duplication. In particular, associations and classes are named elements, having their names in the model. But an association class has a single name, not two different names when regarded as a class or as an association. The same holds for the visibility. In addition, there is a natural restriction that an association class cannot have itself (being a class) at any of its ends (being an association at the same time).

Graphically, an association class is represented by a class symbol attached to the association path by a dashed line, as shown in Figure 10-20. The association path and the class symbol represent the same underlying model element, although they are graphically distinct. That model element has a single name; that single name may be placed on the path, in the class symbol, or on both. The class symbol of the association class can be dragged away from the association path, but the dashed line must remain attached to both the path and the class symbol.

### Section Summary

- ❑ An *association class* is a model element that has both association and class characteristics. It relates a set of classes by defining the collection of links connecting their objects, but also defines a set of features that belong to the relationship itself, and not to any of the related classes.
- ❑ Instances of association classes are called *link objects*. They have the semantics of both links and objects.
- ❑ An association class is shown as a class symbol attached to the association path by a dashed line.

## Uniqueness of Association Classes<sup>19</sup>

All observations given thus far for associations hold true for an association class, too. However, a crucial difference between an instance of a pure association and an instance of an association class is that the former is a pure link without identity, while the latter is a link and an object, having its identity as an inherent characteristic.<sup>20</sup> This fact opens an issue specific for association classes only, the result of which is the need for the uniqueness modifier of the very association class, independently of the uniqueness of its ends. This section discusses association class uniqueness using the examples shown in Figure 10-21.

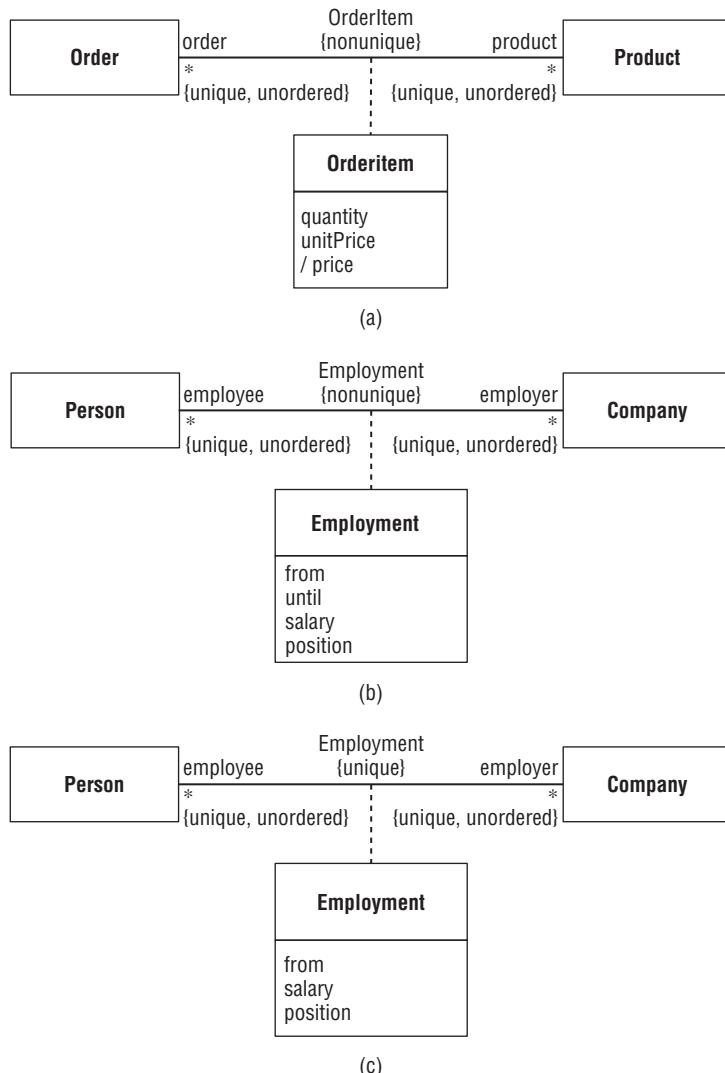
Figure 10-21a shows the conceptual model of an order processing system, with uniqueness modifiers explicitly shown in the diagram. The association end *product* is designated as unique because the intention of the modeler is that it results in the set of (distinct) Products appearing in a certain Order. Similarly, the unique association end *order* gives the set of (distinct) Orders in which the given Product has been

<sup>19</sup>The concept of association class uniqueness does not exist in standard UML; it is specific to OOIS UML, first proposed in [Milicev, 2007].

<sup>20</sup>It could be noted that pure links do not need to have identity, even when there are multiple links relating the same set of objects. The semantics of the association extent, described as a bag of links, and the actions on it successfully handle multiple occurrences of links without requiring their identity. The actions simply add or remove (that is, count) occurrences of links to or from the bag.

## Part III: Concepts

ordered. If any of these sets should be limited in size, the multiplicity constraint could specify that. However, the association class itself may allow duplicate links (as pairs of linked objects), each link being also a separate object with its identity, allowing a Product to appear several times in the same Order, each time as a separate object of OrderItem. Every such object carries important information (such as quantity and price). Note that these objects must have their identity and internal state, because the system should modify that state of a particular object of interest (for example, when setting the quantity).



**Figure 10-21:** Examples of association classes with different and independent uniqueness of the association and its ends. (a) An excerpt from the conceptual model of an order processing system. (b) The conceptual model for employment history tracking. (c) The conceptual model for recording the current employment of Persons in Companies.

A similar case occurs with the model for employment history tracking in Figure 10-21b. Each instance of the `Employment` association class represents the fact that the given `Person` used to be or is still employed in the given `Company`, providing other necessary information about that employment (such as period, salary, and position in the `Company`). Because the same `Person` can be re-employed in the same `Company` several times during different periods, this association class should allow multiple links (as pairs of linked objects). Of course, each of these links would be a separate object with its identity and its own properties that carry the information about the given period of employment. The unique association end `employer` results in the set of distinct `Companies` in which the given `Person` has ever been employed. This similarly holds true for the opposite end.

An interesting opposite circumstance arises when almost the same model conceptualizes a different problem domain. The association class `Employment` in the model shown in Figure 10-21c conceptualizes the current (instead of past) employment of a `Person` in a `Company`, carrying again the additional information about that employment. However, unlike in the previous case, the association class should not allow duplicate links. There should not be two different link objects of `Employment` recording the employment of the same `Person` in the same `Company` at the present time because they would cause clashing of information about that inherently unique employment. This is why the very association class is designated as *unique*, unlike in the previous cases.

To summarize: An association class can be specified as unique or non-unique, completely independently of the uniqueness of its ends. If it is unique, the association class should not allow duplicate links. Otherwise, its semantics are as described for pure associations.

The uniqueness of the association class is an implicit constraint attached to the association class. It is checked at run-time on creation of a link object. If another link object of the same association class already exists and links the same tuple of objects, an exception is raised. Put another way, the tuple of identities of the linked objects constitutes an implicit unique identifier of the unique association class. Note that if an association class is unique, all its ends will always designate inherently unique collections, although they may be specified as non-unique.

### Section Summary

- An association class can be specified as unique or non-unique, completely independently of the uniqueness of its ends.
- Uniqueness of an association class is an implicit constraint attached to the association class. It is checked at run-time on creation of a link object. If another instance of the same association class that links the same tuple of objects already exists, an exception will be raised.

## Actions on Association Classes

A link object is created using the Create Link Object action, not using the Create Object or Create Link action. The Create Link Object action has the same arguments as the Create Link action for the association class (as an association), but returns a reference to the created link object (as an object). For the example

## Part III: Concepts

---

in Figure 10-21a, a link object of `OrderItem` can be created in the OOIS UML native detail-level language with the following:

```
OrderItem oi = new OrderItem(anOrder, aProduct);
```

When an end of a binary association class belongs to a class, then every action on its slot that adds a value to the designated collection actually creates a link object instead of a pure link.

When a link object is created, the default constructor of the association class (being a class) is always invoked. Any other constructor is never implicitly invoked on object creation. (It can be explicitly invoked as an ordinary operation, however.) Therefore, all other constructors for association classes have no particular meaning, although they can be defined. Actually, an instance of an association class can never be created using a Create Object action with optionally specified arguments of a constructor. If such an action is specified statically, a compilation error is reported. If it is invoked dynamically through the reflection mechanism, an exception of type `CreateLinkObjectException` is raised. For example, these actions are illegal:

```
OrderItem oi1 = new OrderItem; // incorrect
OrderItem oi2 = new OrderItem(aQuantity); // incorrect
```

The same holds true for every class that is a specialization of an association class because its instances are also link objects. Note that if an association class is abstract, its instance can be only a direct instance of one of its concrete specializations. The arguments of such a Create Link Object action are the same — they refer to the objects to be linked by the new link object. For that reason, OOIS UML does not allow a class to be directly or indirectly specialized from more than one association class.

A link object can be destroyed either as a link or as an object. Whatever the approach taken, the destruction has the composed semantics of destroying a link and an object. If it is destroyed by a Destroy Link action, its default destructor is invoked.

For a given tuple of objects of the classes attached to the association class's ends, the action `read` on the given association class returns the collection of link objects that connect the objects in the tuple. For example, the following code reads the link object for the association class in Figure 10-21c:

```
Employment e = Employment->read(aPerson, aCompany);
```

If the association class is unique, the multiplicity of the result of this action is 0..1. If the association class is non-unique, the multiplicity is \*. The result, if it is a collection, is always unique, because every link object is a distinct entity. For the example in Figure 10-21a, the following code in the OOIS UML native detail-level language sums up the quantity of the given Product ordered in the same given Order:

```
OrderItem[*] oiCol = OrderItem->read(anOrder, aProduct);
Real q = 0.0;
oiCol->forEach(oi) {
 q = q + oi.quantity;
}
```

For a binary association class whose association end belongs to the opposite class, the collection of link objects attached to the given object can be obtained by addressing the pseudo-property of the object,

named the same as the association class, but starting with a lowercase character. For the example in Figure 10-21a, the following action returns the collection of all link objects of `OrderEntry` that are attached to the given `Order`:

```
anOrder.orderItem->read()
```

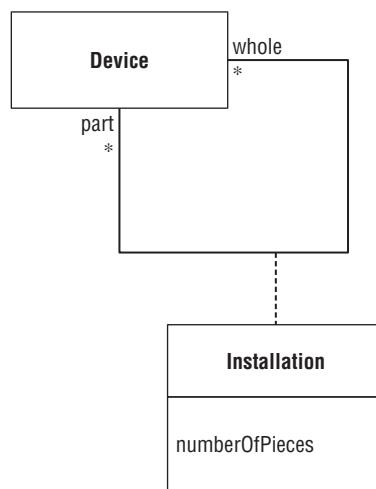
The pseudo-property (`orderItem`, in this case) is a joint manifestation of the association class and the association end opposite from the object's class (the association end `product`, in this case). That association end will be referred to as the “navigated end.”

In case of a reflective association class, such reference to the association class pseudo-property would be ambiguous, because the navigated end is undetermined. Therefore, the navigated association end should be explicitly specified within square brackets. For the example shown in Figure 10-22, an object of the class `Device` has both `part` and `whole` properties, so the specification `aDevice.installation` is ambiguous. The following specification reads the collection of link objects of `Installation` that link the parts of a `Device`:

```
aDevice.installation[part]->read()
```

The following specification reads the collection of link objects of `Installation` that link the wholes of a `Device`:

```
aDevice.installation[whole]->read()
```



**Figure 10-22: Example of a reflective association class**

Such a fully qualified navigation to link objects by specifying both the association class and the navigated association end can be used for non-reflective associations, too.

For such an action to be applicable, the navigated association end (be it explicitly specified within square brackets or implicitly assumed) must be accessible and navigable. In addition, the name of the association

## Part III: Concepts

---

class must be unambiguously resolved within the namespace of the object's class. That is, no true property should have the same name as the association class (just starting with a lowercase character).

The described Read Link Object action returns the collection of link objects linked to the given object in the direction of the navigated association end. This collection is always unique because every link object is a distinct entity. The multiplicity of such collection depends on the uniqueness of the association class and multiplicity and uniqueness of the navigated association end. If the association class is unique or the navigated end is non-unique, then the multiplicity of this collection is equal to the multiplicity of the navigated end; otherwise, its upper bound is unlimited. If the navigated end is ordered, and if the association class is unique or the navigated end is non-unique, the resulting collection is ordered; otherwise, it is unordered.

Note that the described Read Link Object action returns the collection of *all* link objects linked to the given object in the direction of the navigated association end. Typically, this collection may be bigger than (or of the same size as) the collection of objects obtained by a Read Link action for that end. More precisely, if the association class is unique or the navigated end is non-unique, the collections are of the same size because the extent of the association class cannot contain any link object more than the collection returned by the Read Link action for the navigated end. Otherwise, if the association class is non-unique and the navigated end is unique, then the Read Link action for the navigated end may return a collection that is obtained by a projection of the collection of link objects, and thus be of a smaller size.

Of course, a link object, being an object, can be accessed in all other ways applicable to objects (for example, using queries).

Starting from the given link object, the navigation toward the object at one of the link ends is straightforward. It is referred to by specifying the association end as a pseudo-property of the link object. For the example in Figure 10-22, given a link object `anInstallation` of `Installation`, the object connected at the end `whole` is referred to by `anInstallation.whole`. Note that the multiplicity of this expression is always exactly one because a link object has always exactly one object attached at each of its ends. The name of the referred association end must be unambiguously resolved within the namespace of the association class.

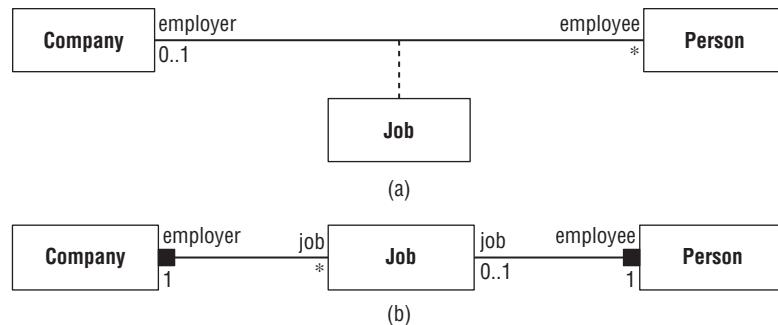
### Section Summary

- ❑ A link object is created using the Create Link Object action.
- ❑ A link object is destroyed as an ordinary object or an ordinary link.
- ❑ For the given tuple of objects, the action `read` on the given association class returns the collection of link objects that connect the objects in the tuple.
- ❑ The collection of link objects that are attached to the given object can be obtained by the Read Link Object action.
- ❑ For reflective association classes, the navigated association end must be specified within square brackets (also applicable to non-reflective association classes).
- ❑ Starting from a link object, the object (always exactly one) connected at an end can be accessed by referring to a pseudo-property named as that end.

## Conceptual Modeling Issues

Being an association, an N-ary association class can also be transformed into a class with N binary associations with already described consequences. For an association class, however, the alternative models are closer in terms of identity because in both cases the entity that is relating other objects is an object (a link object or a pure object).

For binary association classes (which are most common in practice), the transformation preserves even more semantic aspects. An example is shown in Figure 10-23. As opposed to N-ary associations with  $N > 2$ , both alternative models provide the same information-retrieval capabilities, although with slightly different navigation. For example, to address all Persons employed in a Company, you must write `aCompany.employee` for the model in Figure 10-23a, and a bit more complex navigation (using `forEach`) or query with one more hop over the class `Job` for the model in Figure 10-23b. To get the Job of a Person, the navigation is the same in both cases: `aPerson.job`.



**Figure 10-23: Association class versus a class with two simple binary associations. (a) A model of employment tracking system with an association class. (b) A model where the association class is transformed into a class with two binary associations.**

It is interesting to note that, for binary associations (as opposed to N-ary associations with  $N > 2$ ), the external and internal multiplicities are equal. Really, the external multiplicity \* at the end `employee` in Figure 10-23a specifies the cardinality of collections to which a `Company` maps, and is equal to the internal multiplicity at the end `job` across from the class `Company` in Figure 10-23b. The same holds for the opposite end in Figure 10-23a. Therefore, the transformation of a binary association class into a class with two binary associations preserves the semantics of multiplicities, too.

In short, a binary association class can be adequately transformed into a pure class with two binary associations attached to the related classes. The transformation preserves all basic semantics, but its only drawback is that the navigation from one to the other end is more complicated.

When the upper multiplicity bound at one end of a binary association is 1, it can even be transformed into a pure association under certain conditions, while its features are moved to the class at the end across from the end with the upper multiplicity bound equal to 1. For the example in Figure 10-23a, the association class `Job` can be transformed into a pure association with its features moved to the class `Person`. Such transformation may sometimes reduce the complexity of the implementation of the application and its GUI.

### Section Summary

- ❑ A binary association class (which is most common in practice) can be adequately transformed into a pure class with two binary associations toward the related classes. The transformation preserves all basic semantics, including multiplicity, but its only drawback is that the navigation from one to the other end is more complicated.

# 11

## Constraints

Constraints are conditions or restrictions expressed in a natural or in a machine-readable language that extend the basic semantics of the model implied from the definition of the modeling language. In OOIS UML, a constraint specified in a formal language has formal run-time semantics. It represents a condition that must hold during certain intervals of execution. Such a constraint can be specified in different languages, including the Object Constraint Language, a standard language that is part of UML.

### Constraints as Model Elements or as Objects

In standard UML, a constraint is a condition or restriction expressed in a natural or in a machine-readable language for the purpose of declaring some of the semantics of the system beyond what is imposed by the modeling language. A constraint can be specified in a natural language and can be attached to an arbitrary number of model elements. However, such constraints have no formal semantics and cannot have run-time effects. In executable models, OOIS UML deals only with constraints that have strictly formal semantics and, thus, may have run-time effects.

In OOIS UML, a constraint can be defined within the model, as a model element attached to other model elements. In that case, the constraint is checked implicitly by the execution environment at certain moments in run-time. Such constraints ensure a permanent consistency of the object space, as specified by the modeler. On the other hand, the OOIS UML model library allows constraints to be created as ordinary objects within the application's object space. Such constraints represent conditions that are not mandatory rules for the system's consistency, but simple queries about some properties of the object space checked at run-time on explicit interactive or programmatic requests.

### Constraints as Model Elements in Standard UML

A *constraint* is a model element that specifies additional semantics for one or more elements to which it is attached, beyond the basic semantics defined in the modeling language. For example, the conceptual model for the Easylearn school example from Chapter 5, shown in Figure 11-1, includes a constraint attached to the class `Person`, which implements the requirement that an outsourced Course cannot be attended by a Person without a non-empty credit card number.

## Part III: Concepts

---

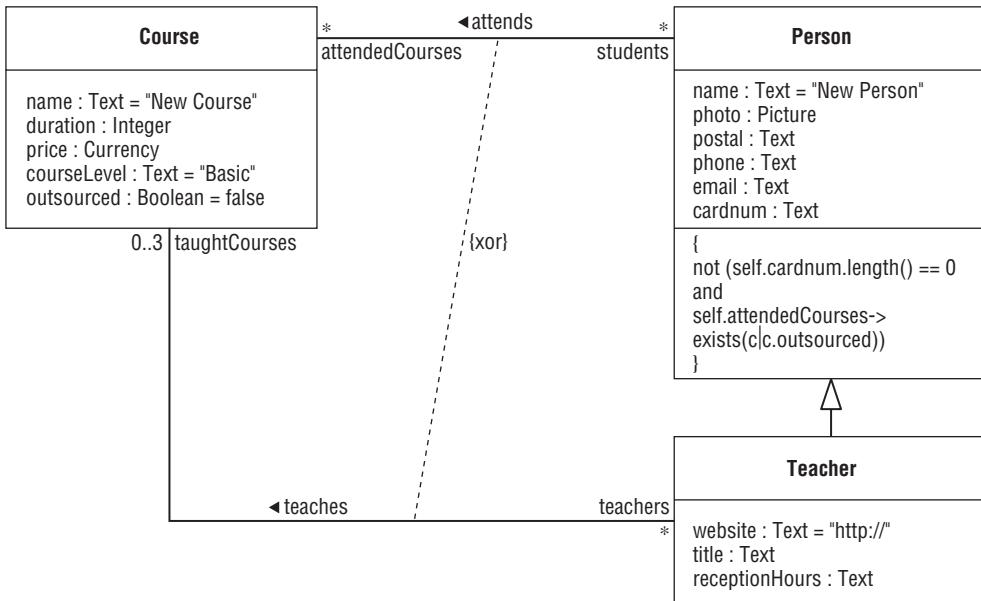


Figure 11-1: Constraints in the Easylearn school conceptual model

Certain kinds of constraints are predefined in UML, while others may be user-defined. One predefined constraint is `xor`, which can be attached to two associations, as shown in Figure 11-1. It specifies that the two constrained associations, representing collections of tuples at run-time, do not intersect (that is, do not have common tuples). For the example in Figure 11-1, the `xor` constraint meets the requirement that a Teacher cannot attend a Course he or she teaches.

A user-defined constraint is expressed using a selected language, whose syntax and interpretation are the responsibility of the modeling tool. One predefined language for writing constraints is the *Object Constraint Language* (OCL). A detail-level programming language may also be appropriate for expressing constraints in some cases.

A constraint is a packageable element. Consequently, it is also a named element, so it may contain an optional name, although constraints are usually anonymous. As a named element, a constraint may be owned by a namespace. However, it does not necessarily apply to the owner namespace itself, but may also apply to the elements in that or other namespaces.

The optional namespace of a constraint also represents its *context*. It is used as the namespace (scope) for interpreting names used in the specification of the constraint. For example, if the context of the constraint is a class, as in Figure 11-1, the name `self` in OCL is used to refer to the object of that class to which the constraint is applied at run-time, while the other names are interpreted in the namespace of that class.

In standard UML, constraints are defined very loosely to allow flexible modeling, especially in the early analysis phases of development, when it is important to quickly and roughly specify the requirements or sketch the design.

First, a constraint can be specified in any available language, even in a natural spoken language or a semi-formal language.

Second, a constraint can be attached to an arbitrary number of model elements of any kind, except to itself. These relationships between the constraint and the constrained elements have only an informative meaning for the reader, indicating that the constraint refers to and constrains those elements in some way. These relationships can be explicitly or implicitly shown in a diagram, as is the case in Figure 11-1. It is the responsibility of the tool to provide a means to determine which elements are involved in the constraint and how. These relationships do not (and often cannot) represent a complete specification of all elements addressed in the constraint and required to evaluate the constraint's specification, however. They do not have any run-time manifestation or other formal semantics.

Third, standard UML does not define the circumstances or events that trigger the evaluation of constraints at run-time. Additionally, it does not define for which instance (or instances) of the constraint's context or other model elements a constraint is evaluated (that is, to which instance will `self` refer). For the example in Figure 11-1, the intention of the modeler was to prevent a Person from attending an outsourced Course without providing the credit card number. But what happens if the attribute `outsourced` of a Course is set to `true` while there are students linked to that Course without credit card numbers? Wouldn't it be extremely ambitious to expect that the tool automatically infer that the constraint attached to the class `Person` should be checked on such an action applied to an attribute of `Course`, and to do that for all objects of `Person` that are linked to the modified object of `Course`?

Even if it sounds reasonable for this simple case, what about much more complicated constraints that analyze a large piece of object space and traverse many objects even in a recursive manner (by invoking recursive and possibly polymorphic operations)? Or else, wouldn't it be extremely ineffective to check every such constraint for all instances on every such action? Obviously, in order to make constraints tractable and executable, some restrictions must be introduced, and there must be some intervention of the modeler to resolve such open issues.

Finally, the run-time effects of a violated constraint are not defined in standard UML, either.

As a result of such an approach, constraints in standard UML do not have formal semantics and are not directly manifested at run-time. OOIS UML models are, on the other hand, fully formal, and have executable semantics. This is why OOIS UML restricts the use of constraints in many aspects to make them automatically interpretable, and provides complete run-time semantics of constraints.

Consequently, such flexible and informal constraints of standard UML can be used in the early analysis phases of development, when it is important to quickly and roughly capture the user requirements or sketch out the design. The analysis model with such constraints is thus informal or incomplete. In the later phases of development, the analysis model developed in standard UML is transformed into a design model, which complies with the OOIS UML profile rules. The design model is then completely formal and can be directly turned into an executable application.

### Section Summary

- ❑ A *constraint* is a model element that specifies additional semantics for one or more elements to which it is attached, beyond the basic semantics defined in the modeling language.

*Continued*

- ❑ Certain kinds of constraints are predefined in UML, while others are user-defined.
- ❑ A user-defined constraint is described using a particular language, whose syntax and interpretation is the responsibility of the modeling tool. One predefined language for writing constraints is OCL. Other detail-level languages can be used, too.
- ❑ A constraint is a packageable element. It may contain an optional name, although constraints are usually anonymous.
- ❑ The optional namespace of a constraint also represents its *context*. It is used as the namespace (scope) for interpreting names used in the specification of the constraint.
- ❑ In standard UML, constraints are very loosely defined to allow flexible modeling and lack formal semantics in many aspects — most notably, in what triggers their execution. The constraints of standard UML can be used in the early analysis phases of development, when it is important to quickly and roughly capture user requirements or sketch out the design.

### Notation

A constraint is shown as a text string enclosed within curly braces {}, optionally starting with its name followed by a colon, as shown in the following example:

```
{outsourcedCourseExists: self.attendedCourses->exists(c|c.outsourced))}
```

When a constraint applies to an element that is shown as a text string (for example, attribute, operation, and so on), the constraint's string in braces may follow the element's string. When a constraint applies to one element that is depicted as a symbol (for example, class, association), the constraint's string may be placed near the element's symbol, preferably near its name. When a constraint applies to two elements shown as symbols (such as two classes or two associations), the constraint may be shown as a dashed line connecting the elements, labeled by the constraint string in braces. Figure 11-1 shows an {xor} constraint between two associations. For three or more paths of the same kind (such as generalization or association paths), the constraint applied to all of them may be attached to a dashed line crossing all of the paths.

Alternatively, the constraint string may be placed in a note symbol attached to each of the symbols for the constrained elements by a dashed line.

### Section Summary

- ❑ A constraint is shown as a text string in curly braces {}, optionally starting with its name followed by a colon.

## Constraints as Model Elements in OOIS UML

In OOIS UML, constraints have formal meaning and run-time effects in the modeled system. This is why some modeling restrictions apply to such constraints. This section discusses the modeling aspects and the runtime semantics of constraints in OOIS UML.

### Modeling Aspects

In order to be formal and executable, constraints in OOIS UML must be specified in a formal language — that is, in OCL or any other formal detail-level language provided by the implementation. The specified constraint must evaluate to a Boolean value. Additionally, it must not have side effects. That is, the constraint must not execute any action that modifies the object space, even within invoked operations. In particular, the specification of a constraint may encompass only the following kinds of elements:

- OCL expressions, because OCL does not define actions that modify the object space.
- Actions that only read the object space.
- Query operations invoked within specifications in OCL or other languages that have no side effects. The methods of such operations must not incorporate actions that modify the object space, and may invoke only query operations.

In OOIS UML, an explicit constraint can be attached to exactly one model element and only to certain kinds of elements.<sup>1</sup> The constrained element determines the context of the constraint, as well other important aspects that affect its run-time manifestation. The following table lists all valid kinds of constrained elements, the purpose of a constraint attached to that kind of element, the context of such a constraint, and the instance to which `self` or a similar reserved name refers when the constraint is evaluated at run-time.

| Constrained Element                                                     | Purpose                           | Context                        | <code>self</code> Refers to |
|-------------------------------------------------------------------------|-----------------------------------|--------------------------------|-----------------------------|
| Class or data type $x$                                                  | Invariant of $x$                  | Classifier $x$                 | Instance of $x$             |
| Property of class or data type $x$ (attribute or owned association end) | Constraint on the property        | Classifier $x$                 | Instance of $x$             |
| Operation of class or data type $x$                                     | Precondition or postcondition     | The operation                  | Instance of $x$             |
| Association                                                             | Constraint on the association     | Package owning the association | N/A                         |
| Association end owned by association                                    | Constraint on the association end | Package owning the association | N/A                         |

A constraint attached to a class or data type represents an *invariant* of that classifier — that is, a rule that must hold for every instance of that classifier at any time. The context of such a constraint is the constrained classifier, and `self` refers to an instance of that classifier for which the constraint is being evaluated.

<sup>1</sup>An exception is the predefined constraint `{unique}` that can be attached to one or more attributes of the same class. Some implicit constraints (such as composition, subsetting, and union of association ends) apply to two or even more elements. This discussion, however, focuses on explicit, user-defined constraints.

## Part III: Concepts

---

When a constraint is attached to a property of a class or data type (that is, to an attribute or association end owned by a class), the context of the constraint is the owner classifier, and `self` refers to the instance of that classifier that owns the property on which the constraint was triggered.

Operations allow special kinds of constraints defined as *preconditions* and *postconditions*. They will be described in Chapter 13. The context of such a constraint is the operation, and `self` refers to the instance of that classifier of which the operation was invoked.

If a constraint is attached to an association or an association end owned by its association, the context is the package owning the constrained association, and `self` (or a similar reserved name) cannot be used in the specification of the constraint.

The list of allowable kinds of constrained elements may be enhanced in the future versions of OOIS UML.

### Section Summary

- ❑ To be formal and executable, a constraint in OOIS UML must:
  - ❑ Be specified in OCL or another formal detail-level language
  - ❑ Have a Boolean result
  - ❑ Not have side effects
  - ❑ Be attached to exactly one class, data type, property, operation, association, or association end

## Run-Time Semantics

The evaluation of a constraint is triggered by every action that modifies the element to which the constraint is attached. In other words, the execution of any action that affects the constrained element is an implicit trigger for the evaluation of the attached constraints.

The following table summarizes the actions that trigger the constraints attached to the allowed kinds of constrained elements. It also specifies to which instance at run-time the constraint will be applied (this will be the instance referred to by `self`).

| Constrained Element               | Triggers                                                                      | Applied to                                             |
|-----------------------------------|-------------------------------------------------------------------------------|--------------------------------------------------------|
| Class or data type <code>x</code> | All actions that create an instance of that classifier.                       | The instance that is created.                          |
|                                   | All actions that modify a slot of an instance of that classifier.             | The instance owning the modified slot.                 |
|                                   | All actions that create or destroy links with an instance of that classifier. | The instance involved in link creation or destruction. |

| Constrained Element                                                                                                                               | Triggers                                                      | Applied to                                |
|---------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------|-------------------------------------------|
| Property of class or data type x (attribute or owned association end), including multiplicity and other implicit constraints attached to property | All actions that modify a slot of the property.               | The instance owning the modified slot.    |
| Operation of classifier x                                                                                                                         | Operation invocation.                                         | The instance whose operation was invoked. |
| Association                                                                                                                                       | All actions that create or destroy links of that association. | N/A                                       |
| Association end owned by association, including multiplicity and other implicit constraints attached to association end                           | All actions that create or destroy links of that association. | N/A                                       |
| Multiplicity of variable                                                                                                                          | All actions that modify the variable.                         | N/A                                       |

For example, an invariant of a class is evaluated for every created object of that class, immediately after its creation has been completed (that is, its constructor has finished). Additionally, the invariant is also triggered on every action that modifies a slot of an object and is checked for the affected object. Note that it is not evaluated for other objects that are not affected by the executed action.

Similarly, a constraint attached to a property is triggered only by the actions that modify a slot of that property and is checked only for the instance owning that slot, and not for the others. Multiplicities of association ends (as implicit constraints on association ends) are evaluated for the object at each end of a created or destroyed link.

An implementation may further optimize the set of actions that trigger certain constraints, but only in a way that preserves the described semantics. For example, an implementation may analyze the specification of an invariant and infer that the actions that modify certain attributes cannot affect the validity of the invariant, so that the run-time environment need not evaluate that invariant on every Write Attribute action. Similarly, an implementation may infer that a constraint cannot be affected by a Destroy Link action, but only by a Create Link action, and may trigger the constraint appropriately.

If an evaluated constraint is violated (that is, it returns `false`), the action that has triggered its evaluation fails and raises an exception of type `ConstraintFailedException` or one of its subtypes (for example, for multiplicity violations). In that case, the failed action is treated as a fault in the application with the consequences described in Chapter 13.

A set of executed actions can be grouped into a *constrained action group*, which affects the evaluation of the constraints triggered by those actions. Instead of evaluating the constraints immediately after each executed action, the evaluation of all these constraints is deferred to the moment of the completion of the entire group. After the entire group is completed, all the constraints triggered by the actions executed within the group are evaluated and possibly fail. This way, several actions can transform the object space in a consistent manner as an atomic transaction. For example, several actions can destroy some links and

## Part III: Concepts

---

create some others, while the multiplicity of the affected association ends is preserved by the entire action group, and not by each separate action.

In the OOS UML native detail-level language, a constrained group of actions is a block of statements enclosed in curly braces, starting with the keyword `group`:

```
group {
 ... // Statements that include actions
}
```

In another programming language used as the detail-level language, a constrained action group may be supported by the class `ActionGroup`. In order to start a constrained group of actions, an instance of that class should be created. To close the group, the operation `close` of that instance should be invoked. For example, in Java or C#, this may look like this:

```
ActionGroup ag = new ActionGroup();
 ... // Statements that include actions
ag.close();
```

Constrained action groups can be nested, both statically (that is, written one within the other) and dynamically (a new one opened before the other is closed, as directed by the control flow). The evaluation of all constraints triggered within a group is deferred until the outermost enclosing group is completed (that is, the one that has no groups enclosing it in the dynamic order of nesting). Following is an example:

```
group { // Outer group
 ...
 group { // Inner group
 ...
 } // The triggered constraints are not evaluated here,
 ...
} // but here, if this is the outermost group
```

At the completion of the outermost constrained action group, the deferred constraints (including multiplicities of attributes, association ends, and variables, as well as other implicit constraints) are evaluated only for the variables, data type instances referred to by variables, and objects that survive the entire group. Constraints related to the objects that have been destroyed within the group (along with their slots and links), as well as to the variables that have gone out of scope (along with the data type instances referred to by them and their attributes) are not evaluated.

Note that constrained action groups also optimize the performance of the system with respect to evaluation of constraints (which may take considerable computation resources and time). If several actions within a constrained group trigger the same constraint for the same element of the object space, the constraint will be evaluated only once for that element at the completion of the entire group, instead of many times if the actions were executed individually (that is, outside the group).

As an overall result, a constraint holds at *certain points in time*, or better said, during certain *intervals* at run-time. In other words, there are some intervals, during which a constraint may not hold. One example is the very interval of the action execution. Since the execution of an action requires some finite time in reality, during some instants in that time a constraint may not hold because the object space is in a transitional state. Another case is the execution of a constrained group of actions, which is also a kind of a transition of the system's object space between two steady states. However, the default semantics of

actions and constraints guarantees that the constraints hold *before* and *after* the execution of an action or a constrained action group (that is, in the steady states of the object space).

### Section Summary

- ❑ The evaluation of a constraint is triggered by any action that modifies the element to which the constraint is attached.
- ❑ If an evaluated constraint fails, it raises an exception.
- ❑ A set of executed actions can be grouped into a *constrained action group*. Instead of evaluating the triggered constraints immediately after each executed action, the evaluation of all these constraints is deferred to the moment of the completion of the entire group.
- ❑ Constrained action groups can be nested (both statically and dynamically). The evaluation of the triggered constraints is deferred to the completion of the outermost dynamically nested group.

## Constrained Action Groups and Conformance Rules<sup>2</sup>

Constrained action groups affect the sense of conformance rules defined so far. Here is an illustrative example. Let the class of the following method have a property `agents` of type `Agent` with a multiplicity 3..3. The purpose of the method shown here is to filter the candidate Agents provided in an unlimited collection `candidates` to the set of acceptable ones.

```
Agents[*] candidates = ...;
group {
 this.agents = candidates; // Conformance violation!
 candidates->forEach (ag)
 if (!this.isAcceptable(ag))
 this.agents->remove(ag);
}
```

The method first stores the given collection to the slot `agents`, and then removes those that are not acceptable. Because the first step could violate the multiplicity constraint, the entire activity is enclosed in a constrained action group. If the slot ends in having exactly three elements after the entire group is executed, the activity completes normally; otherwise, it raises an exception.

Note, however, that the first line within the group is a Set Property action that violates the conformance rule because of incompatible multiplicities of the property `agents` and the variable `candidates` to whose value it is set. Similarly, this holds true for the same activity rewritten in another way:

```
Agents[*] candidates = ...;
group {
 this.agents->clear(); // Static multiplicity rules violation!
 candidates->forEach (ag)
```

<sup>2</sup>This is advanced material that can be skipped on first reading.

## Part III: Concepts

---

```
if (this.isAcceptable(ag) && this.agents->size()<3)
 this.agents->add(ag);
}
```

Here, the action `clear` on the property `agents` violates the static rule for that action because the property does not allow zero cardinality. However, the multiplicity constraint will not be violated at the end of the group, in the regular, expected, and intended case.

This example indicates a general problem. The conformance rule, in particular, the multiplicity conformance rule that is checked statically for the actions upon properties, variables, and arguments initialization, does not make sense when the actions are executed within constrained action groups. The same holds true for the static rules related to multiplicity.

For that reason, OOIS UML allows a relaxed policy for static checking of multiplicity conformance. To that end, the full conformance rule described so far, which includes type, multiplicity, ordering, and uniqueness conformance of typed multiplicity elements (properties, variables, and parameters), is referred to as *strong conformance*. On the other hand, *weak conformance* does not include multiplicity conformance.

Weak conformance shifts the static rules and declarations of some actions on variables and properties toward relaxed constraints. They are summarized as follows:

- Read Value from Variable/Property, as well as `first` and `last`, may return a `null` value even for the variables or properties that do not allow zero cardinality:

```
prop->val() : T[0..1]
prop->first() : T[0..1]
prop->last() : T[0..1]
```

- Clear Variable/Property is applicable to variables and properties that do not allow zero cardinality, but may raise an exception if such a variable or property remains with an illegal cardinality upon the completion of the outermost constrained action group.
- Remove Value and Remove One Occurrence from Variable/Property, as well as Remove Value from Ordered Variable/Property, do not require that the variable or property allows different cardinalities, but may raise an exception if such a variable or property remains with an illegal cardinality upon the completion of the outermost constrained action group.
- Set (Assign) Variable/Property does not require that the assigned value's multiplicity conform to that of the variable/property, but may raise an exception if such a variable or property remains with an illegal cardinality upon the completion of the outermost constrained action group.
- Add Value to Variable/Property does not require that the variable/property allow different cardinalities, but may raise an exception if such a variable or property remains with an illegal cardinality upon the completion of the outermost constrained action group.

A naive compiler would rely on static analysis that simply checks whether or not an action is lexically enclosed in a block of a constrained action group within a method, and apply weak or strong conformance checking, respectively. However, this is not enough because nesting of actions within constrained action groups has dynamic semantics. An action can be written within a method that has no action groups in itself, but that method could be invoked (directly or indirectly) from a constrained action group within another caller method.

OOIS UML leaves the treatment of conformance violations as a semantic variation point in the following way. A model compiler can support one or more of the following compilation options and allow the modeler to select one of them:

- ❑ **Forced strong conformance** — The compiler always checks and forces full multiplicity conformance, and fully obeys all described strong static rules for actions. If any of these are violated, the compiler reports an error, even if the action is specified within a constrained action group.
- ❑ **Weak conformance** — The compiler applies weak conformance and reports errors for its violations. Violations of strong static multiplicity rules in actions, however, may result in warnings, but not errors.
- ❑ **Smart static checking** — If an action is within a constrained group (more precisely, if there is a possibility that the action will be executed within a constrained action group), then weak conformance checking is applied; otherwise, strong conformance checking is applied. The compiler should take into account dynamic nesting of method invocations (that is, operation calls). It should perform a broader static analysis of method dependencies in terms of possible invocations, find out all possible invocations of a considered method (transitively), including polymorphic invocations, and conclude whether or not they are within a constrained action group.

### Section Summary

- ❑ Conformance rules (in particular, multiplicity conformance rules that are checked statically) do not make sense within a context of constrained action groups because they prevent from writing code that moves the system through necessary intermediary states in which multiplicity constraints do not hold.
- ❑ A full conformance rule that includes type, multiplicity, ordering, and uniqueness conformance of typed multiplicity elements is referred to as *strong conformance*. *Weak conformance* does not include multiplicity conformance.
- ❑ A model compiler can allow the modeler to opt for applying weak, strong, or smart conformance policy in static model checking.

### Conceptual Modeling Issues

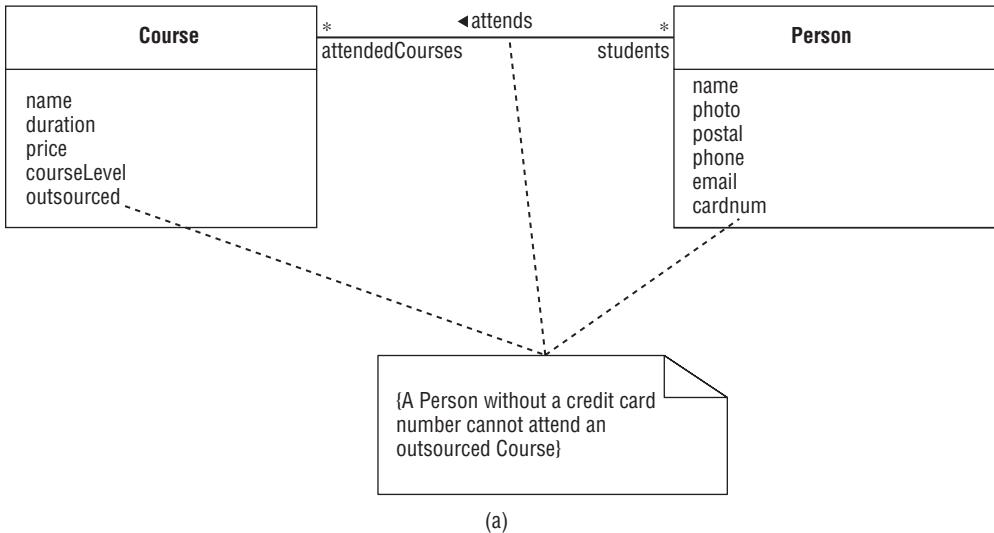
As a consequence of the semantics of constraints in standard UML and OOIS UML, and the conceptual gap between the two, modelers must transform the constraints in the analysis model, specified in standard UML, possibly in an informal language, and attached to several model elements, into fully formal OOIS UML constraints attached to single model elements in the design model, if they want the constraints to have effects at run-time. Otherwise, the constraints would remain informal annotations for documentation purposes, documenting the restrictions posed on the object space probably ensured by other language concepts with executable semantics.

Let's revisit the same example of the Easylearn school system, whose partial analysis model is shown in Figure 11-2a. The analysis model specifies a constraint that a Person without a credit card number cannot attend an outsourced Course. The constraint is written in English, completely informally. Additionally, the constraint is attached to the attributes `Course::outsourced` and `Person::cardnum`, and to the association `attends`, indicating that these elements are involved in the constraint and affect its validity by

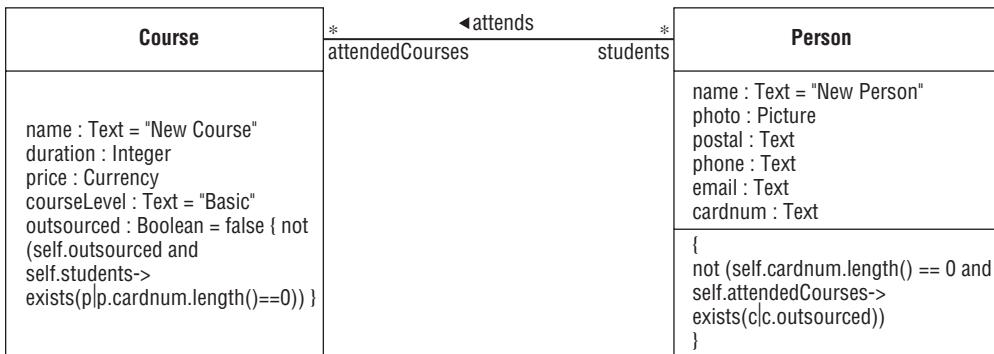
## Part III: Concepts

---

some means. These attachments, however, do not carry any formal semantics (for example, in terms of triggering constraint checking).



(a)



(b)

**Figure 11-2: Constraints in the Easylearn school analysis and design models.** (a) A fragment of the analysis model with a constraint informally specified and attached to several model elements to which it refers. (b) A fragment of the design model with the informal constraint transformed into two formal OOIS UML constraints attached to the model elements to cover all necessary triggers.

The first step in transforming this informal constraint into a formal OOIS UML constraint could be to introduce only the following invariant to the class Person, as shown in Figure 11-1:

```
{not (self.cardnum.length() == 0 and self.attendedCourses->exists(c|c.outsourced))}
```

According to the run-time semantics of OOIS UML constraints, this constraint would be triggered on any action that creates a Person, modifies any of its attributes (cardnum is of sole importance here), or creates or destroys a link with a Person (creating links of attends are of sole importance here). However, this is not sufficient because an action that modifies the attribute Course::outsourced would not trigger this constraint. Therefore, the following scenario would cause the system to go into a state that is not consistent according to the intention of the requirement: an existing Course with outsourced==false is already linked to a Person that does not have a credit card number (which is correct), but then its attribute outsourced is set to true.

To cover this scenario, another constraint should be attached to the attribute Course::outsourced, as shown in Figure 11-2b. This constraint should specify the equivalent rule as the invariant of the class Person, but in the context of the class Course:

```
{not (self.outsourced and self.students->exists(p|p.cardnum.length() == 0))}
```

It is sufficient to trigger this constraint only on actions that modify the attribute Course::outsourced, and it is, thus, not an invariant of the class Course, but is a simple constraint attached to Course::outsourced.

A problem with this approach is that the semantic link between these two constraints, captured in the original informal constraint, is now lost. Thus, during maintenance, one of these may be changed without making a corresponding change in the other. Consequently, it would make sense to define a dependency relationship between the two constraints.

In a general case, the following are some recommendations on how to perform such transformations and identify the model elements to which OOIS UML constraints should be attached:

- ❑ First, transform the specification of the informal constraint to a formal one. Express it in the context of a model element to which the informal constraint naturally belongs. This could be the element that the constraint refers to mostly (the choice can be often multiple). For the given example, this was the invariant of the class Person.
- ❑ Identify the actions that trigger the evaluation of the formal constraint, especially those that affect the validity of the formal constraint. For the given example, these were the actions that trigger the evaluation of the invariant (most notably, the actions that modify the attribute Person::cardnum and association attends).
- ❑ On the other hand, identify the actions on other model elements that may affect the validity of the constraint, but do not trigger the formal constraint. To do that, analyze the formal constraint and identify the model elements referred to by the constraint that do not trigger the introduced constraint. For the given example, this was the attribute Course::outsourced.
- ❑ Introduce other formal constraints, attached to the model elements identified in the previous step, and express them in an equivalent manner, just in the appropriate context. For the given example, this was the constraint attached to the attribute Course::outsourced. Introduce dependencies between these constraints in order to ensure their proper maintenance if needed.

A very advanced modeling tool could help the modeler in doing this transformation by providing hints in this process, or even performing this process semi-automatically (or completely automatically, in some cases), with the approval of the modeler.

### Section Summary

- ❑ The modeler must transform the constraints in the analysis model, specified in standard UML, possibly in an informal language and attached to several elements, into fully formal OOIS UML constraints attached to single model elements in the design model.

## ***Constraints as Objects in OOIS UML***

As described, the constraints defined as model elements are checked implicitly by the execution environment at certain moments in run-time. Therefore, they ensure a permanent consistency of the object space, as specified by the designer. However, there is sometimes a need to simply check and report whether the object space satisfies a certain condition at run-time. Such conditions are not mandatory rules for the system's consistency, but simple queries about some properties of the object space.

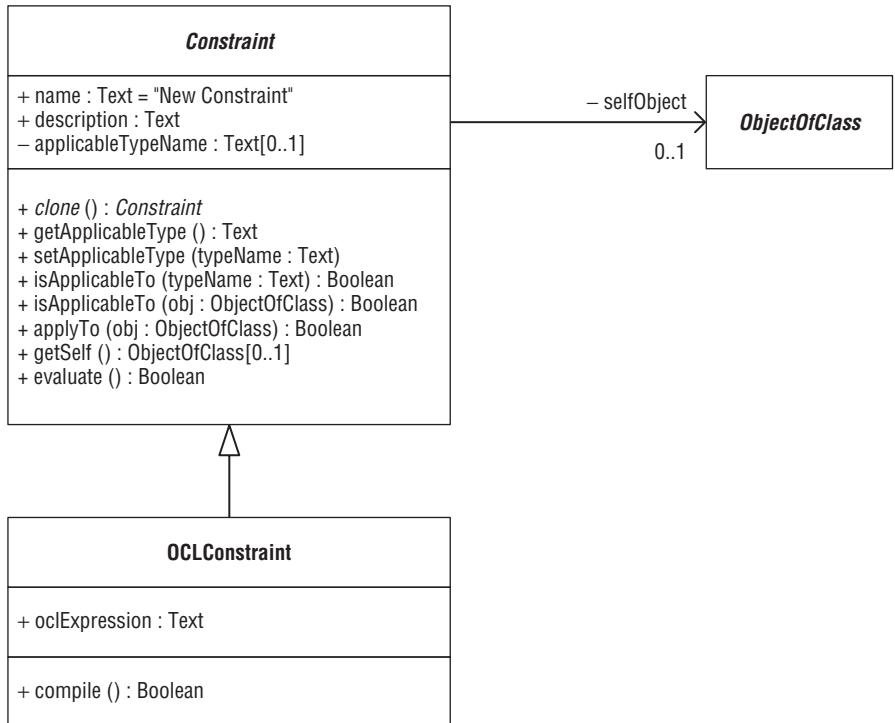
For that purpose, the user can create and define “constraints” as ordinary objects in the application’s object space and evaluate them at run-time. Such “constraints” are actually “condition checkers” that can check certain conditions of the object space when executed. They are evaluated on an explicit request, either from the GUI, or from the code in a detail-level language. When such a condition is violated, it does not necessarily mean that the system is in an inconsistent state, but it may be simply a warning about an unusual situation in the system.

The OOIS UML model library package `Constraints` provides the classes `Constraint` and `OCLConstraint` that support this functionality. The model in this package is shown in Figure 11-3. The class `ObjectOfClass` is an imported class that is a base class of all classes modeled in OOIS UML, including the class `Constraint` shown in the figure.

The abstract class `Constraint` generalizes the concept of constraints (that is, condition checkers) as objects, and provides the features that support some common mechanisms. Additionally, it gathers the common features of constraints that are used or redefined in specializing classes.

The properties of the class `Constraint` have the following meaning:

- ❑ `name` — An arbitrary short name that helps the user to identify the constraint as an object. It does not affect the semantics of the constraint.
- ❑ `description` — An arbitrary description that helps the user to understand the purpose and meaning of the constraint. It does not affect the semantics of the constraint.
- ❑ `applicableTypeName` — A fully qualified name of the class to whose objects the constraint can be applied. If it is not `null`, it specifies the type of objects that will be referred to by the name `self` or a similar name within the specification of the constraint, when the constraint is applied to such an object. It can be `null`, in which case the constraint is not applicable to any object, but is a general (non-parameterized) constraint.
- ❑ `selfObject` — The reference to the object to which the constraint is to be (or has been) applied, and to which the name `self` or a similar reserved name in the constraint’s specification refers when the constraint is evaluated.



**Figure 11-3: The abstract class **Constraint** and the class **OCLConstraint** from an OOIS UML model library allow you to create condition checkers as ordinary objects in the application's object space.**

The operations of this class have the following meanings:

- ❑ **clone** — Abstract operation that supports the Prototype design pattern [Gamma, 1995]. Its method should be defined in the derived class to create a new object of that concrete specializing class and return a reference to it. This operation enables you to create highly configurable systems and GUIs as described for commands in Chapter 6.
- ❑ **getApplicableType, setApplicableType** — Returns/sets the value of the attribute **applicableTypeName**.
- ❑ **isApplicableTo(typeName:Text)** — Returns **true** if the constraint is applicable to an object of the type given by its fully qualified name (that is, if the name **self** or similar can refer to it). The substitution rule is always assumed.
- ❑ **isApplicableTo(:ObjectOfClass)** — Returns **true** if the constraint is applicable to the given object (that is, if the name **self** or similar can refer to it). The substitution rule is always assumed. For example, this operation can be invoked by the generic GUI to indicate whether a selected or dragged constraint can be applied to the target object.
- ❑ **applyTo** — First checks whether this constraint is applicable to the given object (if any) by invoking **isApplicableTo**, and returns **false** if it is not. Otherwise, clones the constraint by invoking **clone**, sets **selfObject** of the clone to refer to the object provided by the argument (if any), and

## Part III: Concepts

---

evaluates the clone constraint by invoking its `evaluate` operation. Then it destroys the clone. It returns `true` if the evaluation succeeded (returned `true`), and `false` otherwise. This operation can be used by the generic GUI to, for example, apply the selected or dragged constraint on the target object.

- ❑ `getSelf` — Returns the `selfObject` reference. This operation is invoked by the methods of the operation `evaluate` of derived classes while the constraint is being evaluated, in order to get the object to which the constraint is applied, and to which the name `self` or similar refers.
- ❑ `evaluate` — The operation that actually evaluates the constraint. It should be overridden in the derived classes to provide the concrete methods for evaluation of different kinds of constraint specifications. It should return the result of that evaluation (`true` if succeeded, `false` if failed). The default method of this operation simply returns `true`.

The classes specializing the class `Constraint` should define concrete mechanisms for specifying and evaluating constraints. The class `OCLConstraint` does this for constraints specified as OCL expressions. Other user-defined classes can do the same for other kinds of constraint specifications. The specializing class should define the methods for the operations `clone` and `evaluate`. The class `OCLConstraint` also provides a helper operation `compile` that checks the correctness of the OCL expression provided in its source form in the attribute `oclExpression` and returns `true` if the expression is correct.

Apart from their use in the generic GUI, constraints as objects can be used programmatically, within user-defined methods. For example, the following code in the OOIS UML native detail-level language applies the given OCL constraint to an object of `Person`:

```
Person aPerson = ...;
OCLConstraint c = new OCLConstraint;
c.setApplicableType("Easylearn::Person");
c.oclExpression = "self.attendedCourses->exists(c|c.outsourced)";
Boolean b = c.applyTo(aPerson);
if (!b) ...
c.destroy();
```

The generic or a customized GUI use the same approach when they apply constraints as objects, as described in Chapter 5. In addition, the same library provides two commands:

- ❑ `CmdEvaluateConstraint` — A command that evaluates the constraint provided as its sole input parameter.
- ❑ `CmdApplyConstraintToObject` — A command that applies the constraint provided as its source input parameter to the object provided as its target input parameter.

These two commands allow for easy customization of the GUI, for example, to evaluate a constraint on double-clicking on it, or to apply a dragged constraint to the object it is dropped onto.

An implementation can provide additional features of these classes for the convenience of their use. For example, an implementation can provide additional information about which part of the object space caused an evaluated constraint to fail, or which part of the OCL expression caused a compilation error.

### Section Summary

- ❑ The OOIS UML model library package `Constraints` contains the classes and commands that allow the user to create and define constraints as ordinary objects in the application's object space, and evaluate them at run-time on an explicit interactive or programmatic request.
- ❑ The abstract class `Constraints::Constraint` generalizes the concept of constraint as an object, and provides the features that support some common mechanisms. Additionally, it gathers the common features of constraints that are used or redefined in specializing classes.
- ❑ The classes specializing the class `Constraints::Constraint` should define concrete mechanisms for specifying and evaluating constraints. The class `Constraints::OCLConstraint` does this for constraints specified as OCL expressions.
- ❑ The commands `Constraints::CmdEvaluateConstraint` and `Constraints::CmdApplyConstraintToObject` allow for easy customization of the GUI, for example, to evaluate a constraint on double-clicking on it, or to apply a dragged constraint to the object it is dropped onto.

## Object Constraint Language

This section describes the Object Constraint Language (OCL) in more detail. Only part of the language is described — that used for constraints attached to structural parts of UML models. Some minor additions related to preconditions and postconditions of operations are described with operations in Chapter 13. For a complete reference to OCL, see the specification [OCL2], on which these descriptions are based. In the OCL expressions in this section, the reserved words of OCL are typed in boldface; this has no semantic significance, but is done for better readability.

### Relation to the UML Model

At the beginning of an OCL constraint, the *context declaration* can show the context of the constraint. This declaration is optional because the context of the OCL constraint is defined in the model, by attaching the constraint to the appropriate model element. The context declaration starts with the reserved word `context` followed by the (optionally fully qualified) name of the context. Following is an example for the Easylearn school example in Figure 11-1:

```
context Person inv:
 not (self.cardnum.length() == 0 and
 self.attendedCourses->exists(c|c.outsourced))
```

## Part III: Concepts

---

The reserved word `inv` specifies that the expression to follow is an invariant of the class that is the context of the constraint. This information is also optional because a constraint attached to a class is always an invariant of that class.

As previously explained, the reserved word `self` refers to an instance for which the constraint is evaluated. It can be omitted, as usual. For example, the following expression is equivalent to the previous one:

```
context Person inv:
not (cardnum.length()==0 and attendedCourses->exists(c|c.outsourced))
```

Alternatively, another name can be introduced to refer to the same object (that is, an alias to `self`) within the context declaration, as follows:

```
context p : Person inv:
not (p.cardnum.length()==0 and p.attendedCourses->exists(c|c.outsourced))
```

The scope of this name is the same as of the context declaration.

Optionally, the name of the constraint (as a named element) can be shown preceding the colon in the context declaration:

```
context Person inv personClassInvariant:
not (cardnum.length()==0 and attendedCourses->exists(c|c.outsourced))
```

All of the previous OCL fragments are equivalent.

Each OCL expression is written in the context of a UML model element and, thus, the types defined in the model and accessible in that context are available within OCL expressions. In OOIS UML, all built-in types are also available in OCL.<sup>3</sup>

Additionally, the navigation over the object space is performed as usual. Standard OCL supports navigation from an object to its slot of a property (attribute or owned association end) or invocation of a query operation of an object, using the dot notation, as already shown in the previous examples.

If the property is multi-valued, the result of the navigation is a collection to which collection operations can be applied over the operator `->`. Consider the following example:

```
aTeacher.taughtCourses->size()>=0 and aTeacher.taughtCourses->size()<=3
```

If the property is single-valued, the result of the navigation can be treated as a collection, as well as a single object (if it exists), to which further navigation can be applied. For example, the following expression ensures that the property `wife` of an instance of the class `Person` is of female gender, if it exists (that is, if the cardinality of the collection is 1). If it does not exist, the navigation to the property `gender` is not performed.

```
aPerson.wife->size()==1 implies aPerson.wife.gender==Gender::female
```

---

<sup>3</sup>Actually, standard OCL introduces some built-in primitive types (such as Boolean, Integer, Real, and String) independently of any UML model. However, because these are built-in types in OOIS UML, they are available anyway.

Standard OCL does not support navigation over association ends that are not properties of classes. However, an implementation of the OOIS UML profile can support it in an extension of OCL using the same notation as in the OOIS UML native detail-level language.

To support type casting, the OCL operation `oclAsType` is available for all types. It results in the reference to the same object to which it is applied, but treated as an instance of the type supplied as the argument of this operation. Further navigation to properties and operations of this result are performed in the namespace of that target type. For example, if the property `p` is defined in the base class `Base`, and another property `p` is defined in the derived class `Derived` (hiding the inherited property), the following OCL expression refers to the property `Derived::p`:

```
context Derived inv:
...self.p...
```

The following one refers to `Base::p`:

```
context Derived inv:
...self.oclAsType(Base) .p...
```

The same operation can be used for downcasting, too.

The following OCL operations are applicable to instances of all types (`OclType` is an OCL built-in type whose instances represent types):

- ❑ `oclIsTypeof(t:OclType)` :Boolean returns `true` if the operand is a direct instance of the given type.
- ❑ `oclIsKindof(t:OclType)` :Boolean returns `true` if the operand is an instance (direct or indirect) of the given type.
- ❑ `oclAsType(t:OclType)` casts the given operand to the given type. The result is undefined if the operand is not of the given type.

The operation `allInstances` applicable to a class in a model results in a set of all existing instances of that class, as shown in the following example:

```
Person.allInstances() ->forAll(p|p.age>0)
```

### Section Summary

- ❑ Optional *context declaration* can show the context of the OCL constraint and may introduce a local name that refers to the same instance as `self`.
- ❑ The optional reserved word `inv` specifies that the expression to follow is an invariant of a class that is the context of the constraint.
- ❑ Optionally, the name of the constraint (as a named element) can be shown in the context declaration.

*Continued*

- ❑ The navigation over the object space in OCL expressions is performed as in the OOIS UML native detail-level language.
- ❑ The following OCL operations are applicable to instances of all types (`OclType` is an OCL built-in type whose instances represent types):
  - ❑ `oclIsTypeof(t:OclType) : Boolean` returns `true` if the operand is a direct instance of the given type.
  - ❑ `oclIsKindof(t:OclType) : Boolean` returns `true` if the operand is an instance (direct or indirect) of the given type.
  - ❑ `oclAsType(t:OclType)` casts the given operand to the given type.
- ❑ The operation `allInstances` applicable to a class in a model results in a set of all existing instances of that class.

## Operators and Expressions

Apart from navigating to slots and operations of classifier instances using the dot notation, OCL allows the use of the following infix operators in expressions: `+, -, *, /, =, <>, <, >, <=, >=`, and, `or`, `xor`, and `implies`.<sup>4</sup> If a classifier defines an operation with the adequate signature that corresponds to one of these operators, it will be called for the infix operator. For example, consider the following infix expression:

`a + b`

This is semantically equivalent to the following expression:

`a.+ (b)`

That is, the classifier of `a` has to have an operation named `+` accepting `b` as the argument.

The operations that correspond to the infix operators must have exactly one parameter. For the infix operators `=, <>, <, >, <=, >=`, and, `or`, `xor`, and `implies`, the operation's return type must be `Boolean`.

Quite similarly, the operators `+, -, and not` can be used as prefix operators. If a classifier defines an operation with the adequate signature that corresponds to one of these operators, it will be called for the prefix operator. Consider the following expression:

`- x`

This is semantically equivalent to invoking the `-` operation on `x` without parameters:

`x.-()`

The operations that correspond to prefix operators must not have parameters. For the prefix operator `not`, the operation's return type must be `Boolean`.

<sup>4</sup>Standard OCL uses `=` instead of `==` for equality, and `<>` instead of `!=` for inequality. However, OOIS UML allows both variants interchangeably, in order to be closer to the syntax of popular programming languages. Similarly, OOIS UML uses `&&` and `and`, `||` and `or`, and `!` and `not` interchangeably.

The precedence order of the operators in OCL, starting with the highest precedence, is as follows (operators with the same precedence level are in the same row):

```

@

., ->

prefix operators not, +, and -

*, /

infix operators + and -

if-then-else-endif

<, >, <=, >=

=, <>

and, or, xor

implies.

```

As usual, parentheses () that enclose sub-expressions can be used to change the default precedence order.

The built-in type `Boolean` implements all logic operators `not`, `and`, `or`, `xor`, and `implies`, while `Integer` and `Real` implement arithmetic and relational operators with the appropriate semantics. For logic operators, these are the following:

- True or anything is true
- False and anything is false
- False implies anything is true
- Anything implies true is true.

The rules for `and` and `or` hold regardless of the order of the operands.

The `if-then-else-endif` operator is a ternary operator (with three operands). Its syntax is as follows:

```
if condExpression then thenExpression else elseExpression endif
```

It is evaluated as follows. The `condExpression` is evaluated; it must result in a Boolean value. If its result is true, `thenExpression` is evaluated and its result represents the result of the `if-then-else-endif` operator. Otherwise, `elseExpression` is evaluated and returned as the result of the operator. Both `thenExpression` and `elseExpression` are mandatory to ensure that the operator always returns a value. Consider the following example:

```
context Employee inv:
 if self.job->size()=0 then self.salary=0 else self.salary>100 endif
```

Sometimes a sub-expression is used more than once in a constraint. The `let` expression allows you to define a local variable that can be used in the constraint. In this example, the `let` expression introduces a local variable `income` of type `Integer`:

```
context Employee inv:
 let income : Integer = self.job.salary->sum() in
 if self.isUnemployed then
```

## Part III: Concepts

---

```
income=0
else
 income>100
endif
```

A `let` expression may be included in any kind of OCL expression. The variable introduced in a `let` expression is visible only within its nesting expression.

Some expressions evaluate to an undefined value. For example, casting with `oclAsType` to a type of which the operand is not an instance or accessing an element out of an empty collection will result in an undefined value. In general, when one part of an expression is undefined, the result of the entire expression will be undefined. There are some exceptions to this rule, however.

First, the results of logical operators (according to the rules given previously) are valid as long as any of the operands are valid (the “anything” in the definition of their semantics includes also the undefined value). Second, the `if-then-else-endif` expression is valid as long as the chosen branch is valid (regardless of the value of the other branch). Finally, there is an explicit operation for testing if the value of an expression is undefined. `oclIsUndefined` is an operation applicable to any value and results in `true` if the value is undefined, and in `false` otherwise.

The OCL reserved words cannot occur anywhere in an OCL expression as the name of a package, a type, or a feature.

### Section Summary

- ❑ The operators `+`, `-`, `*`, `/`, `=`, `<>`, `<`, `>`, `<=`, `>=`, and, `and`, `or`, `xor`, and `implies` are used as infix operators. If a classifier defines an operation with the adequate signature that corresponds to one of these operators, it will be called by the infix operator. That means, the following expressions are semantically equivalent:  
$$a + b$$
$$a.+ (b)$$
- ❑ This similarly holds true for unary operators `+`, `-`, and `not` that can be used as prefix operators.
- ❑ The ternary `if-then-else-endif` operator results in `thenExpression` if `condExpression` is `true`, or in `elseExpression` otherwise.
- ❑ The `let` expression enables you to define a local variable that can be used in an expression.
- ❑ In general, when one part of an expression is undefined, the result of the entire expression will be undefined. There are some exceptions to this rule, however.

## Tuples

In OCL, several values can be composed into a *tuple* of named components. Tuples in OCL are types that represent simple structures of named components, just as `struct` in C or `record` in Pascal. Each component of a tuple can have its own type. The components of a tuple are referred to by their names.

Tuple literals are defined in the following way. They start with the keyword `Tuple`, followed by a pair of curly braces within which the components are defined and separated by commas. Each component is defined by its name, type, and value. The ordering of the components is irrelevant. Consider the following example:

```
Tuple { name : String = 'John Smith', age : Integer = 25 }
Tuple { arr : Collection(Integer) = Set{1, 2, 5}, txt : String = 'Hello' }
```

The values of the components may be specified by arbitrary OCL expressions, written in the scope of the expression enclosing the tuple.

For the literals just given, the tuple types are expressed in this way:

```
TupleType(name : String, age : Integer)
TupleType(arr : Collection(Integer), txt : String)
```

The elements of a tuple are referred to by their names over the dot operator. For example, if `expr` results in a `TupleType(x:Integer, y:Integer)`, then `expr.x` refers to the component `x` of the tuple.

Tuples are mostly used to refer to output parameters of called operations, or to collect some temporary structures of values in complex OCL expressions.

### Section Summary

- A *tuple type* is a structure of named components of certain types:

```
TupleType(c1:T1, c2:T2, ...)
```

- Tuple literals are defined in the following way:

```
Tuple { name : String = 'John Smith', age : Integer
= 25 }
```

- Components of a tuple are referred to by name, over the dot operator.

## Collections

OCL defines the following collection types as part of its built-in type hierarchy:

- `Collection` — An abstract generalization of all collection types. It defines a common interface (a set of operations) of all kinds of collections.
- `Bag` — A specialization of `Collection` that represents a bag, and corresponds to an unordered, non-unique typed multiplicity element.
- `Sequence` — A specialization of `Collection` that represents a sequence, and corresponds to an ordered, non-unique typed multiplicity element.
- `Set` — A specialization of `Collection` that represents a set, and corresponds to an unordered, unique typed multiplicity element.

## Part III: Concepts

---

- ❑ OrderedSet — A specialization of Collection that represents an ordered set, and corresponds to an ordered, unique typed multiplicity element.

All navigations in OCL expressions result in one of the collection types, corresponding to the ordering and uniqueness of the navigated element (that is, attribute or association end). This similarly holds true for results of operations. Only if a navigation over the dot operator results in a single-valued collection can it be further navigated over the dot operator.

The collection types define a large number of predefined operations on collections. All these operations are query operations because they never modify the operand collections. If they result in collections, they project the result into a new collection instead of changing the original one. A collection operation can be invoked over the arrow operator  $\rightarrow$ .

Collections can be defined by literals, whereby the elements of a literal collection are specified within curly braces {}. The concrete type of the collection is written preceding the braces.<sup>5</sup> For example, these are collection literals of the corresponding types:

```
Set{2, 1, 5}
OrderedSet{'alpha', 'beta', 'gamma'}
Bag{2, 0, 3, 5, 0, 1, 1}
Sequence{1, 2, 3, 4, 4, 5, 5, 6}
```

For ordered collection literals, the sequence can be specified by an interval specification  $m..n$ , whereby  $m$  and  $n$  are constant expressions of type Integer. The ordered collection defined in this way consists of a contiguous sequence of integer numbers between  $m$  and  $n$ , inclusively. For example, these collection literals are equal:

```
Sequence{0, 1, 2, 3, 4, 5}
Sequence{0..5}
Sequence{0..(7-2)}
```

In short, a collection can be obtained within an OCL expression only in one of the three defined ways:

- ❑ By a collection literal.
- ❑ By a navigation over the dot operator to a property (meaning reading the slot and obtaining the collection as the value of the slot) or to an operation (meaning calling the operation and obtaining its result as a collection).
- ❑ An operation on a collection may result in another collection (for example, `collection1->union(collection2)`). Operations on collections are explained in the next section.

### Section Summary

- ❑ OCL defines the following collection types as part of its built-in type hierarchy:
  - ❑ Collection — An abstract generalization of all collection types.

---

<sup>5</sup>In the OOIS UML dialect of OCL, collection literals can also be specified as in the OOIS UML native detail-level language.

- Bag — A specialization of Collection that represents a bag, and corresponds to an unordered, non-unique typed multiplicity element.
- Sequence — A specialization of Collection that represents a sequence, and corresponds to an ordered, non-unique typed multiplicity element.
- Set — A specialization of Collection that represents a set and corresponds to an unordered, unique typed multiplicity element.
- OrderedSet — A specialization of Collection that represents an ordered set, and corresponds to an ordered, unique typed multiplicity element.
- All navigations and operation calls over the dot operator in OCL expressions result in one of the collection types, corresponding to the ordering and uniqueness of the result.
- All operations on collections are query operations because they never modify the operand collections. If they result in collections, they project the result into a new collection instead of changing the original one. A collection operation can be invoked over the operator `->`.
- A collection can be obtained within an OCL expression only in one of these three ways:
  - By a collection literal (for example, `Sequence{1, 2, 3, 2, 5, 5, 4}`)
  - By a navigation over the dot operator to a property or to an operation of a classifier instance
  - An operation on a collection may result in another collection (for example, `collection1->union(collection2)`)

## Operations on Collections

Collections are actually template types, with the type of elements being the parameter of the collection type. That is, a collection is always homogeneous and contains only elements of a certain type  $T$  (the substitution rule is always assumed). Actually, a collection of elements of type  $T$  corresponds to a typed multiplicity element with the type  $T$  in OCL UML. The conformance rules prevent mixing collections of elements of unrelated types, as in OCL UML. In the declarations that follow, a collection of elements of type  $T$  is denoted with `Collection(T)`, `Bag(T)`, `Sequence(T)`, `Set(T)`, and `OrderedSet(T)`, depending on ordering and uniqueness. The type  $T$  of a collection can be a collection type, too.

The collection types define a large number of predefined operations that manipulate collections. All these operations are query operations, because they never modify the operand collections. If they result in collections, they project the result into a new collection instead of changing the original one. A collection operation can be invoked over the arrow operator `->`.

Following are the common operations available for the Collection type:

- `size():Integer` — Returns the number of elements in the collection.
- `includes(e:T):Boolean` — Is the given element in the collection?

## Part III: Concepts

---

- `excludes(e:T):Boolean` — Is the given element not in the collection?
- `count(e:T):Integer` — Returns the number of times the given element occurs in the collection.
- `includesAll(c:Collection(T)):Boolean` — Does the collection include all the elements of c?
- `excludesAll(c:Collection(T)):Boolean` — Does the collection include none of the elements of c?
- `isEmpty():Boolean` — Is the collection empty?
- `notEmpty():Boolean` — Is the collection not empty?
- `sum():T` — Returns the sum of all elements in the collection. The type T must support the addition operation +. The + operation must take one parameter of type T and be both associative:  $(a+b)+c = a+(b+c)$ , and commutative:  $a+b = b+a$ . Integer and Real fulfill this condition.
- `product(c:Collection(T2)):Set(Tuple(first:T, second:T2))` — Returns the Cartesian product of this collection and c.
- `asSet():Set(T)` — Returns this collection as a set. Redefined in specializations.
- `asOrderedSet():OrderedSet(T)` — Returns this collection as an ordered set. Redefined in specializations.
- `asBag():Bag(T)` — Returns this collection as a bag. Redefined in specializations.
- `asSequence():Sequence(T)` — Returns this collection as a sequence. Redefined in specializations.

The following are operations added in the Set type:

- `= (s:Set(T)):Boolean` — Do this set and s have exactly the same elements?
- `union(s:Set(T)):Set(T)` — Returns the union of this set and s.
- `union(b:Bag(T)):Bag(T)` — Returns the union of this set and b. The number of occurrences of every element in the resulting bag is the sum of the numbers of its occurrences in this set (can be only 0 or 1) and in b.
- `intersection(s:Set(T)):Set(T)` — Returns the intersection of this set and s.
- `intersection(b:Bag(T)):Set(T)` — Returns the intersection of this set and b as a set.
- `-(s:Set(T)):Set(T)` — Returns the difference of this set and s (the set of elements that exist in this set and not in s).
- `including(e:T):Set(T)` — Returns the union of this set and a set containing only e.
- `excluding(e:T):Set(T)` — Returns this set without the element e.
- `symmetricDifference(s:Set(T)):Set(T)` — Returns the set of elements that exist in this set or in s, but not in both.
- `flatten():Set(T2)` — If the element type is not a collection type, this results in the same set. If the element type is a collection type, the result is the set containing all the elements of all the elements of this set.
- `asSet():Set(T)` — Returns a set identical to this set. This operation exists for convenience only.
- `asOrderedSet():OrderedSet(T)` — Returns an ordered set with the same elements as in this set, in an undefined order.

- ❑ `asBag() :Bag(T)` — Returns a bag with the same elements as in this set.
- ❑ `asSequence() :Sequence(T)` — Returns a sequence with the same elements as in this set, in an undefined order.

The following are operations added in the `OrderedSet` type:

- ❑ `= (s:OrderedSet(T)) :Boolean` — Do this set and s have exactly the same elements in the same order?
- ❑ `append(e:T) :OrderedSet(T)` — Returns an ordered set consisting of all elements of this set, followed by the element e.
- ❑ `prepend(e:T) :OrderedSet(T)` — Returns an ordered set consisting of the element e, followed by all elements of this set.
- ❑ `insertAt(index:Integer, e:T) :OrderedSet(T)` — Returns an ordered set consisting of all elements of this set, with the element e inserted at the position index.
- ❑ `subOrderedSet(lower:Integer, upper:Integer) :OrderedSet(T)` — Returns a subset of this set, consisting of the elements between the positions lower and upper, inclusively.
- ❑ `at(index:Integer) :T` — Returns the element at the position index.
- ❑ `indexOf(e:T) :Integer` — Returns the position of the element e.
- ❑ `first() :T` — Returns the first element.
- ❑ `last() :T` — Returns the last element.
- ❑ `asSet() :Set(T)` — Returns a set with the same elements as in this set.
- ❑ `asOrderedSet() :OrderedSet(T)` — Returns an ordered set identical to this set. This operation exists for convenience only.
- ❑ `asBag() :Bag(T)` — Returns a bag with the same elements as in this set.
- ❑ `asSequence() :Sequence(T)` — Returns a sequence with the same elements as in this ordered set.

The following are operations added in the `Bag` type:

- ❑ `= (b:Bag(T)) :Boolean` — Do this bag and b have exactly the same elements, the same number of times?
- ❑ `union(s:Set(T)) :Bag(T)` — Returns the union of this bag and s. The number of occurrences of every element in the resulting bag is the sum of the numbers of its occurrences in this bag and in s.
- ❑ `union(b:Bag(T)) :Bag(T)` — Returns the union of this bag and b. The number of occurrences of every element in the resulting bag is the sum of the numbers of its occurrences in this bag and in b.
- ❑ `intersection(s:Set(T)) :Set(T)` — Returns the intersection of this bag and s.
- ❑ `intersection(b:Bag(T)) :Bag(T)` — Returns the intersection of this bag and b.
- ❑ `including(e:T) :Bag(T)` — Returns a bag having all the elements of this bag plus one occurrence of e.

## Part III: Concepts

---

- ❑ `excluding(e:T) :Bag(T)` — Returns a bag having all elements of this bag without all occurrences of e.
- ❑ `flatten() :Bag(T2)` — If the element type is not a collection type, this results in an identical bag. If the element type is a collection type, the result is a bag containing all the elements of all the elements of this bag.
- ❑ `asSet() :Set(T)` — Returns a set having all those and only those elements that occur in this bag.
- ❑ `asOrderedSet() :OrderedSet(T)` — Returns an ordered set having all those (and only those) elements that occur in this bag, in an undefined order.
- ❑ `asBag() :Bag(T)` — Returns a bag identical to this bag. This operation exists for convenience only.
- ❑ `asSequence() :Sequence(T)` — Returns a sequence with the same elements as in this bag, in an undefined order.

The following are operations added in the Sequence type:

- ❑ `= (s:Sequence(T)) :Boolean` — Do this sequence and s have exactly the same elements in the same order?
- ❑ `union(s:Sequence(T)) :Sequence(T)` — Returns a sequence consisting of all elements of this sequence, followed by all elements of s.
- ❑ `append(e:T) :Sequence(T)` — Returns a sequence consisting of all elements of this sequence, followed by the element e.
- ❑ `prepend(e:T) :Sequence(T)` — Returns a sequence consisting of the element e, followed by all elements of this sequence.
- ❑ `insertAt(index:Integer, e:T) :Sequence(T)` — Returns a sequence consisting of all elements of this sequence, with the element e inserted at the position index.
- ❑ `subSequence(lower:Integer, upper:Integer) :Sequence(T)` — Returns a sub-sequence of this sequence, consisting of the elements between the positions lower and upper, inclusively.
- ❑ `at(index:Integer) :T` — Returns the element at the position index.
- ❑ `indexOf(e:T) :Integer` — Returns the position of the element e.
- ❑ `first() :T` — Returns the first element.
- ❑ `last() :T` — Returns the last element.
- ❑ `including(e:T) :Sequence(T)` — Returns a sequence having all elements of this sequence plus e as the last element.
- ❑ `excluding(e:T) :Sequence(T)` — Returns a sequence having all elements of this sequence without all occurrences of e.
- ❑ `flatten() :Sequence(T2)` — If the element type is not a collection type, this results in an identical sequence. If the element type is a collection type, the result is a sequence containing all the elements of all the elements of this sequence, in order.
- ❑ `asSet() :Set(T)` — Returns a set having all those and only those elements that occur in this sequence.
- ❑ `asOrderedSet() :OrderedSet(T)` — Returns an ordered set having all those and only those elements that occur in this sequence, in an undefined order.

- ❑ `asBag () :Bag (T)` — Returns a bag identical to this sequence.
- ❑ `asSequence () :Sequence (T)` — Returns a sequence identical to this sequence. This operation exists for convenience only.

### Section Summary

- ❑ Collection types define a large number of operations for flexible manipulation with collections as mathematical and programming concepts (such as intersections, unions, sub-sequences, concatenations, insertions, exclusions, and so on).
- ❑ All these operations are query operations because they never modify the operand collections. If they result in collections, they project the result into a new collection instead of changing the original one.
- ❑ A collection operation can be invoked over the arrow operator `->`.

## Iterations on Collections

The most expressive part of OCL is the capability to address only some elements of collections that satisfy certain criteria, and to produce new collections or Boolean results based on those elements. Such constructs are defined as *iterations* on collections in OCL, and are invoked as the operations on collections, over the arrow operator `->`.

The iterator `select` returns a sub-collection of the given collection, consisting of all those (and only those) elements of the iterated collection that satisfy a given Boolean criterion. Its first form is as follows:

```
collection->select(booleanExpression)
```

For example, the following OCL expression written in the context of the class `Teacher` ensures that the `Teacher` teaches at least one `Course` with price over 100:

```
self.taughtCourses->select(price>100)->notEmpty()
```

The result of the `select` iterator is the collection of all `taught Courses` that satisfy the criterion `price>100`.

As you can see, the context of the Boolean expression within the `select` iterator is the type of the element of the iterated collection. In the given example, it is the class `Course`. The Boolean expression cannot address the element itself (a `Course`, in this case), but only its properties and operations. To refer to the element itself, a more general form of the iterator can be used:

```
collection->select(var | booleanExpressionWithVar)
```

The variable `var` is a name introduced into the scope of the `select` iterator, actually, into its Boolean expression. It is of the type of the element of the collection and refers to every element of the collection on which the Boolean expression is evaluated. Therefore, the Boolean expression can use that name to refer to the element of the collection. Such a variable is called the *iterator* because, when `select` is evaluated, the iterator iterates over the collection and the Boolean expression is evaluated for each value of the iterator.

## Part III: Concepts

---

For example, the following two expressions written in the context of the class Teacher are equivalent:

```
self.taughtCourses->select(price>100)->notEmpty()

self.taughtCourses->select(c | c.price>100)->notEmpty()
```

The type of the iterator variable can be explicitly specified, too. The type must be the type of the collection's elements. Consider the following example:

```
self.taughtCourses->select(c:Course | c.price>100)->notEmpty()
```

In general, the following are the three forms of the select iterator:

```
collection->select(var : type | booleanExpressionWithVar)
collection->select(var | booleanExpressionWithVar)
collection->select(booleanExpression)
```

The reject iterator has the same three forms as select and also returns a sub-collection of the original collection, but consisting of all those and only those elements of the original collection that do not satisfy the given Boolean expression. Consequently, the following expressions are equivalent:

```
collection->reject(var : type | booleanExpression)
collection->select(var : type | not (booleanExpression))
```

The iterator sortedBy can be applied to an ordered or unordered collection and always returns an ordered collection, with the elements ordered by the provided criterion. It has the same three forms:

```
collection->sortedBy(var : type | expressionWithVar)
collection->sortedBy(var | expressionWithVar)
collection->sortedBy(expression)
```

If the original collection is a set, the resulting collection is an ordered set. If the original collection is a bag, the result is a sequence. The type of the result of the provided expression must have the `<` operator defined. The operator must return a Boolean and must be transitive: if  $a < b$  and  $b < c$  then  $a < c$ . The resulting collection has the same size as the original collection. The resulting collection is ordered in the non-decreasing order of the values of the provided expression evaluated for each iterated element of the original collection. For example, the following expression results in a collection of Courses attended by a Person, ordered by the price:

```
aPerson.attendedCourses->sortedBy(price)
```

The iterators select and reject always return sub-collections of the original collections. It is sometimes necessary to construct a completely different collection by iterating over the original collection. The iterator collect iterates over the original collection, evaluates the provided expression for each element of that collection, and adds the result of the evaluated expression to the resulting collection. In other words, the result of collect is the collection of all the results of the evaluations of the provided expression. It has the same three syntactical forms as select and reject, but the provided expression does not need to have a Boolean result — it can be of any type. That type will be the type of the elements of the resulting collection.

```
collection->collect(var : type | expressionWithVar)
collection->collect(var | expressionWithVar)
collection->collect(expression)
```

For example, the following expression written in the context of Teacher results in a collection of real numbers equal to the prices of the taught Courses, increased by 10 percent:

```
self.taughtCourses->collect(c:Course | c.price*1.1)
```

The resulting collection is always non-unique, even if the iterated collection is unique. This is because the provided expression may result in equal values for several elements of the iterated collection. The ordering of the resulting collection is the same as of the original collection. That means, if the original collection is a set or a bag, the resulting collection is a bag. If the original collection is an ordered set or a sequence, the resulting collection is a sequence. The resulting collection is always of the same size as the original collection.

In OCL, navigation through many objects is very common. To make such navigations more concise and readable, there is a shorthand notation for collect. Instead of

```
self.taughtCourses->collect(price)
```

you could also write:

```
self.taughtCourses.price
```

In general, when the left-hand operand of the dot operator is a collection of values, then it is interpreted as a collect over the elements of the collection with the specified property as the expression of collect. That is, the notation in the following two expressions is equivalent:

```
collection.propertyName
collection->collect(propertyName)
```

The same holds true for operation invocations on the elements of a collection. That is, the following notations are equivalent:

```
collection.operationName(parameterList)
collection->collect(operationName(parameterList))
```

The forAll iterator checks whether the provided Boolean expression holds for every element of the iterated collection. It has the same three syntactical forms:

```
collection->forAll(var : type | booleanExpressionWithVar)
collection->forAll(var | booleanExpressionWithVar)
collection->forAll(booleanExpression)
```

Its result is true if the provided Boolean expression evaluates to true for all elements of the iterated collection, and false otherwise. Here is an example:

```
Person.allInstances()->forAll(p|p.age>0)
```

## Part III: Concepts

---

The `forall` iterator has an extended variant that allows arbitrarily many iterator variables to be specified. All iterators will iterate over the complete collection. Effectively, this is a `forall` on the Cartesian product of the collection with itself. Consider the following example:

```
Person.allInstances() -> forall(p1, p2 | p1 <> p2 implies p1.name <> p2.name)
```

This is semantically equivalent to the following:

```
Person.allInstances() -> forall(p1 |
 Person.allInstances() -> forall(p2 |
 p1 <> p2 implies p1.name <> p2.name))
```

The `exists` iterator checks whether the provided Boolean expression holds true for any element of the iterated collection. It results in a Boolean value that is `true` if and only if the Boolean expression results to `true` for at least one iterated element of the collection. It has the same three syntactical forms:

```
collection->exists(var : type | booleanExpressionWithVar)
collection->exists(var | booleanExpressionWithVar)
collection->exists(booleanExpression)
```

The following example returns `true` if and only if there is a person older than 100 years:

```
Person.allInstances() -> exists(p | p.age > 100)
```

The following iterators have the same three syntactical forms and provide the following features:

- `isUnique(var : type | expressionWithVar)` — Results in `true` if the provided expression evaluates to a different value for each element in the collection; otherwise, the result is `false`.
- `one(var : type | booleanExpressionWithVar)` — Results in `true` if the provided expression evaluates to `true` for exactly one element in the collection; otherwise, the result is `false`.
- `any(var : type | booleanExpressionWithVar)` — Returns any element from the collection for which the provided expression evaluates to `true`. If there are more elements for which the expression evaluates to `true`, one of them is returned. If there are no such elements, the result is `null`.

The iterator `iterate` is the most complicated, but the most general one, because most other iterators can be expressed in terms of `iterate`. Its form is as follows:

```
collection->iterate(var : type; acc:type=initExpression | expressionWithVarAndAcc)
```

The variable `var` is the iterator variable, as in the definition of other iterators. The variable `acc` is the accumulator. The accumulator gets an initial value specified by `initExpression` before the iteration. When `iterate` is evaluated, `var` iterates over the collection and the `expressionWithVarAndAcc` is evaluated for each element. After each evaluation of the expression, its result is assigned to the accumulator. In this way, the value of the accumulator is built up during the iteration of the collection. The value of the accumulator is returned as the result of the expression.

For example, consider the following `collect` iteration:

```
collection->collect(e : T | e.aProperty)
```

This is identical to the following:

```
collection->iterate(e : T; acc : TypeOfProperty = Bag{} |
acc->including(e.aProperty))
```

## Section Summary

- Return a sub-collection of the iterated collection, consisting of all those and only those elements of the iterated collection that satisfy the provided Boolean criterion:

```
collection->select(var : type | booleanExpressionWithVar)
```

- Return a sub-collection of the iterated collection, consisting of all those and only those elements of the iterated collection that do not satisfy the provided Boolean criterion:

```
collection->reject(var : type | booleanExpressionWithVar)
```

- Return an ordered collection with the same contents as the iterated collection, but sorted in the non-decreasing order according to the result of the provided expression, evaluated for each element of the iterated collection:

```
collection->sortedBy(var : type | expressionWithVar)
```

- Return any element of the given collection that satisfies the provided Boolean criterion:

```
collection->any(var : type | booleanExpressionWithVar)
```

- Return a new collection consisting of all results of the provided expression evaluated on every element of the iterated collection:

```
collection->collect(var : type | expressionWithVar)
```

- Return `true` if the given Boolean expression evaluates to `true` for every element of the iterated collection, and `false` otherwise:

```
collection->forAll(var : type | booleanExpressionWithVar)
```

- Return `true` if the given Boolean expression evaluates to `true` for any element of the iterated collection, and `false` otherwise:

```
collection->exists(var : type | booleanExpressionWithVar)
```

- Return `true` if the given Boolean expression evaluates to `true` for exactly one element of the iterated collection, and `False` otherwise:

```
collection->one(var : type | booleanExpressionWithVar)
```

*Continued*

- ❑ Return `true` if the given expression evaluates to a different value for every element of the iterated collection, and `false` otherwise:

```
collection->isUnique(var : type | expressionWithVar)
```

- ❑ Iterate through the collection and store the value of the evaluated expression to the accumulator in each iteration:

```
collection->iterate(var:type; acc:type=initExpression
| expressionWithVarAndAcc)
```

### OOIS UML Dialect of OCL

OOIS UML uses a dialect of OCL that introduces some minor lexical, syntactical, and semantic extensions or variations of standard OCL in order to bring it closer to the OOIS UML native detail-level language and make it easier to use both in any context (a constraint or a method). In that way, the modeler has a smoother learning curve, and can adopt the syntax and semantics of both languages more easily.

The first group of extensions of OCL in its OOIS UML dialect is purely lexical. In standard OCL, a comment is written after two minus signs. Everything starting with two minus signs up to the end of line is a comment and is ignored by the compiler (that is, has no semantic significance). Consider the following example:

```
context Teacher inv: -- This is a comment on the invariant of Teacher
```

In the OOIS UML, apart from this kind of comment, the C/C++-like comments are also acceptable, as in the OOIS UML native detail-level language:

```
context Teacher inv: // This is a comment on the invariant of Teacher
self.age > /* this is an embedded comment ignored by a parser */ 20
```

Similarly, standard OCL uses the operators `=` and `<>` for equality and inequality relations, and the keywords `not`, `and`, `and` or `or` for logic operators. Apart from these, the OOIS UML dialect allows the C-like operators with the same meaning, respectively: `==`, `!=`, `!`, `&&`, and `||`.

The second group consists of syntactical extensions. The literals (including collection literals) can be specified as in standard OCL, as well as in the OOIS UML native detail-level language. For example, string-literals can be enclosed in apostrophes ('') or in quotation marks (""):

```
let s : String = 'Hello world!' -- This is a string literal in standard OCL
```

```
let s : String = "Hello world!" // This is a string literal in OOIS UML OCL
```

Collection literals can be written without specifying their type, if the type can be unambiguously derived from the context of the specification. Consider the following example:

```
let c : OrderedSet(Integer) = {1, 2, 5, 3}
```

Variables can be declared as in standard OCL, as well as in the OOIS UML native detail-level language. The following table shows the equivalence of collection types in standard OCL and in OOIS UML:

| Standard OCL   | OOIS UML                  |
|----------------|---------------------------|
| Bag (T)        | T[*]{nonunique,unordered} |
| Sequence (T)   | T[*]{nonunique,ordered}   |
| Set (T)        | T[*]{unique,unordered}    |
| OrderedSet (T) | T[*]{unique,ordered}      |
| T              | T[0..1]                   |

For example, these two declarations are equivalent:

```
let c : Bag(T) in ...
let T[*]{nonunique,unordered} c in ...
```

Non-collection variables in standard OCL are equivalent to the variables of the same type and with multiplicity 0..1 in OOIS UML.

The semantic enhancement in the OOIS UML dialect of OCL is in the type conformance rules for collections. Namely, in standard OCL, the following are the type conformance rules:

- ❑ Type1 conforms to Type2 when they are identical (standard rule for all types).
- ❑ Type1 conforms to Type2 when it is a subtype of Type2 (standard rule for all types).
- ❑ Collection(Type1) conforms to Collection(Type2), when Type1 conforms to Type2. This is also true for Set(Type1)/Set(Type2), OrderedSet(Type1)/OrderedSet(Type2), Sequence(Type1)/Sequence(Type2), and Bag(Type1)/Bag(Type2).
- ❑ Type conformance is transitive (standard rule for all types).

However, Bag(T), Sequence(T), Set(T), and OrderedSet(T) are just subtypes of Collection(T), and there are no other relations between them. Consequently, OrderedSet(T) does not conform to Set(T) nor to Sequence(T). Set(T) does not conform to Bag(T), and Sequence(T) does not conform to Bag(T), either. On the other hand, there is conformance of the corresponding types in OOIS UML. Therefore, the OOIS UML dialect of OCL extends the conformance rules to allow the previously mentioned conformance relations. For example, in the OOIS UML dialect of OCL, Set(T) conforms to Bag(T), OrderedSet(T) conforms to Set(T) and Sequence(T), and so on. As a result, every expression that is regular in standard OCL is also regular in the OOIS UML dialect of OCL, but not vice versa.

Finally, the OOIS UML dialect supports operations on the collections that correspond to all read actions on attributes and association ends. This includes actions on association ends that do not belong to classes, especially on N-ary association ends. Additionally, if the signatures (the name of the operation and the order and types of its parameters) of the corresponding operations differ, the OOIS UML dialect of OCL

## Part III: Concepts

---

supports both variants, the one from standard OCL and the one from OOIS UML. Here are several examples:

- ❑ There is an operation named `at` for ordered collections in OOIS UML that does not exist in standard OCL, and is supported by its OOIS UML dialect.
- ❑ The operation that inserts an element into an ordered collection at the given position in OOIS UML is `addAt(value, position)`, while the corresponding operation in standard OCL is `insertAt(position, value)`; the OOIS UML dialect supports both variants with the same meaning.

The extension also includes different treatment of indices (1-based in standard OCL and 0-based in OOIS UML), as well as the behavior in some exceptional cases (such as out-of-range indices, empty collections, and so on). These aspects are left to the implementation as semantic variation points.

The influence is also the opposite. The implementation of the OOIS UML native detail-level language can support both variants of signatures for read actions on attributes and association ends, and introduce other operations that are not defined in OOIS UML, but exist in OCL. For example, the OCL operations `indexOf`, `isEmpty`, and `notEmpty` do not exist in the list of read actions in OOIS UML, but can be introduced by an OOIS UML implementation.

As a result, there is a tendency of both languages to converge with each other so that both have the same notation (including signatures of operations) and the same semantics of the corresponding operations on collections. This will lead to a unified detail-level language that can be uniformly used in different contexts (that is, in constraints and methods), and ease the learning curve.

### Section Summary

- ❑ OOIS UML uses a dialect of OCL that introduces some minor lexical, syntactical, and semantic extensions or variations of standard OCL in order to bring it closer to the OOIS UML native detail-level language and make it easier to use both languages in different contexts (a constraint or a method).
- ❑ The lexical extensions include the following:
  - ❑ The dialect supports the standard (started with `--`) as well as C++-like comments (starting with `//` or between `/*` and `*/`).
  - ❑ The dialect allows the standard OCL operators `=`, `<>`, `not`, `and`, `and` `or` as well as the C-like operators with the same meaning, respectively: `==`, `!=`, `!`, `&&`, and `||`.
- ❑ Syntactical extensions include the following:
  - ❑ Literals (including collection literals) can be specified as in standard OCL, as well as in the OOIS UML native detail-level language.
  - ❑ Variables can be declared as in standard OCL, as well as in the OOIS UML native detail-level language.

- ❑ A semantic extension is in relaxed type-conformance rules for collections, corresponding to the OOIS UML type conformance rules for typed multiplicity elements.
- ❑ The OOIS UML dialect of OCL supports an extended set of collection operators and navigation over association ends not belonging to classes (including N-ary association ends), for all read actions in OOIS UML.



# 12

## Querying

Queries provide a means to identify a piece of the entire object space of the running system according to certain matching criteria, and return a collection of tuples of objects and data values as a result. A query can be specified in different ways. Two ways available in OOIS UML, by default, are the Object Query Language (OQL) and pattern object structures.

### Queries as Model Elements or Objects

Queries can be defined at design time, as elements of the model, or created at run-time, as objects of the class `Query` from the OOIS UML model library.

#### ***The Semantics of Queries in OOIS UML***

Regardless of the concrete way a query is defined, and whether it is a model element or an object, a query may have its *formal parameters*. A formal parameter of a query is a typed multiplicity element, whose type is a class or a data type, and whose multiplicity is always exactly one. When a query is evaluated, *actual arguments* must be provided for all formal parameters that have no default value. These actual arguments may affect the result obtained from the query.

When evaluated, a query results in a collection of *tuples*. The tuples in the collection are homogeneous, meaning that they all have the same number and types of *components*, or *values*, each component being (a reference to) an instance of a classifier (class or data type). The multiplicity of each component is often exactly one, meaning that a value in a tuple always refers to an instance. The collection can be ordered or unordered, depending on the definition of the query. Similarly, the collection can have duplicate tuples or represent a set, depending on the definition of the query. The *resulting tuple type* of a query is an ordered array of named typed components of its result.

## Part III: Concepts

---

For example, the following OQL query returns a collection of tuples having the tuple type (c:Course, t:Teacher, n1:Text, tc:Course, n2:Text):

```
SELECT c, t, n1, tc, n2
 FROM Course c, c.students:Teacher t,
 c.name n1, t.taughtCourses tc, tc.name n2
 WHERE n1 = 'Computers for Everybody'
```

Logically, the result of a query can be thought of as a table with homogeneous columns. This means that the result is a collection of rows, whereby each row represents a tuple of values, and where all values in one column are of the same type. This representation fully resembles the results of SQL queries in relational databases, although the values here are references to instances of classifiers, not just simple values of built-in data types.

Note that the notion of parameters and the resulting tuple type and tuple collection is general and independent of how the query is actually specified. Any query, whether a model element or an object, and whether specified in OQL, using a pattern object structure, or in another query language, may have parameters and always returns a collection of tuples of a certain tuple type.

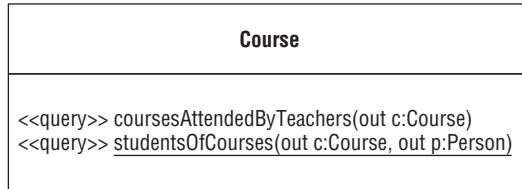
### Section Summary

- ❑ A query may have its *formal parameters*. The type of each formal parameter is a class or a data type, and the multiplicity is always exactly one.
- ❑ When a query is evaluated, *actual arguments* must be provided for all formal parameters that have no default value.
- ❑ When evaluated, a query returns a collection of *tuples*. All tuples in the collection have the same number and types of *components* or *values*, each component being (a reference to) an instance of a class or data type.
- ❑ The *resulting tuple type* of a query is an ordered array of named typed components of its result.

## Queries as Model Elements in OOS UML

Queries as model elements are defined as a special kind of operations of classes and data types tagged with «query», as shown in Figure 12-1. The (input) formal parameters of the query are specified as formal parameters of such an operation having the direction kind *in*. The direction kind is written in front of the parameter's name, and can be omitted, when *in* is assumed.

An operation can be *static*, meaning that it characterizes the classifier instead of every instance of that classifier. A static operation can be invoked for the classifier itself, without stating a specific instance of that classifier. The query of a static query operation does not have implicit formal parameters (that is, it has no parameters other than those stated in the operation signature). Static operations are underlined in UML diagrams, as shown in Figure 12-1 for the operation *studentsOfCourses*.



**Figure 12-1: Notation for queries as model elements. Queries are operations of classifiers tagged with «query».**

If a query operation is not static, it characterizes every instance of that classifier and must be invoked for a particular instance. Such a query has always one implicit formal parameter (not explicitly declared in the operation parameter list) named `this` and with that classifier as its type. When the query operation is invoked for a particular instance of the classifier, the parameter `this` refers to that instance. For the example in Figure 12-1, the query operation `coursesAttendedByTeachers` is non-static.

The result of a query operation is always a collection of tuples. The components of the resulting tuple type are specified as the parameters of the query operation with the direction kind `out`. For the example in Figure 12-1, the resulting tuple type of the query operation `coursesAttendedByTeachers` is `(c:Course)`, while the resulting tuple type of the operation `studentsOfCourses` is `(c:Course, p:Person)`.

A query operation can be invoked as any other operation. A non-static operation must be invoked for a specific instance of the classifier, as in the following example:

```

Course c = ...;
...c.coursesAttendedByTeachers()... // Query operation invocation

```

A static query operation can be invoked for a particular instance of a classifier, in the same manner as a non-static operation, when that instance does not play any specific role in the invocation other than binding the invocation to the invoked static operation. However, it can be also invoked without specifying any instance, as in the following example:

```

...Course::studentsOfCourses()...// Static query operation invocation

```

The result of such an operation is always a collection of tuples that must be accessed in a specific way, because it is not an ordinary type in OOIS UML. For that purpose, the OOIS UML native detail-level language introduces the *tuple type*. The tuple type has a similar meaning as the tuple type in OCL. It is used to define the type of the resulting tuples of evaluated queries. The resulting collection of an invoked query operation is a typed multiplicity element with the multiplicity always set to `*`, and with the type `TupleType`. Following is an example:

```

TupleType(c:Course,p:Person) [*] result = Course::studentsOfCourses();
result->forEach(t) {
 Course c = t.c;
 Person p = t.p;
 ...
}

```

## Part III: Concepts

---

Alternatively, the same can be written more concisely:

```
Course::studentsOfCourses() -> forEach(t) {
 Course c = t.c;
 Person p = t.p;
 ...
}
```

As you can see, the components of a tuple are accessed using the dot notation.

A different approach can be used in other detail-level languages, where the implementation must provide an API for evaluating queries and accessing their results. For example, a conventional OO programming language can use a separate type for storing resulting collections and their tuples:

```
QueryResult res = Course.studentsOfCourses();
for (Tuple t = res.reset(); t!=null ; t=res.next()) {
 Course c = (Course)t.field("c");
 Person p = (Person)t.field("p");
 ...
}
```

The components of a tuple in this case are accessed by invoking the operation `field` of the tuple, with the name of the component provided as a string argument. It is obvious that the binding of a component name to the actual value of the component is performed at run-time. The operation `field` returns a typeless reference to a classifier instance that must be downcast to the actual type of that instance.

The definition of the query operation (that is, the implementation of the query) can be provided in any querying language supported by the environment. OÖIS UML provides two standard means for that, OQL and pattern object structures, as described later in this chapter.

### Section Summary

- ❑ Queries can be modeled as a special kind of operations of class or data type tagged with «query».
- ❑ The formal parameters of a query operation with the direction kind `in` represent the parameters of the query. If a query operation of a classifier is non-static, the query has an implicit parameter `this`.
- ❑ The formal parameters of a query operation with the direction kind `out` represent the components of the resulting tuple type of the query.
- ❑ A query operation is invoked as any other operation. The result of such invocation is always a collection of tuples that can be iterated through.

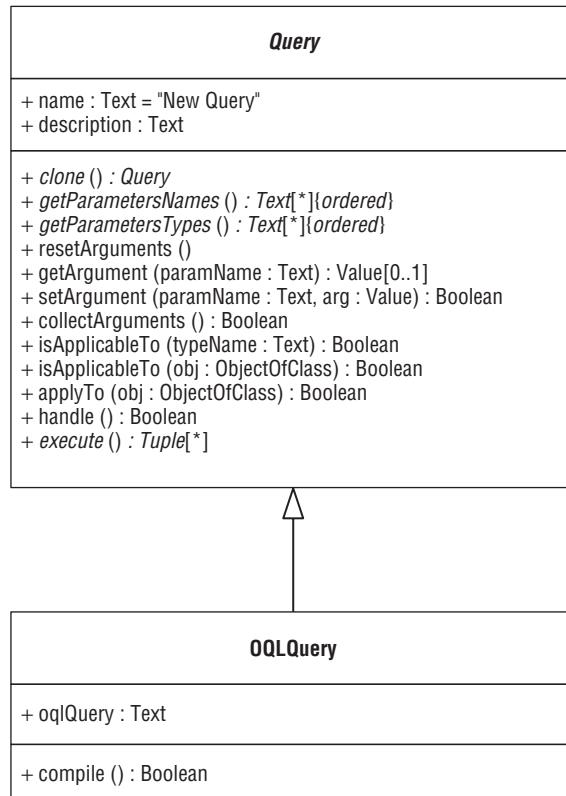
## Queries as Objects in OÖIS UML

The queries as model elements are defined at design time and, depending on the profile implementation, may need recompilation, reconfiguration, or reinstallation of the system when they are created or

updated. However, it is often useful to allow users to create their own queries or import the queries provided by the designers at run-time, without reconfiguring or restarting the system.

For that purpose, the user can create and define queries as ordinary objects in the application's object space and evaluate them at run-time. Such queries are evaluated on an explicit request, either from the GUI, or from a method programmed in a detail-level language. When such a query is executed from the GUI, the GUI may render its result in an appropriate tabular way.

The OOIS UML package `Queries` from the model library provides the classes `Query` and `OQLQuery` that support this functionality. The model of this package is shown in Figure 12-2.



**Figure 12-2: The abstract class `Query` and the class `OQLQuery` from an OOIS UML library package allow you to create queries as ordinary objects in the application's object space.**

The abstract class `Query` generalizes the concept of query as object and provides the features that support some common mechanisms. Additionally, it gathers the common features of queries that are used or redefined in specializing classes.

## Part III: Concepts

---

The properties of the class `Query` have the following meaning:

- `name` — An arbitrary short name that helps the user to identify the query as an object. It does not affect the semantics of the query.
- `description` — An arbitrary description that helps the user to understand the purpose and meaning of the query. It does not affect the semantics of the query.

The operations of this class have the following purpose:

- `clone` — Abstract operation that supports the Prototype design pattern [Gamma, 1995]. Its method should be defined in the derived class to create a new object of that concrete specializing class and return a reference to it.
- `getParametersNames`, `getParametersTypes` — Abstract operations that return ordered collections of names and types (fully qualified names) of the parameters of the query. The specializing classes are responsible for providing the means to extract the parameters of the query. For example, the parameters of an `OQLQuery` are extracted from the OQL specification of the query, and need not be declared explicitly.
- `resetArguments` — Clears the values of all actual arguments.
- `getArgument` — Returns the actual argument for the given parameter name if the argument has already been provided, or `null` if it has not or if the parameter with the given name does not exist.
- `setArgument` — Sets the value of the actual argument for the given parameter name. Returns `true` if successful, or `false` if the parameter with the given name does not exist, or if the argument is of an incompatible type.
- `collectArguments` — Interacts with the user through the GUI to get the values of all actual arguments that have not been provided yet, and sets the values of the actual arguments. Returns `true` if the arguments for all parameters have been provided. In its implementation, it relies on the operations `getParametersNames`, `getParametersTypes`, `getArgument`, and `setArgument`;
- `isApplicableTo(typeName:Text)` — Returns `true` if the query is applicable to an instance of the type given by its fully qualified name. The substitution rule is always assumed. A query is applicable to an object of the given type if any of its parameters accepts that type.
- `isApplicableTo(obj:ObjectOfClass)` — Returns `true` if the query is applicable to the given object. The substitution rule is always assumed. A query is applicable to an object if it is of a type of any of its parameters. For example, this operation can be invoked by the generic GUI to indicate whether a selected or dragged query can be applied to the target object.
- `applyTo` — First checks whether this query is applicable to the given object (if any), by invoking `isApplicableTo`, and returns `false` if it is not. Otherwise, it makes a clone of this query by invoking `clone`, and provides the given object as the actual argument for a parameter that accepts that object to the clone query. If there are more parameters that may accept that object, there might be some implementation-dependent preferences (for example, the parameters `this`, `self`, `input`, `source`, `src`, `target`, `destination`, `dest`, or `dst` may have precedence over the others). Finally, it executes the clone query by invoking `handle` and then deletes the clone. This

operation can be used by the generic GUI to, for example, apply the selected or dragged query to the target object.

- ❑ `handle` — First ensures that actual arguments are provided for all parameters by invoking `collectArguments`. Then invokes `execute` to evaluate the query and get its result. Finally, it renders the resulting collection of tuples in the GUI and returns `true` if successful. This is an application of the Template Method design pattern [Gamma, 1995].
- ❑ `execute` — Abstract operation that actually evaluates the query and returns the resulting collection of tuples. The derived classes should provide the concrete methods for evaluation of different kinds of query specifications. To access the actual arguments, they can invoke `getArgument`.

The classes specializing the class `Query` should define concrete mechanisms for specifying and evaluating queries. The class `OQLQuery` does this for queries specified in OQL. Other user-defined classes can do the same for other kinds of query specifications. The specializing class should define the methods for the operations `clone`, `getParametersNames`, `getParametersTypes`, and `execute`. The class `OQLQuery` also provides a helper operation `compile` that checks the correctness of the OQL query provided in its source form in the attribute `oqlQuery` and returns `true` if the query is correct.

Apart from their use in the generic GUI, queries as objects can be used programmatically, within user-defined methods. For example, the following code in the OOIS UML native detail-level language executes the given OQL query:

```
OQLQuery q = new OQLQuery;
q.oqlQuery = "SELECT c, p FROM Course c, c.students p";
q.execute()->forEach(t) {
 Course c = (Course)t.field("c");
 Person p = (Person)t.field("p");
 ...
}
q.destroy();
```

The generic (or a customized) GUI uses the same approach when it executes queries as objects, as described in Chapter 6. Additionally, the same library provides two commands:

- ❑ `CmdEvaluateQuery` — A command that evaluates the query provided as its sole input parameter.
- ❑ `CmdApplyQueryToObject` — A command that applies the query provided as its source input parameter to the object provided as its target input parameter.

These two commands allow for easy customization of the GUI. For example, evaluating a query by double-clicking on it, or applying a dragged query to the object it is dropped onto is based on these commands.

An implementation can provide additional features of these classes for the convenience of their use. For example, an implementation can provide additional information about which part of the OQL query caused a compilation error, or how the OQL query is compiled to SQL if the latter is used for implementation of queries.

### Section Summary

- ❑ The OOIS UML library package `Queries` contains the classes and commands that allow the user to create and define queries as ordinary objects in the application's object space, and evaluate them at run-time on an explicit interactive or programmatic request.
- ❑ The abstract class `Queries::Query` generalizes the concept of query as an object and provides the features that support some common mechanisms. Additionally, it gathers the common features of queries that are used or redefined in specializing classes.
- ❑ The classes that specialize the class `Queries::Query` should define concrete mechanisms for specifying and evaluating queries. The class `Queries::OQLQuery` does this for queries specified in OQL.
- ❑ The commands `Queries::CmdEvaluateQuery` and `Queries::CmdApplyQueryToObject` allow for easy customization of the GUI.

## Object Query Language

OQL is one standard language predefined in OOIS UML for defining queries. It is a language for posing queries on object space, with the syntactical and semantic flavor of SQL. OOIS UML uses a dialect of OQL that is adapted to the semantics of OOIS UML and closer to the semantics and notation of OCL. It supports only `SELECT` queries for reading the object space, without side effects (that is, without modifications of the object space). The basic form of an OQL `SELECT` query is the same as in SQL:

```
SELECT projectionClause FROM navigationClause WHERE selectionClause
```

Following are the main features of OQL in OOIS UML and its differences from SQL:

- ❑ The navigation in the `FROM` clause is specified in terms of objects of classes and their properties (attributes and owned association ends) instead of tables, records, and fields. The navigation over links replaces the need for joins over foreign keys in SQL.
- ❑ Generalization/specialization is supported with its full semantics. Inherited properties of classes can be accessed as usual, type specialization (downcasting) is supported, and substitution is always assumed (queries always return all direct and indirect instances of the specified classes).
- ❑ Queries can return values of attributes, slots as collections of linked objects or values, as well as references to objects.
- ❑ Queries can be parameterized and nested.
- ❑ OQL queries can be executed programmatically, from methods, or used as ordinary objects in the object space.

All these features and other issues are described in the following sections.

## Semantics of OQL Queries

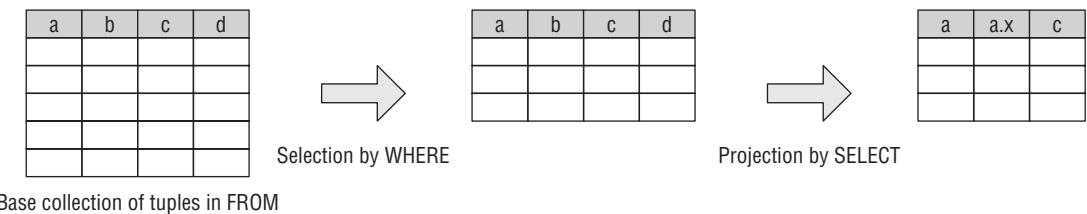
The basic form of an OQL SELECT query is the same as in SQL:

```
SELECT projectionClause FROM navigationClause WHERE selectionClause
```

For example, following is an OQL query for the Easylearn school model from Chapter 6:

```
SELECT p.name, p.email
 FROM Course c, c.students p
 WHERE c.name = 'Computers for Everybody'
```

Therefore, an OQL query has three parts: the navigation clause (*navigationClause* following FROM), the selection clause (*selectionClause* following WHERE), and the projection clause (*projectionClause* following SELECT). The semantics of each of the clauses and their combination is schematically presented in Figure 12-3.



**Figure 12-3: A schematic representation of the semantics of OQL queries. The navigation clause in `FROM` designates a base collection of tuples. The selection clause in `WHERE` filters the base collection of tuples to include only those tuples that satisfy a Boolean condition. The projection clause in `SELECT` retains the number of tuples in the collection, but removes or adds some components (columns).**

The navigation clause specifies navigation over the object structure. The navigation designates a type of tuples of values, each value representing a placeholder for an instance of a certain classifier. The navigations over properties that are association ends also introduce implicit constraints over the returned tuples. The constraints enforce that the objects referred to by the placeholders are linked by the corresponding associations. In the previous example, each tuple designated by the navigation clause has two values named *c* and *p*, where *c* is a placeholder for an object of *Course*, while *p* is a placeholder for an object of *Person* (the class at the navigated end *students*). The implicit restriction introduced by the navigation clause is that the objects referred to by *p* and *c* are linked, so that *p* is an element of the collection designated by *c.students*.

In short, the navigation clause defines a collection of tuples of values, one value for each term in the clause (the terms are separated by commas). The value in the tuple is named by the alias provided in the term, and can be referred to by that name in the clauses of the same query. Therefore, the navigation clause also introduces the names of the values in the tuple, the scope of the names being the same query.

A value in the tuple always refers to an instance of a classifier, which is the type of the value. If the type is a class, the value refers to an object of a class, possibly linked to other objects. If the type is a data type, the value refers to an instance of that data type (for example, as an attribute value of a certain object).

## Part III: Concepts

---

The collection of the tuples designated by the navigation clause consists of a sub-collection of a Cartesian product of all existing instances of the types, such that the values (referring to instances of classifiers) implied by the terms satisfy the following constraint: If a term in the clause specifies navigation over a property (an attribute or an association end), the value implied from it must be an element of the collection designated by the referred property.

For the previous example, the resulting collection of tuples defined by the navigation clause is the collection of tuples  $(c, p)$ , where  $c$  is an object of `Course`,  $p$  is an object of `Person` (the class at the navigated end `students`), and  $p$  is an element of  $c.\text{students}$ . On the other hand, consider the tuple defined by the following `FROM` navigation clause:

```
Course c, c.students:Teacher t, c.name n1, t.taughtCourses tc, tc.name n2
```

This consists of the named elements shown in the following table:

| Name | Type    | Additional Implied Constraint |
|------|---------|-------------------------------|
| c    | Course  | None                          |
| t    | Teacher | t is in c.students            |
| n1   | Text    | n1 is in c.name               |
| tc   | Course  | tc is in t.taughtCourses      |
| n2   | Text    | n2 is in tc.name              |

As a result, the navigational clause defines a collection of all tuples retrieved from the object space satisfying the described criteria. Typically, the collection is a bag because the collections designated by the navigation terms can be bags, and the navigations can be combined over several properties (for example,  $t.\text{taughtCourses}.\text{teachers}.\text{taughtCourses}$ ). Each value in the tuple is of the type defined in the term, and is always of cardinality of exactly one (that is, it is never a null).

The selection clause after `WHERE` further constrains the resulting collection of tuples by specifying an expression (see Figure 12-3). The expression must return a Boolean result. The expression can refer to all the names of the values in the tuple defined by the navigation clause. The expression can also use the operations (possibly invoked by infix or prefix operators) on the values in the tuple or on their slots. The expression is evaluated for every tuple in the collection designated by the navigation `FROM` clause. The resulting collection of tuples is reduced by the selection `WHERE` clause to all these and only these tuples for which the expression evaluates to `true`.

Finally, the projection clause following `SELECT` simply projects the resulting tuple to another tuple by keeping some of the values in the tuple and removing the others (see Figure 12-3). Think of the collection of the tuples resulting from the rest of the query (the navigation and selection clauses) as a table with columns identifying values and rows representing the selected tuples. Then, the projection clause simply removes some columns, while keeping the others. All rows are preserved (no duplicates are removed, unless the `DISTINCT` specifier is used). Optionally, the projection clause renames some columns by providing different names (aliases) for some values in the tuples.

As an extension the value can also be a multi-valued slot of a classifier instance from the tuple, in which case it represents an entire collection as a value in the tuple. For example, consider the result of the following query:

```
SELECT c, c.name courseName, p, p.name personName, p.attendedCourses
 FROM Course c, c.students p
 WHERE c.name = 'Computers for Everybody'
```

The result of this is a collection of tuples with the values of types given in the following table:

| Name            | Type[Multiplicity] |
|-----------------|--------------------|
| c               | Course[1]          |
| courseName      | Text[1]            |
| p               | Person[1]          |
| personName      | Text[1]            |
| attendedCourses | Course[*]          |

Note that a value referring to a slot of an object is of the type and multiplicity of the slot's property.

Ultimately, the result of the entire query is a collection (in a general case, a bag by default, or a set, if the `DISTINCT` specifier is used) of tuples of values. Each element of the tuple has its name (unique within the scope of the tuple), its type (defined by the term in the navigation clause from which the value was projected), and its multiplicity.

The details about the possible contents and semantics of these three clauses are described in the sections that follow.

### Section Summary

- ❑ The basic form of an OQL SELECT query is the same as in SQL:

```
SELECT projectionClause FROM navigationClause WHERE
selectionClause
```

- ❑ The navigation clause designates a collection of tuples. Each value of the tuple corresponds to one term in the clause. Terms are separated by commas. Each value in the tuple is of the type designated by the term. The collection of tuples is obtained by a Cartesian product of all existing instances of the types for each component of the tuple, narrowed by constraints implied from the navigation in terms. If a term defines a navigation over property (an attribute or association end), the value in a tuple must be the value from the collection designated by the referred slot.

*Continued*

- ❑ The selection clause selects a sub-collection of tuples defined by the navigation clause by a Boolean expression applied to every tuple from the original collection. Only the tuples for which the expression evaluates to `true` are retained in the resulting collection.
- ❑ The projection clause maps the resulting collection into another collection by renaming, removing, or adding values to the tuples, or collapsing the collection into a set (`SELECT DISTINCT`).

## ***Navigation in the FROM Clause***<sup>1</sup>

The navigation clause behind `FROM` in OQL queries defines a tuple type. When the query is executed, the navigation clause conceptually results in a collection of tuples as instances of that tuple type. This resulting collection of tuples will be referred to as the *base collection of tuples* of the query.

The navigation clause is a comma-separated list of terms. Each term designates a component (value) in the tuple type defined by the navigation clause. When the query is executed, each term conceptually results in a collection (in a general case, it is a bag) of values of the type determined by the term. The base collection of tuples is obtained as a Cartesian product of the collections designated by the terms, filtered by the constraints implied from the terms.

For example, consider the following navigation clause, as a list of terms:

```
Course c, c.students:Teacher t, c.name n1, t.taughtCourses tc, tc.name n2
```

This defines the following tuple type:

```
(c:Course, t:Teacher, n1:Text, tc:Course, n2:Text)
```

Each kind of term allowed in the navigation clause thus specifies the following:

- ❑ The name of the component (value) in the tuple type
- ❑ The type of the values in the collection designated by the term (that is, the type of the component in the tuple type)
- ❑ An optional implicit constraint over the entire base collection of tuples

The multiplicity of each value in a tuple is always exactly one. The ordering of the collections designated by each term is irrelevant because the result of the entire clause is a Cartesian product in a general case, and is, thus, unordered.

Each kind of term defines a default name of the component. If a term has no navigation, the default name is the name of the class in the term. Otherwise, if a term specifies navigation to a property, the default

---

<sup>1</sup>This is advanced reference material that can be skipped on first reading.

name is the name of the property. The default name can be replaced by the optional alias provided in the term. In that case, the name of the component is the name defined by the alias. The name of the component must be unique within the navigation clause. Therefore, if several terms have the same default names, aliases must be used to resolve ambiguities. An alias can be preceded by an optional keyword AS.

The rest of this section describes two different kinds of terms allowed in the navigation clause and the implicit constraints that they define, as well as the collections they designate when evaluated.

The first kind of term is a simple class reference. The term simply specifies a class and an optional alias. An example is the term Course c in the clause given earlier. The type of the value designated by such a term is that class. The designated collection consists of all existing objects of that class. The collection is always unique (that is, it is a set of all objects of the class). There is no implicit constraint introduced by such a term.

The second kind of term is a term that defines navigation over properties of classifier instances. The navigation must start with a name defined by another term before this term.<sup>2</sup> For example, the following terms define navigations:

```
c.students:Teacher t, c.name n1, t.taughtCourses tc, tc.name n2
```

A term can include a navigation chain over several properties, as shown in the following examples:

```
t.taughtCourses.students
t.taughtCourses.students.attendedCourses
t.taughtCourses.students.attendedCourses.name
t.taughtCourses.students.attendedCourses.students
t.taughtCourses.students.attendedCourses.students.name
```

The number of hops over properties in a navigation chain can be arbitrary, but finite.

The type of the term is that of the last property in the navigation chain. Optionally, the type can be specialized to a subtype of the property's type with the downcasting colon operator:

```
t.taughtCourses.students:Teacher
```

The downcasting operator can appear anywhere in the navigation chain. In that case, the further navigation may refer to properties of the specializing type. Consider the following example:

```
t.taughtCourses.students:Teacher.taughtCourses.students:Teacher.title
```

If the navigation in the term has only one hop (that is, is a navigation with one dot operator only), the uniqueness of the collection of values designated by the term is equal to the uniqueness of the navigated property. Otherwise, the designated collection is always non-unique.

---

<sup>2</sup>As a consequence of this rule, there cannot be cyclic references between terms — that is, there cannot be two or more terms, each of which starts its navigation by the name of the other term in the cycle. An implementation may opt to relax this rule so that a name that starts navigation does not need to be defined before that navigation term, but just somewhere else in the clause. That is, the terms can be listed in an arbitrary order. However, the rule of acyclic references to the names defined in the terms must still hold.

## Part III: Concepts

---

The collection of values designated by a navigation term is obtained by a set of nested iterations through all collections in the navigation chain. For example, the collection designated by this term is obtained by the following set of nested iterations:

```
t.taughtCourses.students.attendedCourses
```

For each instance  $t$  of the type specified by the term that defines  $t$  (Teacher in this case), for each element  $e_1$  in  $t.taughtCourses$ , for each element  $e_2$  in  $e_1.students$ , for each element  $e_3$  in  $e_2.attendedCourses$ , return  $e_3$ . It is obvious that if the chain has more than one navigation hop, the resulting collection can be non-unique (that is, a bag) in a general case because the resulting values can be multiplied.

In addition, such navigation introduces an implicit constraint over the base collection of tuples. The value that is the iterator of the topmost iteration in the set of nested iterations must be equal to the value in the tuple with the name that starts the navigation. For the given example, the collection of elements  $e_3$  designated by the given term must be obtained as described, where the value of  $t$  in this navigation must be the same as the value named  $t$  in the entire resulting tuple.

Because a `SELECT` query results in a collection of tuples of named values, the base collection of tuples can be defined by a nested `SELECT` query. In that case, the semantics of the enclosing query remain the same — just the base collection of tuples is produced as a result of the nested query, instead of being designated and computed from the navigation clause. The base collection of tuples is further simply filtered and projected by the enclosing query. Consider the following example:

```
SELECT p, p.name, p.email
 FROM (
 SELECT p, p.name, p.attendedCourses
 FROM Course c, c.students p
 WHERE c.name = 'Computers for Everybody'
)
 WHERE p.name = 'John Smith'
```

As you can see, the navigation clause in OQL supports only navigation over properties of classes and data types. It does not support navigation over N-ary association ends and association ends owned by associations, nor does it support operation calls.

### Section Summary

- ❑ The navigation clause designates a collection of tuples as its result, referred to as the *base collection of tuples* of the query. This collection is unordered and non-unique, in general.
- ❑ The navigation clause is a comma-separated list of terms. Each term designates a collection of values of the type defined by the term. The value in the tuple can be referred to by the name defined by the term. The ordering of the collection is irrelevant.

- If the term provides an alias, the alias is the name of the value in the tuple. If the term does not provide an alias, the name is the default name defined by the term. The name must be unique within the navigation clause.
- A term can be of one or two types:
  - A reference to a class
  - Navigation over properties of classifiers (possibly with multiple hops)
- Each term designates a collection of values. The base collection of tuples is obtained as a Cartesian product of all these collections of values, constrained by implicit constraints introduced by navigation terms.
- For a term that refers to a class:
  - The default name is the name of the class.
  - The type of the designated values is that class.
  - The designated collection is the set of all existing objects of that class.
- For a navigation term:
  - The default name is the name of the last property in the navigation chain.
  - The type of the designated values is the type of the last property in the navigation chain; downcasting (that is, type specialization) can appear anywhere in the navigation chain.
  - The designated collection is obtained by a set of nested iterations through collections defined by the navigation chain.
  - The term introduces an implicit constraint over the base collection: the value of the iterator of the topmost iteration must be equal to the value in the entire tuple with the name that starts the navigation.

## Selection in the WHERE Clause<sup>3</sup>

The selection clause behind WHERE filters the base collection of tuples to a sub-collection (possibly equal to the base collection) of those tuples for which the Boolean expression specified in the clause results to true (see Figure 12-3). The resulting collection of tuples is of the same type — that is, it has the same named values as the base tuple type designated by the navigation clause.

The selection clause behind WHERE is a Boolean expression that can refer to the named values in the base tuple designated by the navigation clause. It specifies the operations upon those values and ultimately

---

<sup>3</sup>This is advanced reference material that can be skipped on first reading.

## Part III: Concepts

---

returns a Boolean value. However, the concrete syntax and the set of supported operations are left to the implementation, and are actually the most variable and implementation-dependent part of OQL queries.

On the one hand, an implementation can support WHERE expressions with a full compliance to the OOIS UML semantics and syntax flavor. Actually, an implementation can allow OCL Boolean expressions in WHERE clauses, written in the context of the query (that is, referring to named values from the base tuple). In general, a full-fledged implementation could support the following (the list is not limited):

- ❑ Navigation (including multi-hop) over the properties of instances referred to by the values in the tuple
- ❑ Invocation of operations of instances
- ❑ Infix operators that result in corresponding operation invocations as in OCL
- ❑ A full set of operations on built-in types, such as operations on the following:
  - ❑ **Strings (textual types)** — Relational operators (in lexicographical order), inexact string-matching operator (LIKE in SQL), concatenation operations, and so on
  - ❑ **Numeric types (integers, real numbers)** — Relational operators, arithmetic operators, and so on
  - ❑ **Boolean** — All usual logic operators (AND, OR, NOT, XOR)
- ❑ Operations on collections of values returned by other operations, such as those defined in OCL or in the OOIS UML native detail-level language
- ❑ Operations on collections of tuples, such as the operator IN in SQL, which tests whether the tuple that is the left operand is an element of the collection of tuples that is the right operand
- ❑ Nested OQL queries that define collections of tuples as operands of such operations, as in the following:

```
SELECT p.name, p.email
 FROM Course c, c.students p
 WHERE c IN (
 SELECT tc
 FROM Teacher t, t.taughtCourses tc
 WHERE t.name LIKE '%Smith%'
)
```

The concrete syntax and semantics of such expressions is implementation-dependent, although it should have the flavor of OCL and/or SQL.

However, a less ambitious implementation can have a limited support of these elements. For example, it is reasonable to expect that an implementation does not support operation invocations, multi-valued attributes, collection operations, or operations with attributes of user-defined data types.

### Section Summary

- ❑ The selection clause behind WHERE filters the base collection of tuples to a sub-collection (possibly equal to the base collection) of those tuples for which the

Boolean expression specified in the clause results to `true`. The resulting collection of tuples is of the same type as the base collection of tuples.

- The selection clause behind `WHERE` is a Boolean expression that can refer to the named values in the base tuple designated by the navigation clause.
- The concrete syntax and semantics are left to the implementation, but should have the flavor of OCL and/or SQL. For example, OCL Boolean expressions can be used.

## ***Projection in the SELECT Clause<sup>4</sup>***

The projection clause behind `SELECT` keeps all the tuples in the collection obtained from the selection, but may do the following (see Figure 12-3):

- Remove some components (columns) from the tuple type
- Add new components to the tuple type
- Rename some components of the resulting tuple type by providing aliases
- Generalize (upcast) the types of some components of the resulting tuple type

Therefore, the projection changes the type of the resulting collection of tuples, but retains the number of tuples in the collection, because the resulting collection corresponds one-to-one to the collection obtained from selection. The ultimate result of the entire query is, thus, a collection of tuples of the type defined by the projection clause.

The projection clause is basically a comma-separated list of terms. Each term defines one and only one component (value) in the resulting tuple type.

Each term can optionally introduce an alias, which is the name of the corresponding component in the resulting tuple type. If the alias is omitted, the name of the component is the default name defined for that kind of term. The names of the components must be unique in the resulting tuple type. If several terms have the same default names, aliases must be used to resolve ambiguities.

A term can be of one of the following types:

- The name of the component in the tuple obtained from the selection. The default name is that very name. The value in the tuple is that very value (a reference to an instance of a classifier). For example, the terms `c`, `n`, and `p` in the following query are of this kind:

```
SELECT c, n.courseName, p, p.name personName, p.attendedCourses
 FROM Course c, c.name n, c.students p
 WHERE n = 'Computers for Everybody'
```

- One-hop navigation to a slot of an attribute or owned association end of a classifier instance referred to by a value from the tuple resulting from the selection. The default name is the name

---

<sup>4</sup>This is advanced reference material that can be skipped on first reading.

## Part III: Concepts

---

of the navigated property. For example, the terms `p.name` and `p.attendedCourses` are of that kind:

```
SELECT c, n.courseName, p, p.name personName, p.attendedCourses
 FROM Course c, c.name n, c.students p
 WHERE n = 'Computers for Everybody'
```

Such a term designates a typed multiplicity value of the type, multiplicity, ordering, and uniqueness as the property specified by the navigation. For the previous example, the term `p.name` is of type `Text[1]`, whereas `p.attendedCourses` is of type `Course[*]`.

- A special term with navigation to an asterisk (`aName.*`) is an equivalent replacement for the entire list of terms navigating to all properties of the type of the given value. For example, the term `p.*` in the following query is a substitute for a list of terms navigating to all properties of Person:

```
SELECT c, p, p.*
 FROM Course c, c.name n, c.students p
 WHERE n = 'Computers for Everybody'
```

When the tuple type obtained from the selection has only one value, a sole asterisk can be used. It replaces the list of properties of that single value. For example, the following query returns all properties of Course:

```
SELECT *
 FROM Course c
 WHERE c.name = 'Computers for Everybody'
```

The type of a term in the projection clause can be explicitly converted to a more generalized type (upcast), if that generalized type is desired as the type of the component in the resulting tuple type. For example, although the type of the term `t` in the base tuple type in the following query is `Teacher`, the type of the term `t` in the resulting tuple type is `Person`, as the generalization of `Teacher`:

```
SELECT c, t:Person
 FROM Course c, c.teachers t
 WHERE c.name = 'Computers for Everybody'
```

Such upcasting may be desired when the results of two or more queries must be homogenized to have the same tuple type so that they can be gathered in a union.

### Section Summary

- The projection clause behind `SELECT` keeps all the tuples in the collection obtained from the selection, but may do the following:
  - Remove some components (columns) from the tuple type

- Add new components to the tuple type
- Rename some components of the resulting tuple type by providing aliases
- Generalize (upcast) the types of some components of the resulting tuple type.
- The projection clause is basically a comma-separated list of terms. Each term defines one (and only one) component (value) in the resulting tuple type.
- A term can be of one of the following kinds:
  - The name of the value in the tuple obtained by the selection
  - One-hop navigation to a slot of an attribute or owned association end of a classifier instance referred to by a value from the tuple obtained by the selection
  - A special term with navigation to an asterisk (`aName.*`) is an equivalent replacement for the entire list of terms navigating to all properties of the type of the given value.

## ***Ordering and Grouping***

An OQL implementation may support other constructs from SQL applicable to object queries. Some common ones are ordering and grouping with cumulative functions.

The collection of tuples resulting from the entire query can be ordered by some of its values, as in SQL. Consider the following example:

```
SELECT c, p, p.*
 FROM Course c, c.students p
 WHERE c.name = 'Computers for Everybody'
 ORDER BY p.name ASC
```

The `ORDER BY` clause can specify a list of terms from the projection `SELECT` clause. The types of those terms must have an ordering relation defined. Primitive textual and numeric types have such a relation. The resulting collection of tuples is ordered in the ascending or descending order of the specified values, depending on the qualifier `ASC` (default) or `DESC`, respectively, just as in SQL.

As in SQL, a cumulative function can be applied to groups of tuples having the same specified value. For example, the following query returns a collection of triples `(aPerson, totalPrice, avgPrice)` with total price and average price of all courses attended by each Person:

```
SELECT p aPerson, Sum(c.price) totalPrice, Avg(c.price) avgPrice
 FROM Person p, p.attendedCourses c
 GROUP BY p
```

### Section Summary

- ❑ The resulting collection of tuples can be ordered by some of its values.
- ❑ A cumulative function can be applied to groups of tuples having the same value of the specified component.

## Unions

The results of two or more SELECT queries with exactly the same resulting tuple types can be gathered into a union, so that the result of the union includes all tuples from the gathered queries. For example, the following union returns a collection of all Persons that teach the course “Computers for Everybody” or attend at least one Course:

```
SELECT t:Person p
 FROM Course c, c.teachers t
 WHERE c.name = 'Computers for Everybody'
DISTINCT UNION
SELECT DISTINCT p
 FROM Person p, p.attendedCourses
```

All queries in a union must have identical resulting tuple types — with the same number and order of components of the same types and names. Upcasting and aliasing is typically used to cast types and names of terms for that purpose. The result of the union is a collection of tuples having that same tuple type.

The resulting collection is a set (with distinct elements) if the keyword DISTINCT is used before the keyword UNION, as in the previous example. Otherwise, if the keyword DISTINCT is omitted, the resulting collection is a bag obtained by concatenating the collections resulting from the queries. For the previous example, if the keyword DISTINCT were omitted, a Teacher that teaches the named Course and also attends some Courses would appear several times in the resulting union.

### Section Summary

- ❑ The results of two or more SELECT queries with exactly the same resulting tuple types (having the same number and order of components with the same names and types) can be gathered into a union, so that the result of the union includes all tuples from the gathered queries.
- ❑ If the word DISTINCT is used for a union, the resulting collection is a set. Otherwise, it is a simple concatenation of the collections resulting from the gathered queries.

## Parameterization and Nesting<sup>5</sup>

The parameters of a query whose specification is given in OQL can be referred to from the WHERE clause of the OQL specification. A parameter is referred to by its name.

If an OQL query is used to specify a query operation in the model, it is the method of that operation. Consequently, it is embedded in the operation as a namespace, and the parameters of that operation can be accessed from the query. On the other hand, if the OQL query is defined within a query as an object, parameters are defined implicitly, and parameter names must therefore be enclosed in # characters. The implementation should allow # characters to enclose a formal parameter name even inside the methods of query operations. This is why parameter names in OQL queries will always be enclosed between two # characters in the examples that follow.

For example, if the following query has a parameter named `courseName` of type `Text`, the query returns a collection of students of the `Course` with names like the one specified with that parameter:

```
SELECT p
 FROM Course c, c.students p
 WHERE c.name LIKE #courseName#
```

Except to call other nested queries (as explained later), the parameters can be referred to only from the WHERE clause of an OQL query. They take part in the expressions in that clause, and must obey the type conformance rules for the expressions. When the query is evaluated, the actual argument is provided for each formal parameter of the query. Then the value of the actual argument takes part in the evaluation of the expressions as usual.

Although parameters can be referred to from the WHERE clause only, they may indirectly affect the base tuple collection if needed. For example, if the navigation in the FROM clause should start from an object that is a parameter of the query, you can accomplish this by constraining the base term in the WHERE clause:

```
SELECT c, p
 FROM Course c, c.students p
 WHERE c = #course#
```

A part of the FROM clause can be a reference to another query already defined in the model. For example, let the static query operation `Course:::studentsOfCourse` be defined as given previously. As you can see, that query has one parameter named `course` of type `Course`, and has the resulting tuple type (`c:Course, p:Person`). Another query can use that resulting tuple type as part of its base tuple type as follows:

```
SELECT c1, c2
 FROM Course c1, Course:::studentsOfCourse(c1), p.attendedCourses c2
 WHERE c1.name LIKE #courseName#
```

The enclosing query passes the actual argument `c1` to the nested query, so that the nested query returns the tuples (`c, p`), where `p` is a student of `c` and `c` is equal to `c1`. The enclosing query then navigates over

<sup>5</sup>This is advanced material that can be skipped on first reading.

## Part III: Concepts

---

`p` of each such tuple to the attended Courses of `p`, referring to them as `c2`. Note that the nested query introduces its resulting terms (`c` and `p`, in this case) into the base tuple type of the enclosing query, so that the `FROM` clause of the enclosing query can navigate further using these terms.

If the query operation is non-static, the implicit parameter `this` is also available. For example, if the query operation `Course::studentsOfCourse` has no parameters, is not static, and is defined as follows:

```
SELECT c, p
 FROM Course c, c.students p
 WHERE c = #this#
```

then the enclosing query can be defined as follows:

```
SELECT c1, c2
 FROM Course c1, c1.studentsOfCourse(), p.attendedCourses c2
 WHERE c1.name LIKE #courseName#
```

In general, the actual arguments passed to a nested query can be the following:

- Literals
- Formal parameters of the enclosing query (including the implicit parameter `this`)
- Names of the terms defined before the reference to the nested query (as in the previous example).

Of course, the type of every argument must conform to the type of the corresponding formal parameter of the nested query.

The nested query introduces the terms of its resulting tuple type into the base tuple type of the enclosing query, just as if those terms were explicitly defined in the enclosing query's `FROM` clause. The enclosing query's `FROM`, `WHERE`, and `SELECT` clauses can thus refer to those names as usual. Consequently, the names of such implicitly introduced terms must not conflict with the names of other terms in the base tuple type of the enclosing query, either explicitly defined or also implicitly introduced by other nested queries.

When evaluated, the nested query contributes to the collection of tuples of the enclosing query as follows. The resulting tuple type of the nested query takes part in the base tuple type of the enclosing query and the implied Cartesian product as if its terms were explicitly defined in the base tuple type, but with the implicit constraints added to the selection criteria:

- All those (and only those) tuples that are elements of the resulting collection of the nested query will take part in the base tuple collection of the enclosing query.
- The formal parameters of the nested query take the values of the provided actual arguments for each tuple in the base collection.

The support for recursive query nesting is left to the implementation, and is not required by the profile. Note that even though queries do not support conditional constructs (alternatives, `if-then-else` constructs), recursions may still be finite and thus allowable — a recursive call can also be polymorphic, whereby one method of a polymorphic query operation may include a recursive invocation, while another may always return an empty collection (the recursion tail).

### Section Summary

- ❑ The parameters of a query can be referred to from the WHERE clause of the OQL specification.
- ❑ The FROM clause of a query can invoke a nested query, passing actual arguments to it.

## Inline OQL Queries

The detail-level language can support evaluating OQL queries directly, without specifying the queries in the model or creating query objects. The language can provide a means to execute a query whose OQL specification is given as a string, and then to iterate through the resulting collection of tuples and access each of its components. Such queries are called *inline* queries.

The result of an inline query is used as the result of any other query defined in the model. However, because an inline query is provided dynamically and its OQL specification can even be evaluated at run-time, the structure of its resulting tuple type (that is, the concrete components and their types) is also determined dynamically, when the query is evaluated. The resulting collection of an inline query is a typed multiplicity element with the multiplicity always set to `*`, with the type `Tuple`, and uniqueness and ordering determined dynamically, depending on the uniqueness (existence of `DISTINCT` modifier) and ordering (existence of `ORDER BY` modifier) of the query.

Consider the following example:

```
 Tuple[*] result =
 OQLQuery::evaluateOQL("SELECT c, p FROM Course c, c.students p");
 result->forEach(t) {
 Course c = (Course)t.field("c");
 Person p = (Person)t.field("p");
 ...
 }
```

As you can see, the components of a tuple are accessed by invoking the operation `field` of a tuple, with the name of the component provided as a string argument. It is obvious that the binding of a component name to the actual value of the component is performed at run-time. The operation `field` returns a typeless reference to a classifier instance that must be downcast to the necessary type of that instance.

A similar approach can be used in other detail-level languages, where the implementation provides an API for evaluating inline queries, as shown in the following example:

```
QueryResult res =
 OQLQuery.evaluateOQL("SELECT c, p FROM Course c, c.students p");
 for (Tuple t = res.reset(); t!=null ; t=res.next()) {
 Course c = (Course)t.field("c");
 Person p = (Person)t.field("p");
 ...
 }
```

## Part III: Concepts

---

For easier parameterization of queries, the find-and-replace features of strings can be used, if supported by the string types in the detail-level language. Following is an example:

```
String oqlQuery =
 "SELECT c, p FROM Course c, c.students p WHERE c.name like `#courseName#`";
...
String aCourseName = ...;
oqlQuery.replace(`#courseName#`, aCourseName);
QueryResult res = OQLQuery.evaluateOQL(oqlQuery);
```

### Section Summary

- ❑ The detail-level language can support evaluating OQL queries directly, without specifying the queries in the model or creating query objects. Such queries are called *inline* queries.
- ❑ The OQL specification of an inline query is provided at run-time as a string, and its resulting collection of tuples can be iterated through. The components of those tuples can then be accessed dynamically.

## Pattern Object Structures

Queries can be defined using *pattern object structure specifications* that can be presented diagrammatically. Pattern object structure specifications can be designed by demonstration, too.

### Pattern Object Structure Specifications

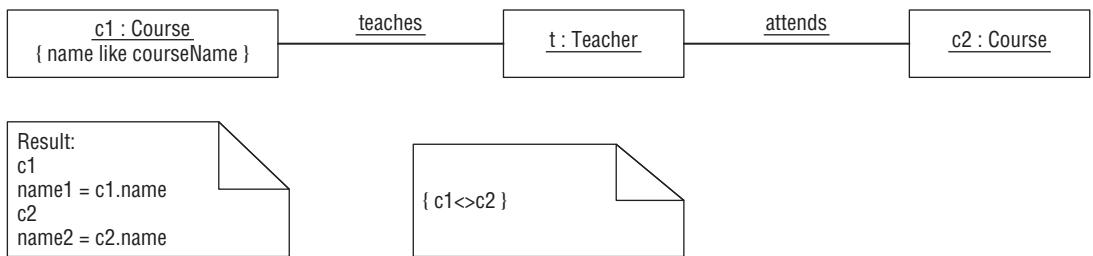
Queries in OOIS UML define searches for certain object structures (that is, for sub-graphs of the entire object space). They can be thought of as definitions of structural *patterns* that should be matched during the search. The patterns include some prototyping objects (representing “placeholders” for instances of certain classes) and prototyping links (representing the requirements that the objects that would match the pattern must be linked in a certain manner). Finally, a query may specify additional selection criteria that may further constrain the pattern matching. The criteria may be expressions that refer to the attribute values of the objects participating in the object structure.

OQL queries (or, more precisely, their `FROM` and `WHERE` clauses) are just one way to describe such pattern object structures. OQL uses textual expressions to describe pattern object structures. Obviously, such textual expressions are rather indirect representations of pattern object structures.

This is why OOIS UML provides a means that directly corresponds to the very nature of pattern object structures — *pattern object structure specifications* can be used to define queries. These specifications are UML object specifications that have the formal pattern matching semantics instead of illustrative semantics. As with all other object specifications, pattern object structure specifications can be depicted in UML object diagrams.

Figure 12-4 shows the UML object diagram for the pattern object structure specification of a simple query for the Easylearn school example. Briefly stated, this specification is semantically equivalent to the following OQL query:

```
SELECT c1, c1.name name1, c2, c2.name name2
 FROM Course c1, c1.teachers t, t.attendedCourses c2
 WHERE c1.name like #courseName# and c1<>c2
```



**Figure 12-4: The UML object diagram for the pattern object structure specification of a simple query**

As you can see, the specification directly shows how the sought-for objects should be linked. The objects in the specification represent prototypical objects, “placeholders” for concrete objects searched for in the actual object space, while the links represent the requirements that the objects that would match the pattern have to be linked accordingly. Additional conditions can be specified as constraints in the scope of every object specification, or in the scope of the very pattern object structure specification, shown in the diagram for the specification within the note symbol in Figure 12-4. Finally, the binding of the query’s output parameters that form the components of the resulting tuple type to the object specifications or their attributes must be defined. The binding is documented in the note symbol in the diagram in Figure 12-4.

In short, pattern object structure specifications are UML object specifications that can consist of the following elements only:

- ❑ Object specifications, which represent prototypical objects (that is, placeholders for objects to be searched for).
- ❑ Link specifications, which introduce requirements that the objects searched for should be linked in a specific way.
- ❑ Constraints, defined in the scope of object specifications or the entire pattern object structure specification. Only those sub-graphs of the object space that satisfy all these constraints will match the pattern. The constraints may refer to the object specifications, their properties, or formal parameters of the enclosing query.
- ❑ Invocations of other nested queries defined in the model, with optionally passing actual arguments to them.

These elements will be described in more detail in the discussions that follow.

### Object and Link Specifications

An object specification within a pattern object structure specification may have its name and class only. The name can be omitted, and is needed only if it is to be referred to from the constraints attached to the specification itself. The class of the object is mandatory, but it may be abstract. Note that an object specification represents a placeholder for a concrete object existing in the object space, while its class represents an implicit constraint on that concrete object to be an instance of that class. This is why the class can be abstract, while the concrete objects that will match the pattern will be direct instances of concrete derived classes.

Link specifications introduce implicit constraints that the objects that match the object specifications have to be linked accordingly.

The object and link specifications designate a collection of tuples analogous to the base tuple collection in OQL queries. For each object specification, there is one (and only one) component in the tuple type defined by the pattern object structure specification. When the query is executed, the tuple collection will consist of all those and only those tuples that satisfy the rules as follows:

- For each object specification, the corresponding value in the tuple is an object of the class specified in the object specification.
- For each link specification, the values in the tuple that correspond to the object specifications at the link specification's ends are linked by a link of the association specified in the link specification, whereby the objects at the link's ends play the corresponding roles as specified in the link specification.

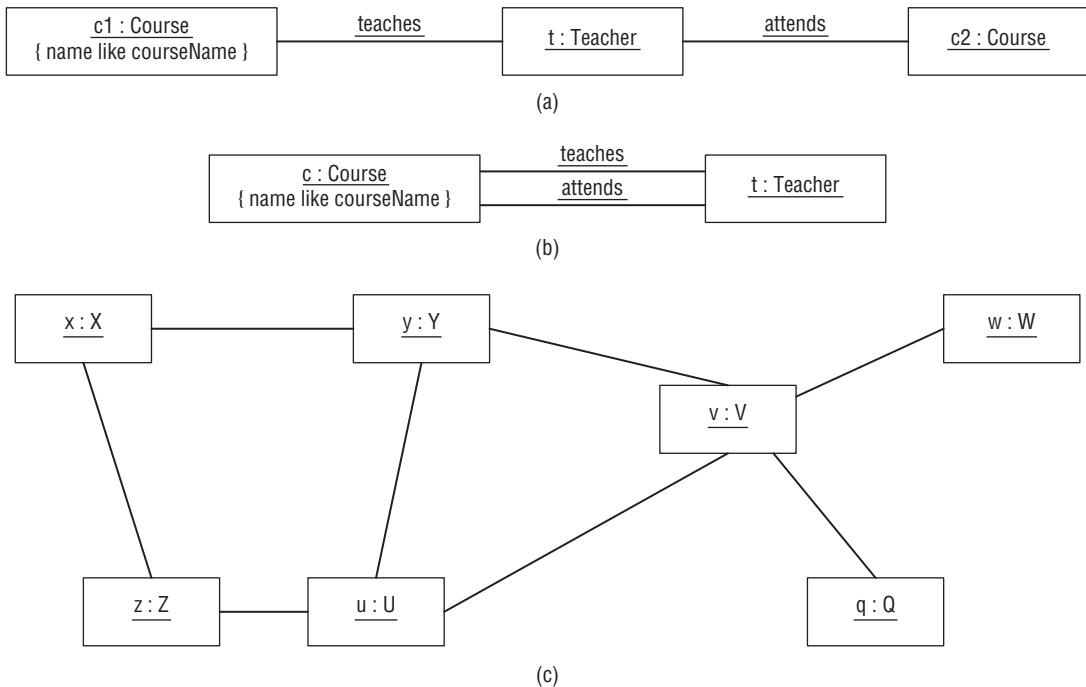
Figure 12-5 shows a few examples. The specification in Figure 12-5a designates a collection of tuples in which an object  $c_1$  of Course is linked to an object  $t$  of Teacher that is  $c_1$ 's teacher, while  $t$  is linked to an object  $c_2$  of Course that is  $t$ 's attendedCourse. Consequently, the resulting tuple type will be  $(c_1:\text{Course}, t:\text{Teacher}, c_2:\text{Course})$  (the order of components is determined by the ordering of the output parameters of the query, and is irrelevant here). The tuple collection will thus consist of all those (and only those) triples  $(c_1, t, c_2)$  that are linked accordingly. Note that, unless there is an explicitly defined constraint that  $c_1 \leftrightarrow c_2$ ,  $c_1$  may (but need not) be equal to  $c_2$ , meaning that the resulting tuple collection would include triples  $(c, t, c)$ , where  $t$  teaches and attends  $c$  (if such triple exists in the object space at all — that is, if it is allowed by the structural model and domain-specific constraints).

On the other hand, the specification in Figure 12-5b strictly requires that the objects referred to as  $c$  and  $t$  be linked with both links of attends and teaches. (The model constraint that a teacher cannot attend a course he or she teaches is not relevant for the moment. Although such a constraint would disable such cases in the actual object space, the query can still specify a search for such — non-existing — structures). In OQL, the equivalent query for this specification would be as follows:

```
SELECT ...
 FROM Course c, c.teachers t, t.attendedCourses c1
 WHERE c.name like #courseName# and c=c1
```

Note that in OQL, two object references,  $c$  and  $c_1$ , must be introduced and an additional constraint  $c=c_1$  must be explicitly stated. As opposed to the OQL query whose base tuple type has three components, the tuple type for the specification in Figure 12-5b has only two:  $(c:\text{Course}, t:\text{Teacher})$ .

As a result, the specification in Figure 12-5a covers a broader set of object structures than the specification in Figure 12-5b (that is, it designates a superset of object structures designated by the specification in Figure 12-5b).



**Figure 12-5: Examples of pattern object structure specifications.** (a, b) Two similar, but semantically different, pattern object structure specifications. The specification (a) designates a superset of object structures designated by the specification (b). (c) An example of a more complex pattern object structure specification having the form of an arbitrary graph.

Figure 12-5c indicates that a pattern object structure specification can be a rather complex graph of interconnected objects, arbitrarily linked according to the structural model. Although an equivalent OQL query can be easily derived from such a graph (the derivation is rather straightforward and can be formalized and automated), such an OQL query would be rather cluttered. This example shows that OQL and pattern object structure specifications basically have the same expressive power, although the latter has better expressiveness in some cases of complex object graphs. This is also a rule of thumb where (graphical) pattern object structure specifications could be preferred over OQL textual specifications. On the other hand, for simple queries with linearly chained objects and more complex selection expressions, OQL may be preferred.

In the current version of OOIS UML, pattern object structure specifications do not support specifications of data types instances at the same level as objects (that is, as nodes in the graphs). This is a disadvantage of using pattern object structure specifications over OQL queries because the latter support attributes of objects as terms in `FROM` clauses.

## Conditions

Additional selection conditions for a pattern object structure specification can be defined by constraints. The constraints can be attached to and written in the context of object specifications or the entire pattern object structure specification.

## Part III: Concepts

---

As usual, the context of a constraint represents the namespace for resolving names used in the constraint. Consequently, the properties of the object designated by an object specification can be referred to directly from a constraint attached to that object specification.

Figure 12-5a shows an example where the constraint name like `courseName` directly refers to the attribute name of that object. As the query operation to which this pattern object structure specification belongs (that is, is a method of) is the enclosing namespace of the specification, the constraint can also refer to the formal parameter `courseName` of the query operation.

The constraints attached to and written in the context of the entire pattern object structure specification can refer to all its contained object specifications by their names. This is the case with the constraint `c1<>c2` in Figure 12-4.

Both of these kinds of constraints can refer to the formal parameters of the query operation to which the specification belongs because it is the enclosing namespace of the specification.

Constraints in general can be specified in any constraint language supported by the implementation. It can be OCL, the expression language used in `WHERE` clauses of OQL, or another detail-level language.

The conjunction of all these constraints defines the selection condition for the base tuple set designated by the very pattern object structure specification. That is, the base tuple set designated by the pattern object structure specification is narrowed to all those (and only those) tuples that satisfy all the constraints attached to object specifications and the very pattern object structure specification.

## Parameterization and Results

When a pattern object structure specification defines the implementation of a query operation in the model, then it represents the method of that operation. Consequently, the pattern object structure is embedded within the namespace of that query operation. This is why the formal parameters of the operation can be accessed within the specification.

When a pattern object specification is used within a query as an object, the formal parameters are implicitly derived from the constraints defined within the specification. The names of the formal parameters must be then enclosed within # characters to be distinguished from other names.

The formal parameters of a query can be referred to only from the constraints and nested query invocations defined within the specification.

As already described, the resulting tuple type of a query operation is defined by the output parameters of the operation. The full definition of the method of a query operation must bind the output parameters to the values defined in the pattern object structure specification. That is, for each output parameter of the query operation, the modeler must define a binding of the parameter to one of the object specifications or some of the model elements accessible from an object specification. The rules for accessing elements from object specifications are the same as for terms in `SELECT` clauses in OQL queries. For the example in Figure 12-4, the bindings are documented in the note and are as follows:

- ❑ The output parameter `c1` is bound to the object designated by the object specification `c1`.
- ❑ The output parameter `name1` is bound to the value of `c1.name`.
- ❑ The output parameter `c2` is bound to the object designated by the object specification `c2`.
- ❑ The output parameter `name2` is bound to the value of `c2.name`.

It is the responsibility of the tool to provide a means to bind the output parameters. The bindings can be documented with notes in diagrams, as shown in Figure 12-4, where the note starts with the keyword Result::.

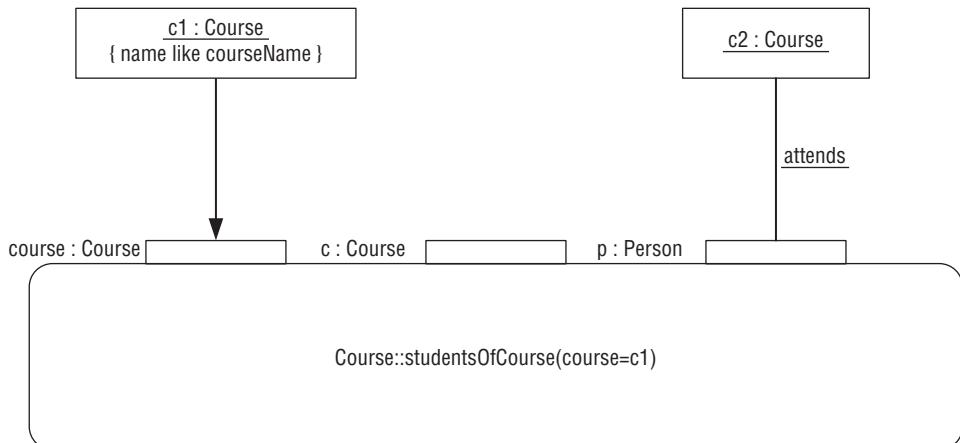
When a pattern object structure specification is the definition of a query as an object, it is left to the implementation to define the way of specifying or deriving the resulting tuple type of the query.

### Nesting

A pattern object structure specification may contain a specification of an invocation of a query operation. Such specification designates an invocation of the nested query when the enclosing query is evaluated.

An example is shown in Figure 12-6. Briefly stated, the meaning of the specification in Figure 12-6 is the same as of the following OQL query:

```
SELECT ...
 FROM Course c1, Course:::studentsOfCourse(c1), p.attendedCourses c2
 WHERE c1.name like #courseName#
```



**Figure 12-6: An example of a pattern object structure specification with the specification of invocation of the query operation `Course:::studentsOfCourse`. The formal parameter `course` of the invoked operation is bound to the object specification `c1`, while the output parameter of the operation `p` is used in the structure — linked with the object specification `c2`.**

In diagrams, an invocation specification is depicted as a rounded solid-outline rectangle, with its parameters shown as small rectangles placed on its border, as in creational specifications. Binding of formal parameters of the invoked operation to actual arguments is shown as arrows directed from an object specification to an input formal parameter. In Figure 12-6, the actual argument for the formal parameter `course` of the invoked operation `Course:::studentsOfCourse` is the object `c1`. The output parameters of the invoked query (that is, its results) may (but need not) be used in the enclosing specification as any other object specifications. In Figure 12-6, the output parameter, `p`, is linked to the object specification, `c2`, while the other output parameter, `c`, of the nested query is not used.

The semantics of invocation specification in pattern object structure specifications is the same as for nested queries in OQL.

## Part III: Concepts

---

Because of the limitations to specifications of objects of classes only, the output parameters of nested queries that are of data types cannot be used in pattern object structure specifications in the current version of OOIS UML. Additionally, the support for recursive query nesting is left to the implementation and is not required by the profile.

### Section Summary

- ❑ *Pattern object structure specifications* are UML object specifications that have the formal pattern matching semantics instead of illustrative semantics. Pattern object structure specifications can be used to specify queries.
- ❑ The object specifications represent prototypical objects, placeholders for concrete objects searched for in the actual object space.
- ❑ Link specifications represent the requirements that the objects that would match the pattern must be linked accordingly.
- ❑ Additional criteria can be specified as constraints in the context of every object specification, or in the context of the very pattern object structure specification.
- ❑ Pattern object structure specifications can contain specifications of invocations of other nested queries defined in the model, with optionally passing actual arguments to them.

## ***Creating Pattern Object Structures by Demonstration***

Like creational object structures, pattern object structure specifications can be designed by *demonstration* instead of using traditional modeling. For example, to specify a pattern object structure specification shown in Figure 12-7, the modeler can create the object structure by issuing commands at run-time, using the generic GUI of an OOIS UML framework. The steps for this scenario look like this:

1. The framework offers an option to use demonstration to specify a method for a query operation, or the specification for a query as an object. The modeler selects this option in order to specify the pattern object structure.
2. The framework opens a working environment that looks the same as the generic GUI.
3. The modeler creates an object of the class *Course* using a generic command for that. The new object appears in the environment. Using the specification dialog for that object, the modeler sets its name to *c1*.
4. The modeler selects the new object of *Course* and specifies the additional constraint name *LIKE #courseName#* for it, using a specialized dialog for that.
5. The modeler creates an object of the class *Teacher*, and another object of the class *Course*, using generic commands for that, and sets the name of the latter to *c2* through its specification dialog.

6. The modeler creates the links between the three objects (for example, by drag and drop).
7. The modeler selects an option of the environment to specify an additional constraint  $c1 <>> c2$  for the entire specification.
8. The modeler selects an option of the environment to specify the bindings of output parameters of the query to the existing objects or their properties. The modeler then specifies that the output parameters are:  $c1$  (bound to the object  $c1$ ),  $name1$  (bound to  $c1.name$ ),  $c2$  (bound to the object  $c2$ ), and  $name2$  (bound to  $c2.name$ ).
9. The modeler selects an option of the environment that indicates that the demonstration is completed and the pattern object structure is specified.

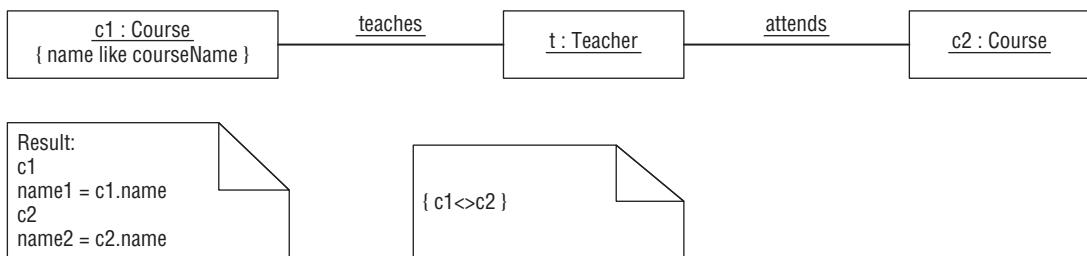


Figure 12-7: The UML object diagram for the pattern object structure specification of a simple query

It is important to note that the result of such a demonstration is always a pattern object structure specification that has exactly the same semantics as an equivalent specification made at design time. An implementation can opt to store the specification as the very object structure created by the demonstration, and to interpret this structure as a query when the query is evaluated at run-time. Alternatively, an implementation may opt to transform the created object structure into another internal form immediately after the demonstration is completed.

As with creational specifications, it is left to the implementation to what extent it will support the concepts of pattern object structure specifications through demonstration. Consequently, OOIS UML requires that a framework provide a means to demonstrate just the simplest pattern structures, without nested query invocations, at least those that contain the following specifications:

- ❑ **Object and link specifications** — The environment must allow the modeler to create objects of abstract classes for that purpose, or at least to specify that the created object of a concrete class is just a placeholder for any object of a generalized, possibly abstract class.
- ❑ **Constraints attached to objects and to the entire pattern object structure specification** — The implementation may limit the extent of supported expressions, operators, and model elements that can be referred to from the constraints.
- ❑ **Bindings of output parameters** — The implementation should provide a means to specify the bindings of output parameters of the query to the elements of the object specification.

### Section Summary

- ❑ Instead of traditional modeling, pattern object structure specifications can be designed by demonstration.
- ❑ Demonstration can be used to design simple pattern object structures, without nested query invocations, but including only the following:
  - ❑ Object and link specifications
  - ❑ Constraints attached to object specifications or the entire pattern object structure specification
  - ❑ Bindings of output parameters

# 13

## Operations and Methods

Operations are behavioral features of classifiers that specify services that can be requested from the classifiers' instances. Operations can have input, output, and return parameters, and can be invoked at run-time. This chapter discusses the modeling rules for operations and the semantics of their invocation. In addition, the chapter describes methods as behaviors that provide implementation of operations. Methods in OOIS UML can be specified in a detail-level language that is supported by the implementation of the profile. The support for error handling is provided through the exception mechanism. Finally, this chapter addresses the OOIS UML concurrency model, comprised of concurrent threads of control accessing the shared object space, as well as the issues of concurrency control.

### Operations

An *operation* is the specification of a service that can be requested from any instance of its owner classifier to affect behavior [Booch, 1999]. An operation is a behavioral feature of a classifier. It specifies that a service can be requested from any instance (direct or indirect) of that classifier. An operation specifies that service, but does not specify *how* this service will be fulfilled. It has a name, and may have formal parameters and return type. Additionally, in UML, operations may have *preconditions* and *postconditions*, which are optional sets of constraints specifying the state of the system when the operation is invoked and when it is completed, respectively. At run-time, an operation of an instance can be invoked. The actual arguments are then supplied. The invocation of the operation is manifested by the behavior specified as the implementation of the operation, which provides the requested service.

#### ***Operations as Behavioral Features***

An operation is a named element. Its enclosing namespace is the classifier that owns it. There can be several operations with the same name in the same classifier. However, these operations must be distinguishable, meaning that they must have different signatures that include their names and the types of their formal parameters. The details have been explained in Chapter 7, in the section

## Part III: Concepts

---

“Namespaces and Visibility.” An operation is not a packageable element — it must be owned by a classifier.

In addition, an operation is also a namespace. The parameters of an operation are named elements defined in the scope of that namespace. The optional method associated with the operation as its implementation is a namespace nested in the namespace of the operation. Actually, the same operation can have more than one method (but only one for each classifier) because the derived classifiers can redefine the method from the base classifier. Consequently, formal parameters of an operation are accessible within its methods.

An operation is a behavioral feature of a classifier. As with any other kind of feature, an operation can be *static* or *non-static*. If it is non-static, it represents a service that can be requested from every individual instance of the classifier. If it is static, it is a service of the classifier itself.

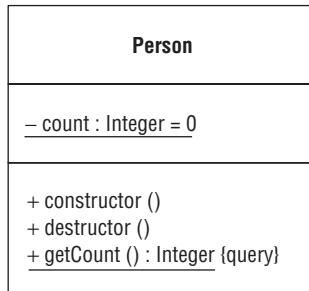
Figure 13-1 shows several examples of common usage of static operations. In all these examples, the service is the responsibility of, and is requested from, the classifier itself, not from any particular instance. (In some of these cases, the service may be the only available way to access an instance of the classifier.) Following are some common uses:

- ❑ Provide or modify information that is common for all instances of the classifier and not specific for a particular instance, such as a collection of all instances, their number, and so on. The static operation `Person::getCount` in Figure 13-1a retrieves the value of the private static attribute `count`.
- ❑ Get a reference to a singleton (one and only existing) instance of the classifier, as suggested by the Singleton design pattern [Gamma, 1995]. The singleton can be a direct or indirect instance of that classifier. The static operation `CommunicationAPI::Instance` in Figure 13-1b returns a reference to the singleton instance of that class.
- ❑ Get a reference to a specific prototype instance of a classifier, or a prototyping structure that can be cloned then, as suggested by the Prototype design pattern [Gamma, 1995].
- ❑ Create and return a new (direct or indirect) instance of the classifier, or an entire structure of related instances, according to the provided parameters that drive the configuration of the instance or the structure.

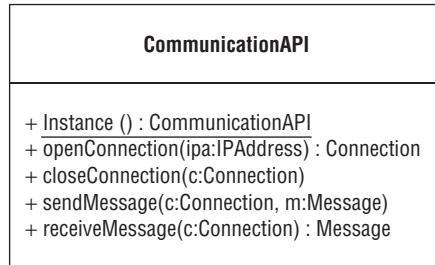
Static operations have a similar purpose as with global procedures and functions in traditional, procedural programming languages because they are not requested from particular objects. However, unlike global procedures, static operations of classifiers have the following characteristics:

- ❑ They are logically “packed” in their classifier, to emphasize their logical relationship with the concept modeled with the classifier.
- ❑ They have names declared in the namespace of the classifier, so that referring to those names must be in accord with the general naming rules.
- ❑ They may be encapsulated in classifiers (that is, may be private or protected), so that other parts of the model can access them in a controlled way.

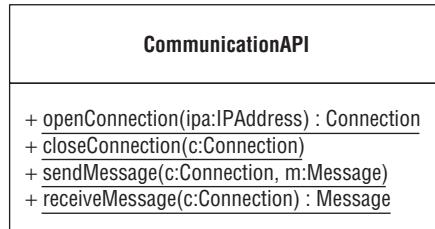
A classifier may contain just static operations and no other structural or behavioral features. Such classifiers are sometimes called *class utilities* (see Figure 13-1c). Although they can be directly instantiated (unless they are abstract), their direct instances do not embody any structure, and do not differ except



(a)



(b)



(c)

**Figure 13-1: Examples of static operations of classifiers.** (a) The static operation `Person:: getCount` retrieves the information common for all instances of the class `Person` — precisely, the value of the private static attribute `count`. (b) The static operation `CommunicationAPI:: Instance` returns a reference to a singleton instance of the class, according to the Singleton design pattern. (c) The class utility `CommunicationAPI` provides a single service access point to an API of a layer, according to the Façade design pattern.

## Part III: Concepts

---

by their pure identity. Such classifiers are actually not intended for instantiation. Instead, a class utility represents a logical grouping of some related services not requested from instances. A class utility is often used to pack the functions and procedures of an API of a subsystem, or to represent a single service access point to a subsystem or a layer, as suggested by the Façade design pattern [Gamma, 1995]. A static operation cannot be abstract in OOIS UML. Static operations (as with other static features) are underlined in the textual UML notation, as shown in Figure 13-1.

An operation can be declared as a *query*, which means that its method must not modify the object space. Such operations are pure functions. Their methods cannot contain any action that modifies the object space, nor can they invoke any non-query operation. Query operations are tagged with the word `query` written in curly braces next to the operation signature in the textual UML notation, as shown in Figure 13-1a. Operations of data types are always query operations.

An operation can be declared as a *leaf*, meaning that its method cannot be redefined in derived classes. Leaf operations are tagged with the word `leaf` written in curly braces next to the operation signature in the textual UML notation.

### Section Summary

- ❑ An *operation* is a named element and a namespace. Its enclosing namespace is the classifier that owns it. The parameters and the (optional) methods of an operation are named elements defined in that namespace.
- ❑ A *static operation* is a service that can be requested from the classifier, not only from every particular instance of the classifier.
- ❑ Static operations are underlined in the textual UML notation.
- ❑ Static operations are used to provide or modify information shared among all instances of the classifier, or a service of the very classifier, not of each particular instance.
- ❑ An operation can be specified as a *query*, in which case its method must not modify the object space (that is, cannot have side effects), but is a pure function.
- ❑ An operation can be declared as a *leaf*, meaning that its method cannot be redefined in derived classes.

## Parameters of Operations

As a behavioral feature, an operation can have *formal parameters* that define how values can be passed between the caller and the invocation of a behavioral feature. The parameters of a behavioral feature are ordered.

Parameters are named elements of the model, defined within the behavioral feature as a namespace. The parameters of the same operation must have unique names within that namespace to be distinguishable. Parameters do not have to have names. However, unnamed parameters cannot be referred to by names within the methods written in textual detail-level languages.

Every parameter has a *direction*, which can be one of the following:

- ❑ **in** — Indicates that values are passed through this parameter from the caller into the behavioral feature.
- ❑ **inout** — Indicates that values are passed through this parameter from the caller into the behavioral feature, and then back from the behavioral feature out to the caller.
- ❑ **out** — Indicates that values are passed through this parameter from the behavioral feature out to the caller.
- ❑ **return** — Indicates that values are passed through this parameter as return values from the behavioral feature out to the caller.

If the direction is unspecified, it is **in** by default. No more than one parameter of a behavioral feature can have the **return** direction.

The direction is specified in front of the parameter in the textual UML notation. The type, multiplicity, uniqueness, and ordering of the sole return parameter (if any) may also be specified as the type, multiplicity, uniqueness, and ordering of the very operation. The return parameter can be omitted from the parameter list in that case. For example, the following are signatures of several sample operations of different classifiers:

```
Course:::calcRevenue (return resultingRevenue : Real) : Real
Person:::getPriceCoef () : Real
Real:::sqrt (x : Real) : Real
Real:::sqr (in x : Real, return result : Real)
Date:::stringToDate(in txt : Text, out date : Date, return status : Boolean)
Text:::toDate(out year:Integer, out month:Integer, out day:Integer):Boolean
```

Even the entire list of parameters can be suppressed in the notation. Again, the absence of parameters in the diagram does not imply their absence in the model.

Parameters are typed multiplicity elements. Their multiplicity, ordering, and uniqueness are specified in the same way as for variables. The default multiplicity (if not explicitly specified) for parameters is 0..1. Following is an example:

```
Integer:::sort(inout array : Integer[*]{nonunique,ordered})
```

The mappings of different cases of multiplicities and ordering into types of the host detail-level language (so that parameters can be used within methods written in that language) is the same as for variables and will be described later in this chapter.

The type of a parameter can be a class or a data type. Additionally, a parameter can be of any type that is built in the host detail-level language or available in its library. In that case, the parameter can be used within the operation's method as usual in that language. However, a model containing such a parameter is dependent on the implementation language and may not be portable to others. For example, if C++ is used as the implementation language, operations can be declared as follows:

```
Text:::toString() : char*
Text:::fromString(s : char*) : Text
```

## Part III: Concepts

---

An input parameter (of `in` or `inout` direction) may have its *default value*. If a parameter has a default value, then an invocation of that operation can be specified without providing an actual argument for that parameter. In that case, the default value will be provided as the missing argument for that parameter. In the textual UML notation, default values for parameters are specified after the `=` sign that follows the parameter specification, as in the following example:

```
Real::log(base : Real = 10) : Real
```

In this example, the operation `Real::log` can be invoked with one actual argument explicitly provided (when the base of the logarithm function is taken to be that value), or without any actual argument (when the parameter for the base of the logarithm function takes the value 10). Following are examples:

```
Real r = ...;
Real p = r.log(2);
Real q = r.log(); // same as r.log(10)
```

Consider another example:

```
Date::incrementDay(numOfDays : Integer = 1) : Date
```

The operation `Date::incrementDay` can be invoked with one actual argument of type `Integer`, when the date instance whose operation is invoked will be incremented with the given number of days and the obtained resulting date will be returned, or without any argument, when the increment is one day:

```
Date d = ...;
Date d1 = d.incrementDay(30);
Date d2 = d.incrementDay(); // same as d.incrementDay(1)
```

The specifications of default values of parameters can be expressions in the selected detail-level language, written in the same way as initializers of variables. The scope (that is, the owned namespace) of those expressions is the owner operation. The expressions are evaluated at run-time, at the time of each invocation of the operation when the actual argument is missing. Consider the following example:

```
Date::daysSince(since : Date = Date::today()) : Integer
```

Whenever the operation `Date::daysSince` is invoked without the actual argument, the operation `Date::today` is evaluated and returns the current day at the time of that invocation.

For multi-valued parameters, the default values are specified in the same way as variables, as shown here:

```
X::f(array : Real[0..3]{unique} = {0.0, Real::sqrt(2.0)}, : Integer = null)
```

For easier implementation in some programming languages that have that kind of restriction, an implementation of the profile may restrict the use of default values of parameters for the last parameters in the ordered collection of the operation's parameters only. That is, a parameter can have its default value only if all other parameters that follow it in the ordered collection of parameters of the operation also have default values. Following is an example:

```
X::f(x : X, y : Y, r : Real = 0, i : Integer = 1) // correct
X::g(x : X, y : Y, r : Real = 0, i : Integer) // incorrect
```

### Section Summary

- ❑ Formal parameters define how values can be passed between the caller and the invocation of a behavioral feature.
- ❑ Parameters are named elements of the model, defined within their behavioral feature as a namespace. Parameters are typed multiplicity elements.
- ❑ Every parameter has its *direction*, which can be the following:
  - ❑ *in* — Indicates that values are passed through this parameter from the caller into the behavioral feature.
  - ❑ *inout* — Indicates that values are passed through this parameter from the caller into the behavioral feature, and then back from the behavioral feature out to the caller.
  - ❑ *out* — Indicates that values are passed through this parameter from the behavioral feature out to the caller.
  - ❑ *return* — Indicates that values are passed through this parameter as return values from the behavioral feature back to the caller.
- ❑ An input parameter (of *in* or *inout* direction) may have its *default value*. If a parameter has a default value, then an invocation of that operation can be specified without providing an actual argument for that parameter. In that case, the default value will be provided as the missing argument for that parameter.
- ❑ The specifications of default values of parameters can be expressions written in the detail-level language. The expressions are evaluated at run-time, at the time of each invocation of the operation when the actual argument is missing.

## Operation Invocation

An operation is invoked using a Call Operation action. This action specifies the following:

- ❑ The operation that is invoked.
- ❑ The target instance whose operation is invoked. This instance is optional if the operation is static.
- ❑ The ordered collection of actual arguments of the invocation.

As previously described, the semantics of the Call Operation action is as follows. When a Call Operation action is executed, it activates the execution of the method corresponding to that operation for the target instance. The values of input arguments (those of *in* or *inout* direction) of the action, as well as the reference to the target instance, are available to the execution of the invoked method. When the execution of the method completes, the values of output arguments (those of *out*, *inout*, and *return* direction) are returned to the caller, and are put on the output pins of the action. Upon receipt of the reply transmission, the execution of the Call Operation action is complete.

## Part III: Concepts

---

These elements of the specification of the Call Operation action and its semantics will be described in the discussions that follow, except for the resolution of the method that is invoked on operation call, which will be explained later in this chapter, in the section “Methods.”

### ***Specification of Operation***

If the Call Operation action is specified as a model element, such as in creational or pattern object specifications, it directly refers to the called operation as another model element. That reference is established directly between those two model elements as independent entities with their inherent identities, and does not rely on the name of the operation as its identifier. The way this relationship is established is the responsibility of the modeling tool. For example, the tool may allow the modeler to directly point to the operation to be invoked, and to select that operation and distinguish it from the others in many different ways. Actually, the problem of identification is the same as with objects in the object space, as described previously in this book.

However, in a textual detail-level language, such an approach cannot be used. Instead, the name of the operation is used as its identifier, and it is the responsibility of the language compiler to map the given name to the operation as a model element. In that case, the syntax of the detail-level language is used to specify the Call Operation action. For example, in Java, C#, and in the OOIS UML native detail-level language, this may look like the following:

```
aCourse.calculateRevenue()
```

In C++, the same call may look like this (assuming that `aCourse` is a pointer to an object):

```
aCourse->calculateRevenue()
```

The rules for mapping the operation name specified in the action to the operation as a model element are defined in the selected detail-level language and may differ from one language to another. This mapping may encompass certain resolution rules for overloaded operations (that is, operations having the same name) and their signature matching (that is, matching of actual arguments to formal parameters according to their types). In general, most common languages have resolution rules discussed in the paragraphs that follow.

The starting namespace in which the name of the invoked operation is resolved is the classifier that is the type of the target instance reference. This type is resolved statically, at compile-time, and may be a generalization of the direct type of the target instance that is referenced at run-time.

For example, suppose that a general class `Base` has an operation `f` with one input formal parameter of type `Integer`, and that a class `Derived` that specializes `Base` has another operation `f` with no formal parameters:

```
Base::f(in : Integer = 0)
Derived::f()
```

The invocation of `f` in the following code is mapped to `B::f`, and not to `D::f`:

```
Base b = new Derived;
b.f(); // Base::f is called
```

Similarly, the invocation of `f` in the following code is irregular, as it refers to `D::f`, because `D::f` hides `B::f` and does not have an input parameter:

```
Derived d = ...;
d.f(3); // Derived::f is referred here
```

To invoke `B::f` for `d`, the qualified name must be used:

```
d.Base::f(3); // Base::f is called
```

Consequently, the modeler should be careful when introducing an operation in a specializing classifier having the same name as an operation in the generalizing classifier, because the former would hide the latter, and the latter would not be accessible over references to the specializing classifier and unqualified operation name.

A similar problem arises when several operations have the same name even in the same namespace. As already explained, there might be several operations with the same name in the same classifier, as long as they are distinguishable (have different signatures). Such overloaded operations introduce the problem of signature matching for operation calls.

For example, let the classifier `X` have several overloaded `f` operations with different signatures:

```
X::f(: Integer)
X::f(: Text)
X::f(: Integer, : Text)
X::f(: Base)
X::f(: Derived)
```

The following calls can be unambiguously resolved because there is exactly one operation whose signature exactly matches the types of the actual arguments for each of these calls:

```
X anX = ...;
Base b = new Derived;
Derived d = ...;
anX.f(0); // X::f(Integer) is referred here
anX.f("Hello"); // X::f(Text) is referred here
anX.f(0, "Hello"); // X::f(Integer,Text) is referred here
anX.f(b); // X::f(Base) is referred here
anX.f(d); // X::f(Derived) is referred here
```

However, if there is no operation whose signature exactly matches the types of the actual arguments, but there are several that may accept those arguments because of the substitution rule or language-specific type conversions, an ambiguity may occur, and the operation call may be illegal. For example, let the classifier `Y` have two `f` operations:

```
Y::f(: Base, : Derived)
Y::f(: Derived, : Base)
```

## Part III: Concepts

---

Let `b`, `d`, and `anY` be defined as follows:

```
Base b = ...;
Derived d = ...;
Y anY = ...;
```

Then the following invocations are unambiguous:

```
anY.f(b,d); // Y::f(Base,Derived) is referred here
anY.f(d,b); // Y::f(Derived,Base) is referred here
```

However, the following invocation is ambiguous and, thus, illegal, because there is no exact matching and both operations can accept the argument of type `Derived` for the parameter of type `Base` because of the substitution rule:

```
anY.f(d,d); // Ambiguity
```

In any case, an operation is a *valid candidate* for call resolution if its signature matches the call (that is, if the provided actual arguments conform to the corresponding formal input parameters of the operation). In general, the compiler for the host detail-level language applies a precise algorithm for resolving the operation call specification in case of overloaded operations, defined strictly for that language. For each operation call specification, that algorithm may have one of three possible outcomes:

- There is only one valid candidate at all, or there is only one valid candidate that is, by any criterion used by the algorithm, preferred over other valid candidates.
- There are no valid candidates for the operation call (that is, there is no operation in the targeted namespace or the inherited namespaces that may accept the types of the provided actual arguments). In this case, the operation call is illegal.
- There are several overloaded operations that are valid candidates for the operation call, and the algorithm cannot prefer any of them, as in the previous example. In this case, the operation call is ambiguous and illegal.

The only positive outcome of the algorithm is the first outcome in the preceding list, in which case the invocation is correct and the sole or the preferred valid candidate will be referred to by the Call Operation action. The other two outcomes would result in a compilation error.

In some languages, the compiler may prefer one valid candidate over the others because of some resolution rules that are not so obvious to the human reader of the code. For example, the resolution algorithm may favor shorter transitive paths in specialization hierarchies when applying the substitution rule, or use any other language-specific resolution rule. In such cases, an operation call can be regular and unambiguous for the compiler, although it may not be completely clear to the human reader which of the candidates is preferred. Of course, such cases should be strictly avoided because they lead to less readable code.

For all these reasons, OOIS UML does not strictly define either the operation call resolution rules or the rule that determines whether two overloaded operations are distinguishable. These rules are left to the implementation of the profile, and are expected to be in accord with the rules used in the host detail-level language. In any case, several general rules are expected to always hold:

- Only the parameters of kind `in` and `inout` may be considered for signature distinguishing and matching (that is, constitute the signature of an operation). Such parameters are referred to as input parameters.

- ❑ Two overloaded operations (having the same name) are certainly distinguishable if they have different numbers of input parameters, or if the types of the input parameters at the same place in the ordered sub-collection of input parameters have unrelated types (types that are not related with generalization/specialization, directly or indirectly).
- ❑ The valid candidate for an operation call that provides exact matching of types (without substitution rule applied) is always preferred over those valid candidates that do not provide exact matching.
- ❑ For the purpose of operation call resolution, an operation having the default value of an argument may be considered as two operations, one without the argument, and one with that argument with no default value. For example, consider the following operation

```
X::f(: Integer = 0)
```

For the purpose of operation call resolution, this can be treated as two operations:

```
X::f(: Integer)
X::f()
```

Therefore, there cannot be another operation `f` in the same classifier that has no parameters because it would be indistinguishable.

In short, the modeler should be careful when overloading operations. To avoid ambiguities, improve the readability of the model and code, and achieve portability, the following modeling guidelines should be followed:

- ❑ Overloaded operations should have either different numbers of input parameters, or their input parameters should have unrelated and significantly different types.
- ❑ There should be no operations that differ only in that one has a parameter with a default value, and the other does not have that parameter.

Even though the listed cases of overloaded operations may be allowed in a certain implementation, they should be strictly avoided. If they cannot be avoided (which occurs rarely in practice), the problem should be simply resolved by renaming the operations — instead of using the same name for two or more operations that cannot be clearly distinguished by a human reader, different names should be used to remove confusion. In that way, the operation calls will be clearly unambiguous in any detail-level language and the model is, therefore, less dependent on that language.

An operation of a classifier instance can be identified dynamically and called through the reflection mechanism. Namely, every instance of a classifier can respond to the operation `callOperation(: Text, : Tuple[0..1])`, which accepts a string parameter that specifies the name of the operation and an optional tuple of actual arguments of the invocation (that is, a set of name-value pairs). The result of the invocation is a tuple consisting of output arguments, as shown in the following example:

```
Course c = ...;
Tuple t = c.callOperation("calculateRevenue", null)
Real r = (Real)t.field("result");
```

An output parameter can be accessed by referring to the field of the returned tuple having the name of the parameter. The invoked operation that can accept the given actual arguments must exist and

## Part III: Concepts

---

must be accessible from the place of invocation of callOperation. Otherwise, the invocation raises an exception.

### Section Summary

- ❑ If the Call Operation action is specified as a model element, it directly refers to the invoked operation as another model element.
- ❑ In a textual detail-level language, the name of the called operation is used as its identifier, and it is the responsibility of the language compiler to map the given name of the invoked operation to the operation as a model element.
- ❑ The rules for mapping the operation name specified in the action to the operation as a model element are defined in the host detail-level language, and may differ from one language to another. This mapping may encompass certain resolution rules for overloaded operations and their signature matching.
- ❑ The modeler should be careful when overloading operations. To avoid ambiguities, improve the readability of the model and code, and achieve portability, the following guidelines should be followed:
  - ❑ Overloaded operations should either have different numbers of input parameters, or their input parameters should have unrelated and significantly different types.
  - ❑ There should be no operations that differ only in that one has a parameter with a default value and the other does not have that parameter.
  - ❑ Overloading of operations in specializing classifiers should be done carefully because such operations hide those from the generalizing classifiers.
- ❑ An operation of a classifier instance can be identified dynamically and called through the reflection mechanism.

### **Target Instance and Arguments Binding**

Quite similar to specifying the operation, the target instance and the bindings of actual arguments to formal parameters of the operation call can be specified explicitly and by direct relationships, if the Call Operation action is specified as a model element. The only difference is that the target instance and the arguments are provided as dynamic values, at run-time, instead of statically, at design time. In particular, this means the following:

- ❑ The target instance of the operation call is provided as the result of another action. The output pin of that preceding action is directly bound to the input pin of the Call Operation action that uses that result as the target instance of the call. If the operation is static, the target instance does not have to be specified.
- ❑ If the operation has input parameters (those of direction kind `in` or `inout`), a set of input pins of the Call Operation action receive the values that represent the input arguments for the call.

These input pins are bound to output pins of other actions that provide those values as their results. The input parameters for which actual arguments are not provided take their default values computed at the time of action activation.

- ❑ If the operation has output parameters (those of direction kind `out`, `inout`, or `return`), a set of output pins of the Call Operation action receive the values that represent the output arguments of the completed call. These output pins can be bound to input pins of other actions that use those values as their inputs.

In a textual detail-level language, the target instance can be given with an expression in that language. The type of that expression determines the classifier of the operation. The result of that expression, computed at run-time, refers to the target instance whose operation is invoked. The way the expression is specified and evaluated is defined in the concrete detail-level language. For example, in the OOIS UML native detail-level language, an operation call may look like this:

```
Person p = ...;
Real r = p.attendedCourses->val().calculateRevenue();
```

In this example, the expression `p.attendedCourses->val()` returns one object of `Course` in the collection of attended courses of the object referred to by `p`, while the dot operator `(.)` designates access to the operation to be called. This may look similarly in other languages.

If the expression that determines the target instance (that is, the left-hand operand of the dot operator) has a result of zero cardinality (that is, it is `null`), the effect is undefined in the OOIS UML native detail-level language, and may depend on the implementation. For example, an implementation may raise an exception in that case, may produce an unpredictable reaction, or may simply ignore the operation call, so that the call is not performed at all. Other detail-level languages may define their behavior in such cases, or may leave it undefined, too.

If the operation is static, the target instance does not have to be specified. Instead, the operation is invoked by simply stating its name (qualified if necessary), as in the following example:

```
Real root2 = Real::sqrt(2);
```

However, the target instance can be specified even for an invocation of a static operation. In that case, that target instance simply serves to determine the classifier whose operation is invoked, and thus the operation name does not need to be qualified, as in the following example:

```
Real r = ...;
Real root2 = r.sqrt(2);
```

In some programming languages, if the target instance of a static operation call is specified by an expression, that expression is not even evaluated at run-time, but is just examined at compile time to determine the type of its result in order to identify the operation. The behavior in the OOIS UML native detail-level language is undefined in this respect.

In the OOIS UML native detail-level language and most other textual languages that can be used as the detail-level language, actual arguments of an operation call are provided in a comma-separated list of expressions between parentheses. For example, consider the static operation `Person::create` that is a creator with the following signature:

```
Person::create(name:Text, age:Integer, address:Text="", email:Text="")
```

## Part III: Concepts

---

Its invocation may look like this:

```
Person::create("John Smith", 25, "Broadway 1, NY", "john@acme.com")
```

The actual arguments are bound to formal input parameters of the operation in order. More precisely, the actual arguments, ordered as they are listed in the operation call (from left to right), are in the same order bound to the input parameters of the operation (the parameters of direction kind `in` or `inout`). If there are remaining input parameters that have no arguments bound to them, they take the default values evaluated at the invocation time. If those parameters do not have default values, the call is illegal. The order of evaluation of expressions for actual arguments is undefined.

Note that this is not the only possible way of binding arguments to parameters. A detail-level textual language can bind arguments by explicitly naming the parameters, without relying on their order, as shown in the following example:

```
Person::create(age=25, email="john@acme.com", name="John Smith")
```

In this case, the missing argument for the parameter `address` takes its default value.

If the operation has output parameters (those of direction kind `out`, `inout`, or `return`), then their values become available at the place of operation invocation once the call action has completed. In that case, the result of a call of that operation is always a tuple having a component for every output parameter. For example, consider the following operation:

```
Course::bestStudent(return student:Person, out name:Text, out grade:Text)
```

In this example, the access to its output parameters in the OOIS UML native detail-level language can look like this:

```
Course c = ...;
TupleType(student:Person, name:Text, grade:Text) result = c.bestStudent();
Person student = result.student;
Text studentName = result.name;
...
```

Or, more concisely, it could look like this:

```
Course c = ...;
Person student = c.bestStudent().student;
```

A similar example of accessing the values of the result directly looks like this:

```
Integer sz = c.bestStudent().student.attendedCourses->size();
```

As you can see, the components of a tuple of output parameters resulting from an operation call are accessed using the dot notation.

In another detail-level language, a similar approach could be used to access the output parameters over an API, as shown in the following example:

```
Course c = ...;
Tuple result = c.bestStudent();
Person student = (Person)result.field("student");
```

```
Text studentName = (Text)result.field("name");
...
```

The components of a tuple in this case are accessed by invoking the operation `field` of a tuple, with the name of the component provided as a string argument. It is obvious that the binding of a component name to the actual value of the component is performed at run-time. The operation `field` returns a typeless reference to a classifier instance that must be downcast to the actual type of that instance.

In the OOIS UML native detail-level language, the sole output parameter of kind `return` can be accessed as any other output parameter (as described earlier), but it is also implicitly assumed as the result of the operation call if no access to a component of the resulting tuple is specified. In other words, the very operation call refers to the return parameter of the operation, as is usual in other classical programming languages. Consider the following example:

```
Course c = ...;
Person student = c.bestStudent(); // The same as: ... = c.bestStudent().student
```

or

```
Integer sz = c.bestStudent().attendedCourses->size();
```

If the operation has no other output parameters except the return parameter, the same approach can be used in other classical programming languages. However, if the operation has other output parameters, its result is always a tuple, and thus the parameter of direction kind `return` can be accessed only within the returned tuple as shown previously, because the native operation call in that language really returns a tuple.

### Section Summary

- ❑ The specification of the target instance and the bindings of input and output parameters of the operation can be explicit, if the Call Operation action is specified as a model element.
- ❑ In case a textual detail-level language is used, the target instance can be given with an expression written in that language.
- ❑ If the operation is static, the target instance does not have to be specified.
- ❑ In the OOIS UML native detail-level language and most other textual languages that can be used as the detail-level language, actual arguments of an operation call are provided in a comma-separated list of expressions between parentheses. The actual arguments are (in the order they are listed) bound to the input parameters of the operation (the parameters of direction kind `in` or `inout`). If there are remaining input parameters that have no arguments bound to them, they take their default values evaluated at the invocation time.

*Continued*

- ❑ If the operation has output parameters (those of direction kind `out`, `inout`, or `return`), then their values become available at the place of operation call once the call has completed. In that case, the result of a call is always a tuple having a component for every output parameter of that operation.
- ❑ In the OOIS UML native detail-level language, the sole output parameter of direction kind `return` can be accessed just like any other output parameter, but it is also implicitly assumed as the result of the operation call if no access to a component of the resulting tuple is specified. If the operation has no other output parameters than the return parameter, the same approach can be used in other classical programming languages.

### Synchronous and Asynchronous Call

So far, the discussion has assumed that operation calls are *synchronous*, meaning that the Call Operation action completes its execution when the method activated by the call is completed (see Figure 13-2a). Once it has issued the call, the caller waits until the invoked method completes. These are also the sole semantics of operation calls in most traditional programming languages, and are, thus, the default semantics of operation calls in OOIS UML.

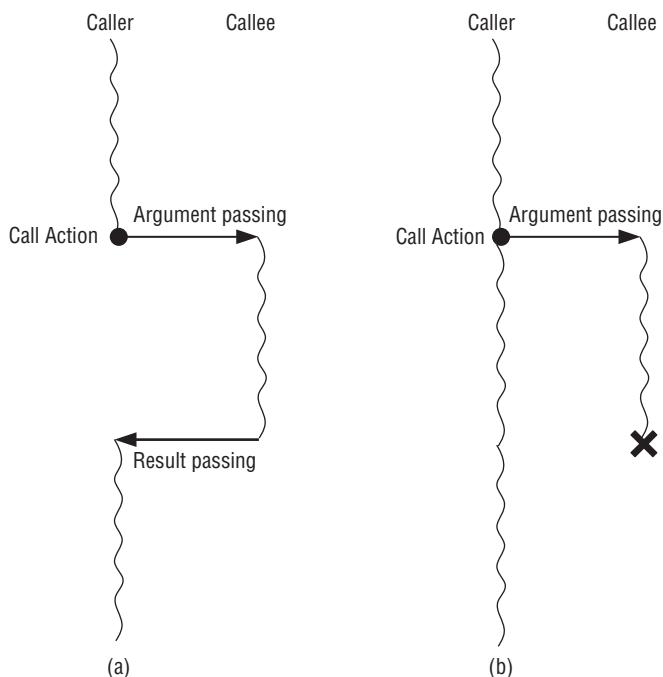


Figure 13-2: Synchronous vs. asynchronous operation call. (a) In a synchronous call, the caller waits until the invoked method completes. (b) In an asynchronous call, the caller proceeds concurrently with the method activated by the call, and does not wait for the output parameters to be returned. Time runs vertically downwards in these diagrams.

However, an invocation can also be *asynchronous*, meaning that the Call Operation action completes its execution as soon as the operation is called. The caller proceeds as soon as the operation call is dispatched, and does not wait the invoked method to complete, nor does it expect the return values (output parameters) from the call (see Figure 13-2b). The caller actually proceeds concurrently with the method activated by the call. The execution of the activated method represents a new *thread of control* that runs concurrently with the caller's thread of control, as shown in Figure 13-2b. When the activated method is completed, the values of the output parameters are simply discarded and not returned to the caller because the action that performed the call has been already completed.

In the OOA UML native detail-level language, an asynchronous call is specified using the operator |. For example, the following operation call action activates the method for the operation execute for the given command aCmd in a new thread that runs concurrently with the caller's thread of control:

```
aCmd.|execute()
```

On the other hand, the following call action blocks the caller execution until the invoked method completes and the output parameters are passed back:

```
aCmd.execute()
```

For static operations, an asynchronous invocation looks like this:

```
|Person::create("John Smith", 25, "Broadway 1, NY", "john@acme.com")
```

In a traditional programming language used as the detail-level language, such a notation is not available, because these languages mostly do not support the notion of asynchronous calls. Instead, the implementation may use the following approach.

For each operation *op* of a classifier, the model compiler may generate a native operation *async\_op* that wraps *op* so that it receives the same set of input parameters, and has no output parameters. It starts a new thread of control (using a built-in or librarian concept of thread in that language), and passes the received input parameters to that thread. The thread then invokes *op* with those parameters. All these subtleties of the implementation are hidden from the modeler, except that the modeler must call *async\_op* instead of *op* if an asynchronous call is needed. Following are a couple of examples:

```
aCmd.async_execute()
Person::async_create("John Smith", 25, "Broadway 1, NY", "john@acme.com")
```

Asynchronous calls are a powerful feature in UML because they provide a means to start a concurrent execution in a simple way — by simply invoking an operation. Additionally, it is worth noting that the initiative for the concurrent execution of a method is with the caller, and not with the classifier that encapsulates the method, as it is in some other languages (for example, threads in Java). A processing that should execute concurrently with the caller's thread of control may be started by a simple asynchronous operation call, and not with creating a thread as an “active object” that encapsulates some method. This is particularly useful for the “background” processing that is started by a user's command issued from the GUI, so that the control can be returned to the GUI thread, which may then process further user commands.

For example, an exhaustive algorithm that traverses a large part of the object space graph and performs some calculation may take some time to complete. During that computation, it would be annoying for the user if the caller's thread (that is, the thread of the GUI which accepted the user's command to start

## Part III: Concepts

---

that computation) blocked until the computation has completed, which would mean that the GUI does not accept any other user command during that time. That would cause the GUI to “freeze” for a while. Instead, the command can perform an asynchronous call to the operation that performs that computation in the background and continue to process user’s commands. On the other hand, the background method may update a status bar that shows its progress to the user. The same holds true for some other computations that are suitable for background processing, such as exporting part of the object space to a file in some format, preparing a report for printing, and so on.

Although a powerful concept, asynchronous calls should be used carefully and sparingly. Their careless and excessive usage may cause a proliferation of concurrent threads during the execution, and thus increase the conflicts on shared objects and system overhead because of context switches. This may significantly degrade the performance of the system. Additionally, lots of concurrent threads increase the complexity of the system, decrease its readability, and make its understanding, testing, verification, and maintenance more difficult. It may also lead to more bugs because of illegal concurrency conditions, such as race conditions, deadlocks, and other design flaws.

Therefore, the architecture of the system should prescribe the cases in which asynchronous calls could be used, and provide strong reasons for that. Some compelling reasons have just been described. In those cases, the asynchronous calls would be localized within the methods for the `execute` operation of specific commands (classes that specialize the class `Command`) that are activated on users’ actions, and implement the background processing like the ones described.

### Section Summary

- ❑ In a *synchronous* call, the caller waits until the invoked method completes.
- ❑ In an *asynchronous* call, the caller proceeds as soon as the operation call is dispatched, and does not wait the invoked method to complete, nor does it expect the return values (output parameters) from the call.
- ❑ In a classical programming language used as the detail-level language, an asynchronous call must be done over a native (synchronous) call of a wrapper operation that starts a concurrent thread of control.
- ❑ Although a powerful concept, asynchronous calls should be used carefully and sparingly. In general, the architecture of the system should prescribe the cases in which asynchronous calls could be used, and provide strong reasons for that. One compelling reason for the use of asynchronous calls is for the start of “background” processing when the user issues a command from the GUI, so that the control can be returned to the GUI thread that may process further user’s commands.

## Preconditions and Postconditions

Consider a simple operation (actually, a pure function) that calculates the square root of a real number. The operation could be specified as a static operation of the class `Real`:

`Real::sqrt(in arg:Real, return result:Real)`

It is clear that the square root, as a unique function of a real number, is defined for non-negative real arguments only. Therefore, the operation `sqrt` should accept only non-negative real numbers. But how to ensure that? How to clearly specify that to the callers? What happens if the caller violates this requirement of the operation?

One solution would be to introduce a subtype of `Real` that represents non-negative numbers only. Another solution is to use a *precondition*, which is a constraint that is attached to an operation and defined in the context of the operation. It defines the condition that must be satisfied at the time of the call in order to obtain the proper response of the called operation, so that the method can rely on it. The precondition may refer to the parameters of the operation or other accessible parts of the object space, to check whether that space and the actual arguments satisfy the necessary condition for the operation call. For the given example of the `sqrt` operation, the precondition can be expressed as follows:

```
arg>=0
```

Quite similarly, you may want to ensure that the implementation of the operation (that is, its method) satisfies a certain condition, so that the caller can rely on the provided result. In other words, you may want to define a *postcondition* that must be satisfied upon the operation's completion, to ensure that the operation has performed correctly.

For the considered example of the `sqrt` operation, the postcondition may ensure that the result is acceptably close to the exact value, or, more precisely, that the difference of the argument and the squared result is less than an acceptable error margin. Let's assume that the class `Real` has a static operation `abs` that returns the absolute (non-negative) value of its argument, and a static attribute `epsilon` that defines the absolute error margin. The postcondition of the operation `sqrt` can be expressed as follows:

```
abs(arg-result*result)<=epsilon
```

Preconditions and postconditions can be specified the same as all other constraints in UML. They are attached to the operation and are defined in the context of the operation. The effect of a failed precondition or postcondition is the same as for any other failed constraint (that is, an exception is raised). When a textual notation is used for preconditions and postconditions in OCL, the keywords `pre` and `post` are used as prefixes to OCL expressions, as shown in the following example:

```
context Real::sqrt(arg : Real) : Real
 pre: arg>=0
 post: abs(arg-result*result)<=epsilon
```

Sometimes a postcondition must refer to the value that a certain property had before the operation was called. For that purpose, the postfix operator `@pre` is used in OCL. For example, assume that the operation `reducePrice` in the class `Product` reduces the price of the product by the given percentage:

```
Product::reducePrice (in percent : Real)
```

The precondition of this operation should ensure that the parameter `percent` has a value between 0 and 1. On the other hand, the postcondition could ensure that the price of the product is equal to the value of that price before the reduction, multiplied by the value of `1-percent`:

```
context Product::reducePrice (percent : Real)
 pre: percent>=0 and percent<=1
 post: price = price@pre * (1-percent)
```

## Part III: Concepts

---

The identifier `price` refers to the slot of the instance of `Product`, which is the target instance of the operation call. The identifier `price@pre` refers to the value of the property `price` of that instance at the start of the method execution.

This operator can be used for any feature of a classifier (that is, a property or an operation). If the feature is followed by other operators, the suffix `@pre` is postfixed to the feature name, before the other operators, as shown in the following example:

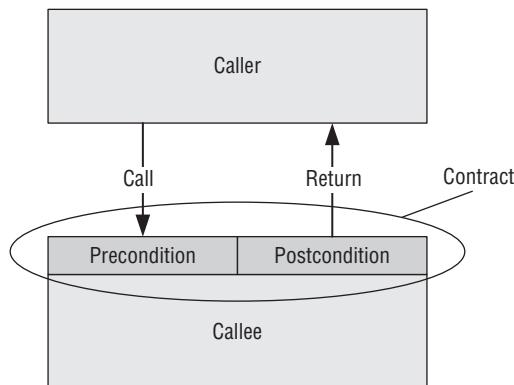
```
context Company::increaseProfit()
post: getAnnualProfit() > getAnnualProfit@pre()
```

When the pre-value of a feature evaluates to an object, all further features that are accessed by that object are the new values (upon completion of the operation). Consider these examples:

- `a.b@pre.c` takes the old value of property `b` of `a`, and then the new value of `c` of that old value.
- `a.b@pre.c@pre` takes the old value of property `b` of `a`, and then the old value of `c` of that old value.

The `@pre` postfix is allowed only in OCL expressions that are part of a postcondition. Referring to the current value of a feature of an object that has been destroyed during execution of the method results in an undefined value. Also, referring to the previous value of an object that has been created during execution of the method results in an undefined value.

The purpose of a precondition is to ensure that the caller guarantees that a certain constraint is met if it wants the called operation to behave correctly (see Figure 13-3). Otherwise, if the precondition is not satisfied, it cannot expect the operation to behave as expected, or the operation simply must not be called.



**Figure 13-3: The programming by contract metaphor.** Two parties, the caller and the callee, mutually agree on their obligations. The caller must ensure that the precondition is met before the call is performed, if it wants the callee to behave properly. If that obligation is satisfied, the callee guarantees that the postcondition will be satisfied on return.

On the other hand, a postcondition ensures that the implementation of the operation is proper and does not have any hidden deficiencies (such as bugs). It ensures that if the precondition is satisfied, the caller can be sure that the effect of the operation, its results, or the entire object space is in a consistent and correct state. If the postcondition is not satisfied, it indicates that the operation did not behave well because of some unexpected deviations (such as bugs in its implementation).

In that sense, precondition and postcondition represent a kind of a *contract* by which two parties, the caller and the callee, mutually agree on their responsibilities (see Figure 13-3). If the responsibilities are satisfied by both parties, their collaboration is guaranteed to behave correctly and to keep the system in a consistent state. Therefore, the proper definition of preconditions and postconditions improve the reliability and robustness of software. This is the main idea of *programming by contract*, a metaphor that encourages the proper use of preconditions and postconditions to improve robustness of collaborations between software components.

As with many other concepts and metaphors in programming that improve the robustness of software but do not affect its core structure or behavior, preconditions and postconditions could be completely ignored and not used at all in a system's model. The system will not lose any of its functionality in that case. However, the system or its parts may lose in its robustness. Therefore, preconditions and postconditions are useful modeling concepts that may improve a system's quality. If they are used, however, they must be specified consistently, carefully, and completely.

Preconditions and especially postconditions have another possible use. During the system analysis and specification, the semantics and effect of an operation can be specified by its postcondition. For example, the postcondition of the operation `Real::sqrt` specifies that the operation should produce the square root of the given number with the acceptable accuracy. The postconditions of other examples given in this discussion may have similar purpose. System analysts can use postconditions to specify the semantics of operations in the conceptual model in a concise and declarative way, without dealing with implementation aspects. Developers can then implement the methods and use the postconditions to verify the implementations. Finally, if they are not needed, the postconditions can be removed from the model of a productive system to reduce the run-time overhead.

### Section Summary

- ❑ A *precondition* is a constraint that is attached to an operation and defined in the context of the operation, which defines the condition that must be satisfied at the time of the call in order to obtain the proper response of the called operation.
- ❑ A *postcondition* is a constraint that is attached to an operation and defined in the context of the operation, which must be satisfied upon the operation's completion to ensure that the operation has performed correctly. It must be ensured by every implementation of the operation.
- ❑ The postfix operator `@pre` is used in OCL to refer to a value of a feature before the operation was called.

# Methods

A *method* is the behavioral element of the model that provides the implementation of an operation. When an operation of a classifier instance is invoked, one execution of the method attached to that operation is activated. Methods in OOS UML can be specified using a detail-level language. One such language is the OOS UML native detail-level language.

## **Methods as Implementations of Operations**

A *behavior* in UML is an element of the model that specifies how instances of a classifier behave, react to outer stimuli, or change their states during their lifetimes. A behavior is an element of the model that can be instantiated at run-time. The instantiation of a behavior is its activation — that is, its execution.

When a behavior is associated as the method of a behavioral feature, it defines the implementation of that feature — that is, the computation that generates the effects of the behavioral feature.

As described in Chapter 5, a specializing classifier can associate a different method to an inherited operation. When the inherited operation is invoked for an instance of the specializing classifier, the method from the specializing classifier is invoked, and not the one from the generalizing classifier. This effect is called *polymorphism*.

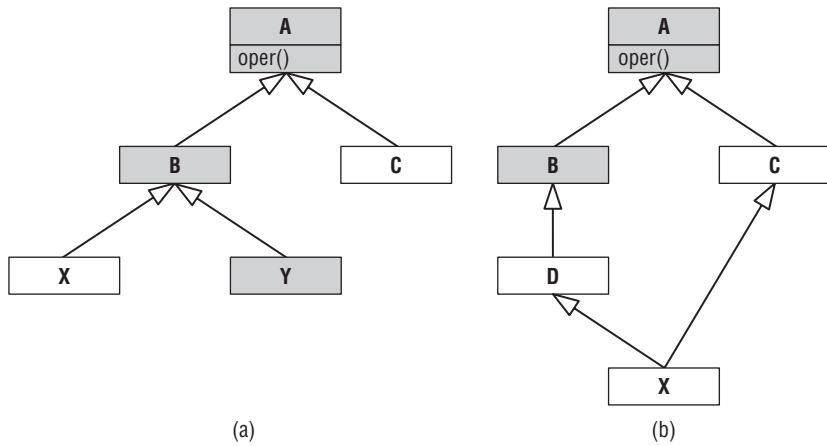
If the direct classifier of the instance defines the method for the invoked operation, that method is activated. However, if that classifier does not provide a method for that operation, one of the methods inherited from the generalizing classifiers is activated.

Although the method resolution policy may appear to be straightforward, it is not always so. Consider the examples shown in Figure 13-4. The topmost class A in the hierarchy defines an operation `oper`, and provides a method for it. The specializing classes that also provide (redefine) the methods for this operation are filled in gray in the picture. The question is which method is activated for the leaf classes, X, if `oper` is called for its direct instances? In the case of single inheritance, as shown in Figure 13-4a, there is always only one path starting from the considered classifier upstream the generalization relationships. The method that the considered classifier in the hierarchy inherits is the first method encountered on that path. For the example in Figure 13-4a, the method that the class X inherits is the one defined in the class B, while the class Y has its own method, and the class C inherits the method from the class A.

In the case of multiple inheritance (as shown in Figure 13-4b), there might be several paths starting from the considered classifier. This is why there might be several candidate methods that the considered classifier may inherit. For the example in Figure 13-4b, the class X may inherit the method defined in the class B (because it is the first encountered method over the path X-D-B-A), or the method from the class A (because it is the first method encountered over the path X-C-A). The resolution policy in that case is not defined in OOS UML and may depend on the implementation. One approach could favor closer methods (if any) — that is, those that are encountered in a smaller number of hops over generalization edges — while another implementation can select the first encountered method in the depth-first or breath-first traversal of the hierarchy.<sup>1</sup> It is expected that the implementation follows the rules of the host implementation language, if that language allows multiple inheritance at all.

---

<sup>1</sup>Note that such an approach requires the inherent ordering of generalization relationships that start from a classifier. Such ordering does not exist in the standard UML metamodel, and should be introduced by the implementation.



**Figure 13-4: The problem of method resolution in case of (a) single and (b) multiple inheritance.**

For the reasons described, in order to avoid the non-determinism of method resolution and preserve the portability over different implementations, a model should not have such cases. If they occur, they should be resolved by explicitly specifying the desired method in the problematic classifier (x in the example shown in Figure 13-4b).

In UML, methods can be specified in many different ways. Ultimately, the specifications are expressed in terms of actions upon the object space and other control structures that dictate the way and ordering of their execution. One way of specifying methods available in OOIS UML is using a detail-level language. One predefined detail-level language is the OOIS UML native detail-level language described next.

### Section Summary

- ❑ A *behavior* in UML is an element of the model that specifies how instances of a classifier behave, react to external stimuli, or change their states during their lifetimes.
- ❑ When a behavior is associated as the method of a behavioral feature, it defines the implementation of that feature (that is, the computation that generates the effects of the behavioral feature).
- ❑ In UML, methods can be specified in many different ways. One option is to use a detail-level language.

## The OOIS UML Native Detail-Level Language<sup>2</sup>

The OOIS UML native detail-level language is a Java-like textual language for specifying methods in OOIS UML and other parts of OOIS UML models at the level of detail. It inherits all Java lexical rules,

<sup>2</sup>This is advanced reference material that can be skipped on first reading.

## Part III: Concepts

---

syntax, and semantics for the parts of the Java language used within Java methods, and extends them by the constructs specific for OOIS UML as described in this book.

In particular, the method body may consist of statements and comments. Comments are similar to those in Java and can be the following:

- ❑ Started by `//` and terminated by the first subsequent end of line
- ❑ Enclosed within the character pair `/*` and the first subsequent inverse character pair `*/`

Following are examples:

```
// This is a comment
/* This is a comment
that spans over
few lines.
/* It is still being continued...
... up to here */
```

Other OOIS UML statements include the following:

- ❑ Compound statements (blocks) enclosed in curly braces `{}`
- ❑ Declarations of variables
- ❑ Expressions terminated with semicolon `(;)`
- ❑ Control flow statements (`if-else`, `switch`, `for`, `while`, `do-while`, `break`, `continue`, and `return`)
- ❑ try-catch blocks for fault tolerance (described later in this chapter)

Some of these constructs will be briefly described in the discussions that follow. Important elements of methods are *variables* that represent local references to objects and data values. For the details of the syntax and semantics of the parts inherited from Java, see books and manuals on Java.

### Section Summary

- ❑ The OOIS UML native detail-level language is a Java-like textual language for specifying methods in OOIS UML.
- ❑ The method body may consist of statements and comments. Comments are the same as in Java.

## Variables

Within methods, the results of actions can be stored in local *variables*, which serve as local temporary storage for passing data between actions indirectly. The values of variables are local to the methods they belong to, and cannot be accessed from other contexts, although they can refer to objects of the common object space. Formal parameters also behave as variables within methods.

According to the basic semantics of modern OO programming languages, OOIS UML variables always store *references* to instances of classifiers (classes or data types), thus representing intermediaries to those instances, allowing substitution, as in the following examples:

```
Integer i = new Integer;
Person p = new Teacher;
```

In this example, the variable named `i` refers to a newly created instance of the data type `Integer`, whereas the variable `p` is declared to be of type `Person`, but it actually refers to a newly created object of class `Teacher`, because the substitution takes place here.

Variables are typed multiplicity elements, which means that they have a type, multiplicity, ordering, and uniqueness specifications, as already described in Chapter 7.

In the OOIS UML native detail-level language, a variable is introduced by its declaration. Each variable must be declared before it is used. A declaration of a variable introduces the variable into the current namespace and optionally initializes its value. A declaration of the variable has the following syntax:

```
variable-declaration ::= type multiplicity-specopt var-decl-list ;
var-decl-list ::= var-declaration
var-decl-list ::= var-declaration , var-decl-list

var-declaration ::= identifier
var-declaration ::= identifier = initializer

multiplicity-spec ::= [multiplicity] multiplicity-propertiesopt
multiplicity-properties ::= { ordering , uniqueness }
multiplicity-properties ::= { uniqueness , ordering }
multiplicity-properties ::= { ordering }
multiplicity-properties ::= { uniqueness }

ordering ::= ordered
ordering ::= unordered
uniqueness ::= unique
uniqueness ::= nonunique

initializer ::= expression
initializer ::= collection
collection ::= { expression-list }
expression-list ::= expression
expression-list ::= expression , expression-list
```

If the multiplicity is not specified, the default multiplicity for variables is `0..1`. If ordering and uniqueness are not specified, the default settings are `ordered` and `nonunique`.<sup>3</sup>

For single-valued variables with allowed cardinality `0`, a special symbol `null` is used to denote the absence of the value in the variable (that is, the actual cardinality zero):

```
p = null; // p's cardinality is now 0; p does not refer to any instance
...
if (p != null) ... // Does p refer to any instance?
```

---

<sup>3</sup>This rule is not part of standard UML — it is an extension of the OOIS UML profile.

## Part III: Concepts

---

If the lower bound of the multiplicity is greater than 0, the initializer is mandatory if strong conformance is applied; otherwise it is optional. If the initializer is omitted, the variable has the initial cardinality zero (it stores no value). The initializer must specify at least as many initial values as the lower multiplicity bound specifies, if strong conformance is applied. Each expression within the initializer must result in a reference to an instance of the appropriate type. For a single-valued variable with multiplicity 0..1, the initializer can be `null`. For example, these are valid declarations of variables:

```
Real[0..3]{unique} array = {0.0, Real::sqrt(2.0)};
Person[1] p = new Person;
Teacher t = null;
```

If a multi-valued variable is ordered, the expressions in the initializer list are evaluated in the order they are listed, and the elements of the variable are initialized in the same order. Otherwise, the order of evaluating expressions in the initializer list and initializing elements of the variable is undefined.

Note that single-valued variables with the default multiplicity 0..1 behave like ordinary Java references. Similarly, multi-valued variables with multiplicity \* and default ordering and non-uniqueness behave like dynamic arrays.

The type of a variable can be the following:

- A classifier (class or data type) from the UML model
- A built-in scalar primitive type as in Java — that is, one of the following:
  - `boolean` — Either `true` or `false`
  - `char` — 16-bit Unicode character
  - `byte` — 8-bit signed two's complement integer
  - `short` — 16-bit signed two's complement integer
  - `int` — 32-bit signed two's complement integer
  - `long` — 64-bit signed two's complement integer
  - `float` — 32-bit IEEE 754-1985 floating-point number
  - `double` — 64-bit IEEE 754-1985 floating-point number

For example, these are legal declarations:

```
Course[*]{nonunique,ordered} selectedCourses;
int[1..3]{unique} oneToThreeInts = {1,2};
boolean b1 = true, b2 = false, b3 = b2;
float[0..1] f1 = null, f2 = 0.0;
```

Instances of these built-in primitive types are pure scalar values. They have no features (structural or behavioral). They can be used as pure values in Java-like expressions and as arguments of operations, or values of variables. Comparison between two instances of the same or compatible primitive types is value-based. Values of `short` and `byte` are always promoted to `int` before being evaluated — they are only stored, and never operated upon.

These primitive types cannot be specialized or, by any other means, used or extended in the model, except as described.

The variables of these types are declared as with all other OOIS UML variables, except that the variables without explicitly specified multiplicity have default multiplicity 1..1. These variables can be initialized as other OOIS UML variables, using the same rules regarding the initialization and the syntax for literals. Additionally, the default initial values of variables (if not explicitly initialized) are `false` for `boolean`, `\u0000` (the character with Unicode 0) for `char`, and zero for all other numeric types. Following are some examples:

```
bool b; // Equivalent to: bool[1] b = false;
float[0..1] f; // f is null
double[1] d; // d is 0.0
char c; // Equivalent to: char[1] c = '\u0000'
```

The set of actions on variables, including iterations, is basically the same as with properties of classifiers, and will not be repeated here.

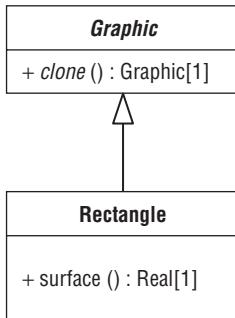
## Section Summary

- A *variable* is a local temporary storage for passing data between actions indirectly. Variables serve to store results of actions within methods.
- A variable is a typed multiplicity element.
- At one moment in run-time, an instantiation of a variable stores zero or more references to instances of data types or classes. The cardinality and uniqueness of the collection of references must conform to the specification of the variable as a multiplicity element. The type of each referenced instance must conform to the type of the variable (substitution is assumed).
- A variable must be declared and initialized by the necessary number of initialization expressions, so the initialization satisfies the variable's multiplicity and uniqueness.
- The set of actions on variables, including iterations, is basically the same as with attributes.

## Casting

The OOIS UML native detail-level language (as with most other conventional OO programming languages) is *statically typed*, meaning that type-conformance rules are checked at compile time. Any violation of type-conformance rules is reported as a compilation error. However, sometimes a value (for example, a variable or a return value of an operation) actually, at run-time, always refers to an instance of a specializing type, although its statically declared type is a generalizing type. In other words, it is possible that the semantics of the program ensure that the referenced instance is always of a specializing type, and it is necessary to access it as a specialized instance, although the type-conformance rules do not allow that.

For example, let's assume that the generalizing class `Graphic` shown in Figure 13-5 provides an abstract operation `clone` that clones a graphic object and returns its exact copy.



**Figure 13-5: Classes  
Graphic and  
Rectangle and the  
polymorphic operation  
clone**

A specializing class `Rectangle` redefines the method of this operation and, of course, returns an instance of `Rectangle` as a clone. Therefore, the semantics of the program ensure that the result of an invocation of `clone` for an instance of `Rectangle` is always an instance of `Rectangle`, although the result type of `clone` is `Graphic`. However, it may be necessary that a caller of `clone` access the returned object as an instance of `Rectangle`:

```
Rectangle r1 = ...;
Rectangle r2 = r1.clone(); // Error: type-conformance violation
Real s = r2.surface(); // We wanted to do this
```

This code will cause a compilation error because of a type-conformance rule violation, because the return type of the operation `clone(:Graphic)` doesn't conform to the type `Rectangle` of the variable `r2`.

For this purpose, the OOIS UML native detail-level language provides the *cast* operator, which requires a type conversion of the value that is the operand of the operator into the specified target type. Its syntax is as follows:

```
(type) expression
```

Here, the *expression* provides the operand of one type, and the *type* specifies the target type. The result of this operator is of the specified target type, but only if the result of the operand expression is really of that type. The type of the expression is checked at run-time, and an exception is raised if it is not of the target type.

For the given example, the cast operator can be used to resolve the described conflict:

```
Rectangle r1 = ...;
Rectangle r2 = (Rectangle)r1.clone(); // Correct: type conversion
Real s = r2.surface();
```

The cast operator is mostly used to convert a value of a generalizing type to a specializing type, as in the previous example. Such conversion is usually referred to as *downcasting*, as opposed to *upcasting*, which is a synonym for the substitution rule.

## Section Summary

- ❑ *The cast operator results in the same value as the result of the operand expression, but with the target type. If the operand expression is not of the target type (checked at run-time), an exception is raised.*

## Implementations in Other Detail-Level Languages

Classical OO programming languages generally do not support all variants of multiplicity and other characteristics of variables with the appropriate conformance rules. Because OOS UML allows methods to be written in any available detail-level language supported by an implementation of the profile, this section briefly analyzes how variables can be implemented in other detail-level languages, and how they are coupled with other elements of the model, especially with other operations invoked from the methods written in those languages.

A classical OO programming language can simply support variables with only two kinds of multiplicity:

- ❑ **Single-valued variables with multiplicity 0..1** — These are ordinary references or pointers of the adequate type that may refer to an instance (that is, have the cardinality one), or be `null` (that is, have the cardinality zero). For example, in C++, pointers have these characteristics:

```
Car* aCar = parkingQueue->first(); // * denotes a pointer in C++
if (aCar!=0) ... // 0 is a symbolic value of a null pointer
```

- ❑ **Multi-valued variables with multiplicity 0..\*** — These can be implemented as collections, which are also instances of a class in those languages. The collection class can implement the relevant actions for multi-valued variables through its operations. For example, in C++, a template class `Collection`, parameterized with the type of the variable, may be used like this:

```
Collection<Car*> parkingQueue;
parkingQueue.add(aCar1);
parkingQueue.add(aCar2);
parkingQueue.add(aCar3);
```

If collections cannot be statically typed, the type of the element can be given as the argument of the construction of a collection instance, and checked on each action that may violate type conformance. In the following code, we assume that the class `Collection` is designed specifically for the purpose of supporting the semantics of OOS UML collections:

```
Collection parkingQueue = new Collection(Class.forName("Vehicles.Car"));
parkingQueue.add(aCar1);
parkingQueue.add(aCar2);
parkingQueue.add(aCar3);
```

In the previous example, the argument of the constructor of `Collection` uses the Java reflection service `Class.forName` to find the object that represents the class `Vehicles::Car` by its fully

## Part III: Concepts

---

qualified name. It will be used then to check the type conformance of references that will be added to the collection. If the type conformance rule is violated, an exception will be raised at run-time.

Similar to the dynamic type conformance checking, multiplicity can be also specified and checked at run-time, instead of compile time. If that approach is taken, collections can have arbitrary multiplicities specified at run-time, precisely, at the moment of their creation. For example, the following statement will raise an exception of type `UpperBoundViolationException` at the last action `add(aCar4)`:

```
Collection aSmallParking =
 new Collection(Class.forName("Vehicles.Car"), 0, 3); // Multiplicity 0..3
aSmallParking.add(aCar1);
aSmallParking.add(aCar2);
aSmallParking.add(aCar3);
aSmallParking.add(aCar4);
```

The same approach can be used in C# as in Java. To dynamically fetch an object's type through reflection, `Type.GetType` would be used in C# instead of `Class.forName` in Java.

It is up to the implementation which approach will be used.

The rest of this section discusses some issues related to conformance rules and other aspects of such implementations.

### Type Conformance Rules

Single-valued variables, implemented as simple references or pointers, inherently support static checking of type conformance rules, if the language has static rules related to this aspect and supporting the substitution rule. For example, in C++:

```
Car* aCar = ...; // * denotes a pointer in C++
Vehicle* aVehicle = aCar; // Substitution rule
```

Such languages also inherently support transitive substitution. For example, in Java or C#, if `FamilyCar` is a subtype of `Car`, which is a subtype of `Vehicle`, it is correct to write the following:

```
FamilyCar aFamilyCar = ...;
Vehicle aVehicle = aFamilyCar;
```

As for multi-valued variables (that is, collections), if static type checking is supported, the situation is also analogous to the OOIS UML conformance rules. For example, in C++:

```
Collection<Vehicle*> parking;
parking.add(aCar);
parking.add(aFamilyCar);
```

Note, however, that this approach does not allow assignments of multi-valued variables. For example, it is not possible to do something like this in C++, even if the operator = is overloaded for the template class Collection:

```
Collection<Vehicle*> vehicleParking;
Collection<Car*> carParking;
vehicleParking = carParking; // Incorrect!
```

If dynamic type checking is applied, the type conformance rule can be supported both at the level of a single value being added to a collection, as well as the assignment of the entire collection, but the type checking is not done statically. Therefore, instead of compile-time errors, run-time errors are possible. For example, in Java:

```
Collection vehicleParking =
 new Collection(Class.forName("Vehicles.Vehicle"));
Collection carParking =
 new Collection(Class.forName("Vehicles.Car"));
vehicleParking.set(carParking); // Correct
```

The analogous implementation in C# would simply use `Type.GetType` instead of `Class.forName`.

## Multiplicity Conformance Rules

Because single-valued variables include both zero and one cardinalities, they always conform to each other regarding multiplicity. On the other hand, assignment of a single-valued variable to a multi-valued variable can be supported by a separate operation of the class `Collection`. For example, in Java, this may look like this:

```
Car aCar = ...;
Collection aParking =
 new Collection(Class.forName("Vehicles.Vehicle"));
aParking.set(aCar);
```

On the other hand, multiplicity conformance for multi-valued variables, implemented with the class `Collection`, can be checked at run-time. For example, this code in Java will raise an exception:

```
Collection aSmallParking =
 new Collection(Class.forName("Vehicles.Vehicle"), 0, 3);
Collection aHugeParking =
 new Collection(Class.forName("Vehicles.Vehicle"));
aSmallParking.set(aHugeParking); // Exception!
```

## Ordering Conformance Rules

For single-valued variables, ordering has no meaning. Multi-valued variables, on the other hand, can be simply implemented and always treated as ordered. Indeed, if a multiplicity element is declared as unordered in UML, this does not mean that ordering does not exist, but that it is under the control of the implementation. If the implementation always preserves the ordering and the variables in a detailed-level language are always taken as ordered, all ordering conformance rules are satisfied.

## Part III: Concepts

---

### Uniqueness Conformance Rules

For single-valued variables, uniqueness has no meaning. Multi-valued variables can be treated as non-unique by default. In other words, the class `Collection` that implements multi-valued variables does not prevent duplicate values and is, thus, non-unique. On the other hand, a specializing class `UniqueCollection` can implement a unique collection. Note that `UniqueCollection` inherits all operations from `Collection`, but redefines the methods for adding new values to the collection because that is the only difference in behavior between unique and non-unique variables (a unique collection will not add a value that already exists in it). In Java notation, this may look like this:

```
class Collection {
 ...
 public void add(Object) throws UpperBoundViolationException ...
 ...
}

class UniqueCollection extends Collection {
 ...
 public void add(Object) throws UpperBoundViolationException ...
 ...
}
```

Because of the specialization relationship between these classes and the substitution rule, the uniqueness conformance rule is implicitly supported. Below is an example in Java:

```
Collection aHugeParking = new UniqueCollection(Class.forName("Vehicles.Vehicle"));
```

### Interoperability

All the presented information holds true for variables and actions performed on them within the code of the same method written in the selected conventional detail-level language. However, the code written in such language may invoke an operation defined in the UML model, whose parameters are typed multiplicity elements in general. So, how does an operation invoked from the detail-level language interoperate with the caller code in terms of arguments conformance rules?

Considering the passing of arguments to the invoked operation as an assignment of one variable (the actual argument) to another variable (the formal parameter), the same rules already described can be applied to operation invocation. In brief, this means the following:

- ❑ Single-valued parameters of operations, including return parameters, are treated in the detail-level language as references or pointers having the multiplicity 0..1. Multi-valued parameters (including return parameters) are treated as references or pointers to `Collection` if they are non-unique, or to `UniqueCollection` if they are unique. All static rules applicable to those types are checked at compile time. For example, if the operation `Message::sendTo(receiver:Person[1])` is called from a C# code, the following code will not compile because the actual argument is not of type `Person`:

```
Vehicle v = ...;
Message msg = ...;
msg.sendTo(v); // Compilation error!
```

- ❑ The type conformance rules for single-valued variables can be checked at compile time. For multi-valued variables, on the other hand, type conformance rules can be checked at

run-time, possibly causing exceptions if the type of the elements of the collection as the actual argument does not conform to the type of the formal parameter. For example, if the operation `Message::sendTo(receivers:Person[*])` is called from Java code, the following code will raise an exception if the actual argument does not contain only objects of type `Person`:

```
Collection receivers = ...;
Message msg = ...;
msg.sendTo(receivers); // Possible exception!
```

- ❑ The multiplicity conformance of the actual argument to the formal parameter is always checked at run-time, because it cannot be statically checked. Therefore, the invoked operation can raise an exception because of the failure of its precondition that checks the multiplicity conformance of arguments. For example, if the operation `Message::sendTo(receiver:Person[1])` accepts a parameter of multiplicity 1, the following code will raise an exception when the actual argument is `null`:

```
Message msg = ...;
Person receiver = null;
msg.sendTo(receiver); // Exception!
```

- ❑ The uniqueness conformance rule is statically checked. Namely, if a multi-valued formal parameter of an operation is unique, it is considered as declared to be of type `UniqueCollection` in the detail-level language. Therefore, the actual argument must also be of the same type.
- ❑ The ordering rule is not considered either at compile time or at run-time.
- ❑ All these rules also hold true for the return parameter of an operation. If the return parameter is single-valued, it is treated as a pointer or reference in the detail-level language. If it has the multiplicity 1, it is guaranteed that it cannot be `null`. If the return parameter is multi-valued, it is declared to be of type `Collection` or `UniqueCollection`, depending on its uniqueness. It is guaranteed that the collection will contain only values of the appropriate types, and will have the cardinality allowed by its declared multiplicity. For example, if the operation `Book::getLibrarianClassifications` is defined in the UML model to return a parameter of type `LibrarianClassification[1..5]{ordered, unique}`, its result is treated as being of type `UniqueCollection` in the detail-level language, and it is guaranteed that the returned collection has at least one value and at most five values of type `LibrarianClassification`:

```
UniqueCollection libClasses = aBook.getLibrarianClassifications();
LibrarianClassification primaryClassification =
 (LibrarianClassification)libClasses.first(); // Type-safe,
// certainly not null
```

## Section Summary

- ❑ A classical OO programming language, used as a detail-level language for methods of operations, may simply support variables with only two kinds of multiplicity:
- ❑ Single-valued variables with multiplicity 0..1, implemented as simple references or pointers to objects of classes that may have `null` values

*Continued*

- ❑ Multi-valued variables with multiplicity `0..*`, implemented as collections (that is, as instances of class `Collection`)
- ❑ Type conformance rules can be checked statically for single-valued variables and dynamically for multi-valued variables.
- ❑ Multiplicity conformance rules are implicitly satisfied for single-valued variables, but must be checked dynamically for multi-valued variables.
- ❑ Collections can always be ordered, so that order conformance rules are always implicitly satisfied.
- ❑ A subclass `UniqueCollection` implements unique multi-valued variables and redefines the method for operation `add`, so that uniqueness conformance rules are also statically checked.

### Compound Statements

A *compound statement* (or a *block*) is a sequence of statements enclosed in curly braces `{ }`. Syntactically, it is a statement and can appear whenever a statement is expected. In particular, some other statements (such as, for example, `if` or `while`) expect one statement in their bodies, and it is often necessary to have more than one in such a body. In such cases, a block can be used, as shown in the following example:

```
if (oldCourse!=null && newCourse!=null) {
 oldCourse.students->remove(aPerson);
 newCourse.students->add(aPerson);
}
```

A compound statement is a namespace owned by its enclosing compound statement. All named elements (variables) declared within a block are accessible only from the nested namespaces (inner blocks) and not from outer namespaces. This implies the usual hiding rules as in other traditional block-structured programming languages. Following is an example:

```
{ // Outer namespace
 int x = 1; // The first x
 x = 2;
 { // Inner namespace
 int x = 0; // The second x, hides the first x from the outer namespace
 x = 2; // Refers to the second x from the inner namespace
 } // Closure of the inner namespace
 x = 3; // Refers to the first x from the outer namespace
}
```

The entire body of a method is implicitly a compound statement.

The effect of execution of the statements within a compound statement is always as if they are executed sequentially, one by one, in the same order as they are listed in the compound statement. This, however, does not mean they must really be executed in that particular order. An implementation may schedule their execution in any order, or even compute them concurrently, provided that their ultimate effect is as

if they had been executed sequentially in order. For example, because the following three statements do not have mutual dependencies, they can be executed in any order, or even concurrently:

```
{
 a=b+c;
 d=e-f;
 g=h*i;
}
```

## Declarations

A *declaration* introduces one or more local variables into the scope. The declaration declares the variables' names, type, multiplicity, uniqueness, and ordering. The scope of the variables declared in a declaration is the namespace (the compound statement) in which the declaration is given. That means that the variable can be accessed from that namespace (block) and all nested namespaces (blocks), unless it is hidden by other variables from those nested namespaces having the same names.

Syntactically, a declaration is a statement and can appear whenever a statement is expected, not necessarily at the beginning of a block.

Every method of a non-static operation has an implicitly declared local variable named `this`. For a method of a classifier `X`, `this` is of type `X` and of multiplicity 1. During the execution of the method, `this` refers to the instance that is the target of the method's operation call — that is, the instance in the context of which the method is being executed (the hosting instance). It cannot be redirected to refer to another instance. Static operations do not have the implicitly declared variable `this`.

## Expressions

The OOIS UML native detail-level language supports expressions as in Java. The expressions include operands and operators. Expressions usually imply some computation and produce some results. The following groups of operators are supported in the OOIS UML native detail-level language:

- ❑ **Name resolution operator (`:`)** — This is used to access named elements by their qualified names as described in Chapter 7, in the section “Namespaces and Visibility.”
- ❑ **Feature access operator (`.`)** — Used to access a feature (a property or an operation) of a classifier instance. The left-hand operand must be a value that refers to a classifier instance, and the right-hand operand must be the name of a feature of that classifier.
- ❑ **Variable or slot feature access operator (`->`)** — Used to access a feature of a variable or a slot, and not of a classifier instance referred to by its value.
- ❑ **Operation call operator (`(args)`)** — Invokes an operation. The left-hand operand must refer to an operation of the target instance, while the parentheses enclose the actual arguments. The details were explained in the section “Operation Invocation,” earlier in this chapter.
- ❑ **Creation operator (`new`)** — Used for the Create Classifier Instance actions, as described in Chapter 8, in the section “Creation and Destruction of Instances.”
- ❑ **Explicit conversion (cast) operator (`((type))`)** — Used for explicit type conversions of expressions. The operand following the parentheses must be an expression resulting in a type that can be converted into the given target type. The result of this operator has the target type.

## Part III: Concepts

---

- **Arithmetic operators** — These accept only operands of scalar primitive numeric types and char. They have the same semantics as in Java. The following table shows the operators.

| Operator        | Usage                    | Explanations                                                                                                                  |
|-----------------|--------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| <code>++</code> | <code>expr++</code>      | Postfix increment/decrement. Increments/decrements the operand and results in its old value (before the increment/decrement). |
| <code>--</code> | <code>expr--</code>      |                                                                                                                               |
| <code>++</code> | <code>++expr</code>      | Prefix increment/decrement. Increments/decrements the operand and results in its new value (after the increment/decrement).   |
| <code>--</code> | <code>--expr</code>      |                                                                                                                               |
| <code>+</code>  | <code>+expr</code>       | Unary + results in the same value as the operand.                                                                             |
| <code>-</code>  | <code>-expr</code>       | Unary - results in a negative value of the operand.                                                                           |
| <code>*</code>  | <code>expr * expr</code> | Multiplication                                                                                                                |
| <code>/</code>  | <code>expr / expr</code> | Division. If both operands are integers, the result is an integer.                                                            |
| <code>%</code>  | <code>expr % expr</code> | Remainder from integer division                                                                                               |
| <code>+</code>  | <code>expr + expr</code> | Addition                                                                                                                      |
| <code>-</code>  | <code>expr - expr</code> | Subtraction                                                                                                                   |

- **Logic operators** — These accept only operands of type boolean. They have the same semantics as in Java. They always return a result of type boolean. The following table shows the operators.

| Operator                | Usage                             | Explanations |
|-------------------------|-----------------------------------|--------------|
| <code>&amp;&amp;</code> | <code>expr &amp;&amp; expr</code> | Logical and  |
| <code>  </code>         | <code>expr    expr</code>         | Logical or   |
| <code>!</code>          | <code>!expr</code>                | Logical not  |

- **Bitwise and shift operators** — These accept only operands of scalar primitive numeric types and char. They have the same semantics as in Java. The following table shows the operators.

| Operator           | Usage                        | Explanations       |
|--------------------|------------------------------|--------------------|
| <code>&amp;</code> | <code>expr &amp; expr</code> | Bitwise and        |
| <code> </code>     | <code>expr   expr</code>     | Bitwise or         |
| <code>^</code>     | <code>expr ^ expr</code>     | Bitwise xor        |
| <code>~</code>     | <code>~expr</code>           | Bitwise complement |

| Operator | Usage                             | Explanations                                 |
|----------|-----------------------------------|----------------------------------------------|
| <<       | <code>expr&lt;&lt;expr</code>     | Left shift                                   |
| >>       | <code>expr&gt;&gt;expr</code>     | Right arithmetic shift (preserves the sign)  |
| >>>      | <code>expr&gt;&gt;&gt;expr</code> | Right logical shift (filling with zero bits) |

- ❑ **Relational operators** (`<`, `<=`, `>`, `>=`) — These accept only scalar primitive numeric types and `char`, and result in `boolean`. If the relationship is satisfied, the result is `true`; otherwise, it is `false`.
- ❑ **Equality operators** are `==` (equal to) and `!=` (not equal to) — These accept operands of all types. The result is a `boolean`. The equality is value-based for operands of data types and scalar primitive types, and identity-based for class types.
- ❑ **Conditional operator** (`? :`) — This is an `if-then-else` as a ternary operator. The expression `a?b:c` results in `b` if `a` is `true`, and in `c` if `a` is `false`.
- ❑ **Assignment operator** (`=`) — This has the semantics of write actions for slots of classifier instances and for variables.
- ❑ **Compound assignment operators** (`+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `>>>=`, `&=`, `^=`, `|=`) — The expression `a+=b` has the same meaning as `a=a+b`, except that `a` is evaluated only once. The rest of the rules are the same as for the elementary operators. This also holds true for the other compound assignment operators.

Every expression has its type. The typing rules for the expressions that deal with scalar primitive types are the same as in Java. Implicit conversions are applicable as in Java.

The precedence of operators and their grouping is the same as in Java. The order of execution is the same as in Java. The operator `::` has the highest priority. The operator `->` has the same priority as the dot operator `(.)`. Both are left-associative (group from left to right). Parentheses that enclose sub-expressions may change the default order of evaluation, as usual.

If any operand is `null` or undefined, the result is undefined. The effect depends on the implementation — it may raise an exception or simply behave unpredictably.

## Control Flow Statements

Control flow statements are used to direct the order in which other statements are executed — that is, to test conditions and execute statements based on the results of those conditions. The OOIS UML native detail-level language supports all control flow statements from Java. These are briefly summarized here.

### if-else

This is the conditional statement, the usual `if-then-else` construct. Its syntax is:

```
if (expression) statement1 else statement2
```

The expression within parentheses must result in a `boolean` value. If its value is `true`, then `statement1` is executed; `else statement2` is executed. The `else` part is optional. If it is missing and the result of `expression` is `false`, control passes to the next statement.

## Part III: Concepts

---

Each of the statements *statement1* and *statement2* can be a single elementary statement, in which case it is terminated with a semicolon, or a compound statement (block). If more than one statement must be executed in one of the branches, a block should be used.

The `else` part binds to the nearest previous unclosed `if`. Consider the following example:

```
int [*]{ordered} array = ...;
int sz = array->size();
int sum = 0;
if (sz>0)
 for (int i=0; i<sz; i++)
 if (array->at(i)>0) sum+=array->at(i);
else
 sum=-1;
```

Although this code looks like the `else` part belongs to `if (sz>0)`, this is just an illusion because of indentation, and indentation has no semantics (it is ignored by the compiler). Actually, this code is completely equivalent to the following:

```
int [*]{ordered} array = ...;
int sz = array->size();
int sum = 0;
if (sz>0)
 for (int i=0; i<sz; i++)
 if (array->at(i)>0) sum+=array->at(i);
 else sum=-1;
```

Because it is probably not what was intended, to bind the `else` clause to the proper `if`, a block should be used:

```
int [*]{ordered} array = ...;
int sz = array->size();
int sum = 0;
if (sz>0) {
 for (int i=0; i<sz; i++)
 if (array->at(i)>0)
 sum+=array->at(i);
}
else sum=-1;
```

You could similarly use the following:

```
int [*]{ordered} array = ...;
int sz = array->size();
int sum = 0;
if (sz>0)
 for (int i=0; i<sz; i++) {
 int val = array->at(i);
 if (val>0) sum+=val;
 }
else sum=-1;
```

## switch

A switch statement evaluates an expression that results in an integer or an enumeration. The result is used to find an appropriate case label among those listed inside the following block. If a matching case label is found, control is transferred to the first statement following the label. If not, control is transferred to the first statement following a default label. If there is no default label, the entire switch statement is skipped. The expression following each case must be a constant expression and must evaluate in a unique value. There can be at most one default label within a switch. Here is an example:

```
char c = ...;
int i;
switch (c) {
 case '0': case '1': case '2': case '3': case '4':
 case '5': case '6': case '7': case '8': case '9':
 i = c-'0'; break;
 case 'A': case 'B': case 'C':
 case 'D': case 'E': case 'F':
 i = c-'A'+10; break;
 case 'a': case 'b': case 'c':
 case 'd': case 'e': case 'f':
 i = c-'a'+10; break;
 default:
 i = 0; break;
}
```

Pay attention to the fall-through semantics of switch. Control is transferred to the first statement following the matching case, and falls through the following case. If this is not needed (that is, if control should exit the switch after a case branch is processed), a break should be used to terminate the branch. The break statement exits the first enclosing switch.

The fall-through semantics are a potential cause of errors. Namely, the following code would not behave as it was intended because i will always be 0 in the end:

```
switch (c) {
 case '0': case '1': case '2': case '3': case '4':
 case '5': case '6': case '7': case '8': case '9':
 i = c-'0';
 case 'A': case 'B': case 'C':
 case 'D': case 'E': case 'F':
 i = c-'A'+10;
 case 'a': case 'b': case 'c':
 case 'd': case 'e': case 'f':
 i = c-'a'+10;
 default:
 i = 0;
}
```

## while and do-while

The while statement has the following form:

```
while (expression) statement
```

## Part III: Concepts

---

The expression must evaluate to a boolean. The statement can be a block and represents the body of the loop. This loop is a usual “while-do” loop with the exit at the top. The expression is evaluated and, if it results in `true`, the statement is executed repeatedly until the expression evaluates to `false`. Consequently, the body of the loop might execute zero or more times.

The `do-while` loop is a loop with the exit at the bottom:

```
do statement while (expression);
```

Now, the Boolean expression is evaluated after the body of the loop is executed. Consequently, the body of the loop is executed at least once, and then is repeated while the expression evaluates to `true`. In fact, it is equivalent to the following:

```
statement
while (expression) statement
```

`for`

The `for` statement has the following form:

```
for (init_expr; boolean_expr; incr_expr) statement
```

It is equivalent to the following, except that `incr_expr` is always executed if a `continue` is encountered within the loop:

```
{
 init_expr;
 while (boolean_expr) {
 statement
 incr_expr;
 }
}
```

The initialization expression `init_expr` can be a comma-separated list of expressions that are evaluated left-to-right, or a declaration. It is computed only once, before the loop is entered. The loop is a “while-do” loop with the exit on the top. `boolean_expr` is evaluated at each iteration and the loop is terminated once it results to `false`. At the end of each iteration, the `incr_expr` is computed. It can also be a comma-separated list of expressions that are evaluated left-to-right.

This form provides a compact means to construct loops that iterate a variable over a range of values until some logical end to that range is reached. The range can be a numeric interval or a linear data structure (like a collection), or another non-linear structure (like an object graph). For example, the following is the idiom for a loop that iterates an integer variable through the range from 0 to  $n-1$  (inclusive):

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

`init_expr` and `incr_expr` can be lists of expressions, allowing several iteration variables:

```
int i, j;
for (i=0, j=n-1; i<=j; i++, j--) {
 //...
}
```

All of the expressions in the `for` construct are optional. If `init_expr` or `incr_expr` is left out, its corresponding part in the loop is simply omitted. If `boolean_expr` is left out, it is assumed to be true. For example, this is a “forever” loop:

```
for (;;) ...
```

Such a loop is usually terminated in some other way (for example, using a `break` statement within its body).

## break

A `break` statement exits the innermost `switch`, `while`, `do`, or `for` construct. That is, when a `break` is executed, control is transferred behind the first enclosing `switch` or `loop`, as in the following example:

```
for (int i=0; i<n; i++)
 if (array->at(i)==0) break;
 // array->at(i) is 0 here if i<n
```

The `break` statement could be used to terminate an outer loop, too. If this is needed, a label should be put in front of that loop and the `break` statement should refer to the label, as shown in the following example:

```
outer:
 for (...) {
 for (...) {
 for (...) {
 ...
 if (...) break outer;
 ...
 }
 }
 }
```

## continue

The `continue` statement can be used within a `while`, `do`, or `for` loop. It skips to the end of the loop’s body and evaluates the Boolean expression that controls the loop. In case of a `for` loop, it skips to `incr_expr`, as in the following example:

```
for (...) {
 ...
 if (...) continue;
 ... // This part is skipped if continue is executed
}
```

A `continue` can refer to a label just as a `break`, being applied to an outer instead of the innermost loop.

## **Output Parameters and Return**

A method implicitly terminates when its outermost compound statement (block) is completed. However, a method can be terminated explicitly by a simple `return` statement:

```
return;
```

## Part III: Concepts

---

If the method's operation has the output parameter of `return kind`, the value of that output parameter can be set to the value of the expression given behind `return`:

```
return val-1;
```

The expression behind the `return` statement must have a type that can be assigned to the type of the `return` parameter. When the `return` statement is encountered during the execution of the method, the expression is evaluated and its result is assigned to the `return` argument. Then the method terminates.

When the method terminates, in instances when the operation call was synchronous, the output arguments are transferred back to the caller, the control is returned to the caller, and its synchronous Call Operation action completes. In the case when the call was asynchronous, the termination of the method simply terminates its thread of control and the output arguments are discarded.

Output parameters of the method's operation are visible within the method as local variables implicitly declared at the beginning of the outermost block of the method. Values can be assigned to these variables during the execution of the method as with all other variables declared within the method. The ultimate values they have on method's completion (that is, on `return` or on exit from the outermost block) will be returned to the caller as output arguments. For example, consider the following operation:

```
Course::bestStudent(return student:Person, out name:Text, out grade:Text)
```

Its method can look like this, assuming that the class `Person` has an attribute `overallGrade` of type `Integer` having always a non-negative value:

```
grade = 0;
name = "";
student = null;
this.students->forEach(s) {
 int g = s.overallGrade.toInt(); // Integer::toInt() returns an int
 if (g>=grade) {
 student = s;
 name = s.name;
 grade = g;
 }
}
```

If an output parameter has a default value in the operation's definition, that parameter's default value is used as the initializer of the implicit variable in the method.

### Section Summary

- ❑ A *compound statement* (or a *block*) is a sequence of statements enclosed in curly braces {}.
- ❑ A *declaration* introduces one or more local variables into the method. The declaration declares the variables' names, type, multiplicity, uniqueness, and ordering.

- The type of a variable can be a classifier (class or data type) from the UML model, or a Java built-in primitive type.
- Every method of a non-static operation has an implicitly declared local variable named `this`. During the execution of the method, it refers to the instance that is the target of the method's operation call (the hosting instance).
- The OOIS UML native detail-level language supports expressions as in Java. The expressions include operands and operators. Expressions usually imply some computation and produce some results.
- The following control-flow statements are supported as in Java: `if-else`, `switch`, `while`, `do-while`, `for`, `break`, and `continue`.
- A method terminates when its outermost compound statement (block) is completed or when a `return` statement is executed.
- If the method's operation has the `output` parameter of `return` kind, the value of that output parameter can be set to the value of the expression given behind `return`.
- Output parameters of the method's operation are visible within the method as local variables implicitly declared at the beginning of the outermost block of the method. The ultimate values they have on the method's completion (that is, on `return` or on exit from the outermost block) will be returned to the (synchronous) caller as output arguments.

## Exceptions and Exception Handling

During the execution of a method, many things may go wrong. The object space the method is dealing with can be in an unexpected state, a value of an input parameter can be unacceptable, or an invoked operation can terminate irregularly. Exception handling is an efficient mechanism for dealing with exceptional cases in a robust way, without cluttering the code for the main regular path of execution.

### ***Notion of Exception and Exception Handling***

During the execution of a method many things can go wrong, such as the following:

- The input parameters may have unexpected or unacceptable values.
- The called operations may return unexpected or unacceptable values in their output parameters.
- Constraints (including preconditions and postconditions that prevent the previous two cases from occurring) may fail.
- The objects the method operates upon can be in unexpected or inappropriate states.
- An action may fail (for example, because of a hardware or communication failure).
- The running method may be unable to produce an appropriate result for any reason whatsoever.

## Part III: Concepts

---

To be robust, the system must handle all such cases. The old fashioned approach to coping with such problems would assume checking the values, the state of the system, and all parameters and returned values, and undertaking certain steps in one case or the other. Basically, this means using `if` statements along the main control path in a method, as shown here:

```
boolean status;
status = anObject1.oper1(...);
if (!status) {
 // handle an error in oper1
}
status = anObject2.oper2(...);
if (!status) {
 // handle an error in oper2
}
// etc.
```

Such an approach has many obvious drawbacks:

- Dealing with exceptional cases at the same time and place as with the regular case is tedious and error-prone — an exceptional case may be easily missed.
- The code for handling exceptional cases is mixed up with the code for the regular main path. The code becomes cluttered and difficult to understand and maintain.
- To provide the information about the status of their execution, operations should reserve special output parameters (usually return values) for that status.

Instead of this obsolete approach, modern programming languages provide a better means for *exception handling*. In that approach, an *exception*, which is simply an instance of a certain type, can be *raised* (or *thrown*) at the place where some irregular situation is encountered. When an exception is thrown, control immediately passes to the code that is found as the *handler* for that exception that *handles* (or *catches*) the exception. For instance, the previous example could be rewritten in the following construct that uses exceptions:

```
try {
 anObject1.oper1(...);
 anObject2.oper2(...);
 // the main path...
}
catch (Exception1 e1) {
 // handle an exception in oper1
}
catch (Exception2 e2) {
 // handle an exception in oper2
}
```

As you can see, the main path of control, which deals with the regular case, is not cluttered with the code that handles exceptions. Instead, it is written as if the exceptions will not occur. It is just surrounded by a `try` block, which means that the code within the block may raise an exception. If no exceptions are raised, the entire `try` block terminates regularly and the `catch` parts are skipped. If, however, an exception is raised during the execution of the `try` block, the execution of that block is stopped and control is transferred to the first `catch` block that follows it and that can accept the raised exception as a value (according

to usual type conformance rules). The catch blocks are, thus, handlers of the appropriate exceptions. If there is no appropriate handler following the try block that can handle the thrown exception, the exception is passed on to a handler in a try block that dynamically encloses this block, as described shortly.

This approach makes the programming task much easier. The programmer may concentrate first on the regular, main path of execution, without taking care of exceptional cases. Then, once the main path is constructed, the programmer can carefully analyze what can go wrong along that main path and provide a handler for each such exception. The handlers (that is, the catch blocks) are clearly separated from the main path. If an exception cannot be handled in the same method, it also can be propagated to the caller, as described shortly.

## Section Summary

- ❑ An exception, as an instance of a certain type, can be *thrown* (or *raised*) at the place where some irregular situation is encountered. When an exception is thrown, the control immediately passes to the code that is found as the *handler* for that exception.

## Exception Types

An exception is an instance of a certain type. Standard UML and OOIS UML allow exceptions of any type — a class or a data type, or even a native type of the host implementation language. An implementation can, however, impose additional restrictions on the types of exceptions, respecting the restrictions of the host language. For example, if Java is used as the host implementation language, all exception types must be specializations of the `Throwable` class from Java.

Conceptually, a raised exception is a notification of an occurrence of an unexpected error condition during the execution. Additionally, an exception, as an instance of its type, may carry the information about that error condition (for example, its cause, place of fault, and so on). For these reasons, the occurrence of an exception at run-time per se is not the only information. Its type, state, and possible identity also carry the information about the error condition. In general, this is why all modern OO languages allow exceptions to be instances of classifiers.

The type of the exception is the first carrier of information about the error condition because exceptions are caught according to their type. Introducing a new type of an exception, even without adding some properties to it, allows a programmer who cares only about such an exception to catch it exclusive of other types of exceptions that might have a common generalization. In general, new types of exceptions should be created when programmers want to handle one kind of exception and not another by simply relying on their type in a `catch`, instead of relying on different values of their properties to determine whether they really care about an exception, or it is simply caught by accident and is irrelevant.

Another carrier of information about the error condition could be the values of the properties of an exception. The slots of an exception can carry the information about the cause, place, time, significance, and other characteristics of the error condition. Additionally, it is often useful to provide a human-readable description of the exception. Such a description can be provided in a textual attribute of the

## Part III: Concepts

---

exception type, or through an operation that returns a sentence that is dynamically constructed from the static strings and dynamic values of the properties, as in the following example:

```
"The value for the key " + key + " does not exist."
```

Such a description can be used as an error message ultimately displayed to the user.

If an exception is an instance of a data type, it has no identity, but its pure value (including its attributes) carries the information as described. When such an exception is handled, it is discarded as any other data type instance as soon as nothing refers to it. Conceptually, such exceptions are only significant until they are handled. On the other hand, an exception can be an object of a class, having its identity. Such an object has to be destroyed by an explicit Destroy Object action and thus may survive its handling. Such exceptions may be used for more important error conditions that should be logged for later analysis. Note that there is no need for any special mechanism for supporting persistence and logging of such exceptions — OOIS UML objects of classes inherently persist until they are destroyed by an action. This means that an exception that is an object of a class is implicitly logged and may survive the thread that raised and handled it.

### Section Summary

- Standard UML and OOIS UML allow exceptions of any type — a class or a data type, or even a native type of the host implementation language.
- An implementation can impose additional restrictions on the types of exceptions, respecting the restrictions of the host implementation language.
- Conceptually, a raised exception is a notification of an occurrence of an unexpected error condition during the execution.
- The type, identity, and properties of an exception may carry the information about the occurrence of the error condition (for example, its cause, place, time, significance, description, and so on).

## **Throwing and Catching Exceptions**

An exception can be explicitly thrown using the `throw` statement:

```
throw expression
```

The expression in `throw` must result in a single value. That value, referring to an instance of a type, is thrown as an exception.

An exception can also be propagated from a synchronous operation call because it is generated within the method of that operation. Actions can also throw exceptions in error conditions, as described throughout the book. Finally, a constraint that is not satisfied after an action is executed, or a failed precondition or postcondition, also throw exceptions.

When an exception is raised, control is immediately passed to the corresponding handler that can catch that exception. Exceptions are caught in a `try-catch` block that has the following general form:

```
try
 block
 catch (exception_type identifier)
 block
 catch (exception_type identifier)
 block
 ...
 ...
```

There can be any number of `catch` clauses, including none.

The `try` block is executed first as a usual compound statement. If no exceptions occur during that execution, the entire `try-catch` block completes and control passes to the next statement (the `catch` blocks are skipped). During the execution of the `try` block, an exception can occur in one of several ways:

- ❑ It can be thrown explicitly by a `throw` statement executed within that block.
- ❑ It can be raised by an executed action (for example, as a consequence of a failed constraint).
- ❑ It can be propagated from the execution of the method of a synchronously called operation.

When an exception is thrown, the execution of the `try` block is aborted and the `catch` clauses are examined in order to find one that can accept the exception. The type matching rules for exceptions are usual — substitution is always assumed (that is, a `catch` with a generalizing type can accept an exception of a specializing type).

If such a `catch` clause is found, control passes to the block of that clause. That block is executed as any other compound statement. If an adequate `catch` is not found, the exception is passed out from the `try-catch` block to an enclosing `try-catch` block (if any) and a `catch` in that block is searched for. If an exception is not caught within the entire method, it is propagated out from the method:

- ❑ If the call of the operation is synchronous, the exception is propagated out from the method to the caller, and out from the Call Operation action that invoked that operation, so that the exception occurs within the caller's block that executed that action.
- ❑ If the call of the operation is asynchronous, the exception is handled by a default handler provided by the implementation.

Note that the notion of the “main starting point” of the execution is not defined in OOIS UML, and it is the responsibility of the execution environment to start the “main thread of control.” Therefore, it is the responsibility of the implementation to provide a default exception handler for uncaught exceptions.

The block of a `catch` clause can be considered as a recovery code. It can try to recover from the error condition signaled by the caught exception and continue the execution, or it can just clean-up the damage and re-raise the same or another exception with a `throw` statement.

Each `catch` clause in a `try-catch` construct is examined in turn, from first to last, to see whether the type of the exception conforms to the type declared in the `catch` clause. When such a `catch` clause is found,

## Part III: Concepts

---

its block is executed with its *identifier* set to the value passed as the exception. The identifier behaves like a local variable declared in the catch block. The remaining catch clauses are not evaluated further. This is why the order of the catch clauses and the types they accept are significant. For example, if `Base` is a generalizing type of `Derived`, the catch clause accepting a `Derived` will never be executed in this example:

```
try {
 // ...
}
catch (Base aBase) {
 //...
}
catch (Derived aDerived) {
 //...
}
```

If a catch block raises an exception, the exception is propagated out from the try-catch block as if it were thrown anywhere else in the try block and not caught by any catch.

### Section Summary

- ❑ An exception can be explicitly thrown using the `throw` statement.
- ❑ During the execution of a method, an exception may occur in one of several ways:
  - ❑ It can be thrown explicitly by a `throw` statement.
  - ❑ It can be raised by an executed action (for example, as a consequence of a failed constraint).
  - ❑ It can be propagated from the execution of the method of a synchronously called operation.
- ❑ Exceptions are caught in a try-catch block.
- ❑ When an exception is thrown, the execution of the try block is aborted and the catch clauses are examined in order to find one that can accept the exception. If such a catch clause is found, control passes to the block of that clause. If an adequate catch is not found, the exception is passed out from the try-catch block to the enclosing try-catch block (if any) and a catch in that block is searched for. If an exception is not caught within the entire method, it is propagated out from the method to the synchronous caller.

## ***Declaring Exceptions Thrown by Operations***

It is clear that the types of the exceptions that can be propagated to the caller from a called operation constitute an important part of the operation's signature. The caller should know what to expect from the operation in exceptional cases, just as much as the caller must know what the operation returns as its output parameters on a regular completion.

This is why UML provides a means to specify the types of the exceptions that a behavioral feature of a classifier can raise. This is done by relating a behavioral feature (such as operation) with a collection of types in the model. Maintenance of these relationships is the responsibility of the modeling tool. There is no special notation for the exceptions declared for an operation in the textual representation of the operation.

Of course, if an operation declares the type  $T$  as its exception, it can also raise an exception of any subtype of  $T$ . For that reason, if the operation throws exceptions of several types that have  $T$  as their common supertype, it is sufficient to declare only  $T$  as the exception type. However, this is not always a good idea because it hides potentially useful information from the caller. It is better to provide as specific and complete information about the raised exceptions as possible. In particular, if the operation can throw instances of all direct specializations of  $T$ , it is correct to declare only  $T$  as an exception. But if it throws only some of them, it is better to list them all.

The semantic effect of the declarations of an operation's exception types is a semantic variation point in UML, as well as in OOIS UML. Basically, these declarations have at least a documentational purpose, because they provide useful information about the behavior of the operation in exceptional cases. Declarations of exceptions, however, can also have some impact on the semantic correctness of the code of methods. It is up to the implementation to define that semantic impact.

For example, an implementation that uses Java as the host implementation language can enforce stricter rules for interpreting exception declarations. A method of an operation must not propagate an exception of a type not declared in the operation's interface. This means that a method of an operation  $op1$ , which synchronously calls an operation  $op2$  that declares a type  $T$  as its exception, has only three choices:

- Catch the exception of type  $T$  and handle it, without propagating it to the caller.
- Catch the exception of type  $T$  and map it into one of the exceptions that  $op1$  declares (that is, throw one of these in the handlers for  $T$ ).
- Declare  $T$  as its exception type and let the exception propagate to the caller.

Such rules promote declarations of exceptions to a kind of a contractual obligation of the operation. It improves the robustness of the system because the exceptions become part of the contracts between the participants of interaction, strictly checked at compile time.

However, execution of actions can raise many types of exceptions thrown by the execution environment. Declaring all these types explicitly for almost every operation may be really impractical. This is why Java introduces the categorization of exception types into *checked* and *unchecked* exceptions. Checked exceptions must be explicitly declared and the compiler checks the foregoing strict rules. In particular, if an operation does not declare any checked exception, it must not throw or propagate any. Unchecked exceptions, on the other hand, do not need to be declared, and they can occur and be propagated anywhere — the caller should expect an unchecked exception at any operation call.

C++, on the other hand, uses a less strict approach. It does not distinguish between checked and unchecked exceptions. If an operation declares some exception types, it must not throw exceptions of any other type. If, on the other hand, an operation does not declare any exception type, it can throw any exception.

To allow more direct implementations in these different languages, OOIS UML leaves the interpretation of exception declarations as a semantic variation point.

### Section Summary

- ❑ UML provides a means to specify types of the exceptions that a behavioral feature of a classifier can throw.
- ❑ Interpretation of the declarations of exception types for a behavioral feature is a semantic variation point in UML and in OOIS UML.

## Concurrency and Fault Tolerance

In a real information system, many running applications may access the same object space concurrently. They compete to access objects and change their states by performing actions. Their concurrent execution must be controlled in a way that preserves the consistency of the object space, as defined by the semantics of OOIS UML.

During the execution of a real system, many irregular events can occur, including failures in hardware, network, and software. A robust system must cope with all such events, trying to preserve the consistency of the object space.

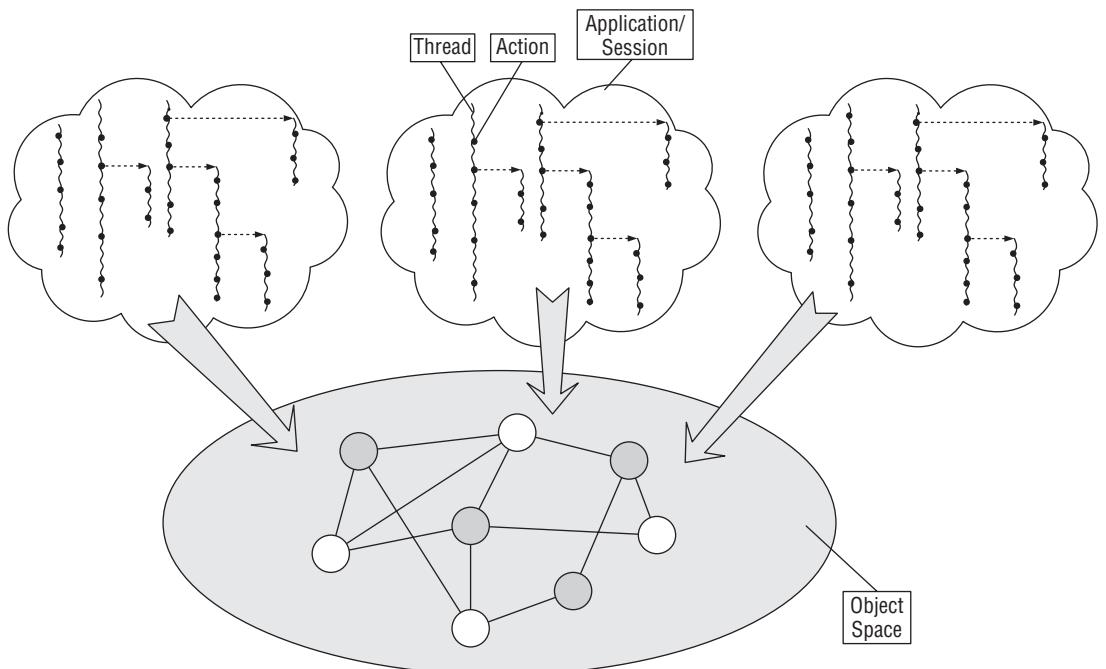
The traditional concepts of locking and transactions that provide efficient solutions to these problems are adapted to the OO nature and semantics of OOIS UML, and are described in this section.

### Concurrency Model in OOIS UML

Figure 13-6 depicts the OOIS UML concurrency model. The object space (consisting of objects, their attribute values, and links) provides the underpinning structure for applications that run simultaneously and access that object space. Applications are programs that perform some processing upon the object space as the central structural repository. Applications may interact with the users through a GUI, or may perform some batch processing, without interacting with the users. The applications access the object space by means of executing actions that read or modify the structure. The notion of application has no additional formal meaning in the context of OOIS UML and is intentionally left undefined to cover different cases of running programs in practice.

The behavior executed by applications consists of methods. An activated method executes a sequence of actions in the order determined by the control flow statements from the method. The method may invoke an operation synchronously, when the control passes to that operation's method and returns to the caller when the invoked method completes. The sequence of actions executed within methods dynamically nested by synchronous operation calls is referred to as a *thread*. Actions in a thread are executed sequentially, or at least in a manner that preserves the semantics of their sequential execution.

A method may invoke an operation asynchronously. When this happens, the caller method continues the execution of its actions in the same thread, while the asynchronous call starts a new concurrent thread of actions performed by the invoked method and its nested methods activated by synchronous operation calls. As a result, an application execution may encompass many concurrent threads, each executing a sequence of actions upon the common underpinning object structure.



**Figure 13-6: The OOIS UML concurrency model.** Simultaneous applications or sessions execute concurrent threads. Threads are sequences of atomic actions that access the underlying object space, initiated with asynchronous operation calls.

Asynchronous operation calls are the only initiators of threads. The execution environment is responsible for initiating one or more “main” threads for an application, which will then initiate the others. A main thread may be the one that processes the user’s actions through the GUI, or the one that processes requests that arrive to the middle application tier over the Internet. Usually, these main threads are not visible to the developer. Instead, the developer’s responsibility starts from the implementation of commands issued on these requests, which may then call other operations synchronously or asynchronously.

The effect of the actions executed in one thread is always as if they had been executed sequentially, one after the other, in the total order dictated by the control flow statements. This does not necessarily mean that the actions must really be executed in that particular order, or even sequentially. The compiler or the execution environment can schedule the actions upon the object space in any order, or execute them even in parallel, provided that their ultimate effect on the object space is preserved. For example, if two actions are completely independent, they can be executed in any order or even in parallel. However, adopting the sequential semantics of actions in a thread makes the definition and understanding of cumulative effects of actions easier.

Actions access objects, attribute values, and links, and read or modify the state of the object space. The actions from the same thread do this one after the other, without competing for the same structural resources at the same time. Actions from different threads in the same or different applications, on the other hand, run concurrently and may compete for the same elements of the object space as for shared resources. Consequently, they must access the shared object space in a controlled way that preserves the semantic consistency of the structure, as will be described later in this chapter.

## Part III: Concepts

---

The object space is a shared structure. Actions that access that object space, and concurrent threads that execute sequences of actions, are the only elements of the OOIS UML concurrency model. Other coarse-grained concepts (such as session or application) do not have semantic impact on the concurrency model, and are, thus, not important in OOIS UML. Instead, these concepts have their meaning in the architectural context, as elements for grouping threads into those initiated within interactions with one user (session) or handled within one address space of an executable program or a coherent set of features (application).

The described OOIS UML concurrency model is completely analogous to the commonly accepted model implied by the architectures of database applications usually deployed in practice. In those systems, the underpinning structure resides in a relational database. The behavior is, however, also embodied in concurrent threads that execute actions upon that structure. Instead of the relational database, OOIS UML uses object space.<sup>4</sup> Such a model makes the traditional concurrency and fault tolerance concepts (proven and widely adopted in practice) reusable for OOIS UML.

### Section Summary

- ❑ The OOIS UML concurrency model includes the following characteristics:
  - ❑ The underpinning structure is provided by the object space.
  - ❑ All actions access the same object space as a commonly shared structure.
  - ❑ A *thread* is a sequence of actions executed within methods dynamically nested by synchronous operation calls.
  - ❑ Threads run concurrently and are initiated by asynchronous operation calls.

## Concurrency Control

An action performs reading or modification of the object space in an atomic manner. In other words, the execution of an action is *isolated* from any influence of other actions executed concurrently with that action. This means that an action upon the object space has an effect as if there were no other concurrent actions around it. For example, consider the following action:

```
anEmployee.company->replace(oldCompany, newCompany)
```

This action consists of two elementary structural modifications. First, the existing link between `anEmployee` and `oldCompany` is deleted, and second, a new link is established between `anEmployee` and `newCompany`. The atomic nature of action execution ensures that another action cannot modify the objects referred to by `anEmployee`, `oldCompany`, and `newCompany` between the time or at the same time these two elementary modifications are made. For example, it cannot happen that the considered action removes the link with `oldCompany`, and then another concurrent action deletes the object referred to by `anEmployee`, so that the new link with `newCompany` cannot be created.

---

<sup>4</sup>Once again, this does not prevent an implementation from using a relational database to store the object space. It simply means that OOIS UML does not deal with that lower level of abstraction of a relational database.

In brief, the semantics of isolated execution of every action means that the action's effect on the object space is as if all other concurrent actions were executed before or after the considered action.

OOIS UML defines the semantics of isolated action execution, but it does not prescribe the way it should be implemented. Of course, the traditional concept of *locks* can be used for implementation. Before an action performs any access to the object space, it locks one or more objects that it will involve in its execution. When an object is locked, other actions cannot acquire locks on the same object, and the considered action is sure that the object will not be modified during its execution. When the action completes, it unlocks the objects it has locked, so that other actions can execute.

Isolated execution of individual actions is not sufficient. Consider the method for the operation `Course::calcRevenue` mentioned in Chapter 5, which calculates the revenue from a course by summing up the payments from its students:

```
double sum = 0.0;
this.students->forEach(s)
 sum += s.getPriceCoef().toDouble();
sum *= this.price.toDouble();
return new Real(sum);
```

The method for the operation `Person::getPriceCoef` reads the slot `attendedCourses` of the object of class `Person`, and returns the price coefficient according to the number of objects linked on that slot. In summary, this simple piece of code may encompass execution of many actions that read the underlying object space in a single thread. Although each of them is executed as isolated, it may easily happen that between two executed actions an action from another thread modifies the objects under consideration and affects the result of this method.

For example, an action from another thread can add or remove a link to the property `attendedCourses` of some of the students of the object referred to by `this`, may add or remove a link from `this.students` so that the `forEach` loop will traverse the incorrect collection of objects, or it can even delete some objects in the collection `this.students` so that the operation call `s.getPriceCoef` fails. In the worst case, even the object referred to by `this` can be deleted by a concurrent action.

Here is another example of a similar potentially irregular case:

```
if (aTeacher.taughtCourses->size()<3)
 aTeacher.taughtCourses->add(aCourse);
```

The intent of the code is to add a course to the collection of courses taught by the given teacher, provided that the addition would not violate the multiplicity constraint of `taughtCourses`. However, the code does not actually fulfill this intention. It may happen that another action from a concurrent thread is executed between the Read Property action in the `if` condition and the Create Link action in the second row. That action may simply create another link between the same two objects.

These examples illustrate how non-atomic execution of groups of actions can lead to an inconsistent state because of a *race condition*. A race condition is an error condition caused by incorrectly interleaved actions from concurrent threads. A race condition usually happens in the cases like the one given in the last example. An action that modifies the object state is performed conditionally, while the action that tests the condition and the modification action are not performed atomically.

## Part III: Concepts

---

There are many similar examples of race conditions. In general, almost any processing of the object state that includes dependent actions on several objects, with or without modifications, may lead to a race condition. It follows that there is a need to *isolate an entire group* of actions. ODIS UML supports the notion of *isolated action group* in a way that will be described later in this chapter. The actions in such a group are executed as if there were no interleaving of other actions during the execution of the group.

Isolation of a group of actions can be achieved by the following locking policy. When an action in an isolated group is executed, it locks the objects as usual, but it does not release its locks when it completes. The next action in the sequence acquires its locks, and so on. When the entire group is completed, all locks acquired during the execution of the group are released. If isolated groups are nested, the locks are released when the outermost isolated group is completed. Of course, if an object is already locked by a previous action in the same thread, an action in the same thread immediately gets the lock. In fact, locks belong to the thread and not to an action.

The concrete implementation can select one of many variants of locking. One aspect of locking is exclusiveness. A locking policy may provide the sole concept of exclusive (or non-shared) lock with binary logics — an object can be either locked or unlocked. If an object is locked by one thread, another thread cannot get a lock on the same object until the first thread releases the lock. However, such a policy may be too restrictive. If an action simply reads and does not modify an object, why shouldn't another action read the same object, too, as long as it does not modify it?

For that reason, a policy may introduce two kinds of locks: a *read* (or shared) lock and a *write* (or exclusive) lock. An action that only reads an object acquires a read lock; one that modifies an object acquires a write lock. A read lock can be obtained only if there is no write lock on the object. A write lock can be acquired if there is no other write or read lock on the object. In that way, a reader may be sure that no one else is writing the same object while it is reading the object, but many readers can concurrently read the object, holding read locks on it. This policy is usually referred to as the *many readers-single writer* policy. It may significantly improve concurrency of the system because of reduced conflicts on the objects (when several actions wait for a lock on an object).

Another aspect of the locking policy is the scope of a lock. Basically, the entire object can be locked, protecting all its slots. However, a locking policy can introduce different kinds of locks for actions that affect attributes or links, allowing concurrent access to attributes and links of the same object.

Whatever the policy is, it should allow that the same thread gets a more restrictive lock if it already holds a lock on the same object, and if there are no other threads that prevent that. For example, if a thread holds a read lock on an object, another action in the same thread can obtain a write lock on the same object, provided that there are no threads that hold any other lock. Similarly, if a thread holds a lock on an attribute of an object, it can obtain a lock on the entire object, provided that there are no other threads that hold any lock on any part of the same object.

What happens if an action cannot obtain the requested lock because the object is locked by another thread? It is a good idea to wait for that lock to be released so it can be acquired, but how long?

Consider the following case. A thread  $T_1$  has locked an object  $O_1$  and is waiting for a lock on an object  $O_2$ , a thread  $T_2$  has locked  $O_2$  and is waiting for a lock on an object  $O_3$ , and so on, a thread  $T_n$  has locked  $O_n$  and is waiting for a lock on an object  $O_1$ . In that situation, all threads are blocked and cannot continue their execution because there is a circular wait for the resources they are mutually holding locked. Such a case is called a *deadlock*.

Deadlocks are among the most severe error conditions in concurrent programs. Although techniques for prevention, avoidance, or detection and resolution of deadlocks exist, they are usually too costly or not applicable to real complex information systems.

One simple way to prevent a deadlock is to limit the time of waiting for a lock. When a thread requests a lock, it is blocked and a timeout interval is started. If the action acquires a lock before the time is out, the action is performed and the thread continues. If, however, the time runs out, the action must give up the execution.

When a wait for a lock is timed out, it should be treated as a failure during the execution of an action. Really, it may be a consequence of a deadlock, when the system is in an irregular state, or can be simply treated as the lack of response from the object space. This is why a locking timeout is signaled as an exception in OOIS UML.

### Section Summary

- ❑ Execution of an action is *isolated* from any influence of other actions executed concurrently with that action. The action's effect on the object space is as if all other concurrent actions had been executed before or after the considered action.
- ❑ Isolation of action execution can be implemented using the traditional concept of *locks*. An implementation can support exclusive or shared locks.
- ❑ A *race condition* is an error condition caused by incorrectly interleaved actions from concurrent threads.
- ❑ The actions in an *isolated action group* are executed as if there were no interleaving of other actions during the execution of the group.
- ❑ Isolation of a group can be achieved by releasing the locks acquired by the actions in the group when the entire group is completed.
- ❑ A *deadlock* is an error condition when a set of threads get blocked and cannot continue their execution because they are waiting for the resources that they are circularly holding.
- ❑ To prevent deadlocks, actions wait for locks for a limited time. If the time is out and the lock is not acquired, an exception is raised as a notification of a failure.

## Fault Tolerance and Transactions

Actions can fail upon execution, which basically means that the system can't accomplish the action. First, the hardware or communication lines can cause a failure that prevents the action to execute. Second, a computer that hosts the object space can refuse to respond to a client's request because of a transient failure or overload. Finally, a failure may result from the execution environment in the following cases:

- ❑ Constraint checking fails.
- ❑ A requested lock cannot be acquired and the waiting time is out, or the environment detects a concurrency conflict on a shared resource in another way.

## Part III: Concepts

---

- ❑ A reference to an object becomes dangling, because a concurrent thread has deleted the referenced object. This can happen in the following way: a variable that is local to a method refers to an object, but the thread in whose context the method is being executed has not locked the object, and another thread can delete that object. When the first thread comes to an action that involves the object referred to by the variable, it requests a lock on that object, but the request is immediately refused because the object does not exist any more. This is another case of a race condition. In the following example, the object referred to by `c` may not exist when the action in the second line is executed:

```
Corse c = new Coruse;
c.name = "OOIS UML";
```

All such failures are notified by exceptions raised by the execution environment during execution of an action. An exception is, therefore, a signal of a failure that must be properly handled if the system is supposed to survive that failure. The exception handling must ensure that the system (especially the object space) remains in a consistent state and continues with its service if possible, probably with a degraded operation. Such an approach to overcoming failures aimed at improving the reliability of the system is called *fault tolerance*.

There are two general strategies to recover the system from a failure: *forward error recovery (FER)* and *backward error recovery (BER)*.

FER tries to overcome the failure by fixing the problem and then to continue with the service. In terms of exceptions, FER, in general, means handling an exception and going on with execution:

```
try {
 // Some actions
}
catch (ExceptionType e) {
 // Handle the exception, recover the system, and go on
}
```

There are no general concepts and techniques for FER. Concrete steps taken on a failure are always ad-hoc and implemented for each particular case. FER techniques do not rely on any dedicated recovery mechanisms provided by the language or execution environment.

On the other hand, BER tries to return the system to a previous consistent state before the failure occurred. BER assumes certain concepts and special recovery mechanisms provided by the language or execution environment:

- ❑ During the execution, the system may pass through a *recovery point*. A recovery point is a point in execution of the system in which the state of the system is considered as consistent. The BER-supporting mechanism is capable of taking a snapshot of that state so that the system can be rolled back to that state in case of a failure.
- ❑ After passing a recovery point, the system performs actions and moves to a new consistent state.
- ❑ If there is a failure during the execution of the actions, the recovery mechanism can return the system to a previous consistent state at the last recovery point. Such return is called a *rollback*. The execution may then retry the same group of actions, or take another path of execution.

Principally, this may look like the follows:

```
recovery_point;
try {
 // Some actions
}
catch (ExceptionType e) {
 roll_back_to_last_recovery_point;
}
```

Traditionally, *transactions* are used for BER in information systems. In OOIS UML, a *transaction* is an isolated group of actions that inherently supports backward error recovery. The actions in a transaction execute one after the other, and acquire locks without releasing them until the entire transaction is completed. If all actions in the transaction complete successfully, the entire transaction *commits*, and the object space moves to the state determined by the effects of the actions. If an exception occurs during the execution of the transaction, the transaction can be *rolled back*, and the object space is left in the state it had had before the transaction started. Therefore, the execution of the entire transaction is *atomic*. It is always isolated, and its ultimate effect on the object space is as if all its actions have been completed successfully (if the transaction is committed), or the transaction did not start at all (if it is rolled back).

In the OOIS UML native detail-level language, a transaction is just a kind of a compound statement (block) started with the keyword `trans`:

```
trans {
 // Some actions performed atomically
}
```

Transactions can be arbitrarily nested as other blocks, statically (in the code), or dynamically (by operation invocation). An outer transaction commits when all its nested transactions commit. If any of the actions in any of the transactions fails, the entire outermost transaction rolls back. Consider the following example:

```
trans { // T1
 a1
 a2
 trans { // T2
 a3
 a4
 }
 trans { // T3
 a5
 a6
 }
}
```

The outermost transaction, `T1`, commits if all actions `a1` to `a6` succeed. If the action `a5` fails, the effects of all actions performed before it (that is, of actions `a1` to `a5`) are discarded.

If another traditional language is used at the level of details, the following approach can be used. To start a transaction, an object of a class `Transaction` from a library for that language must be created. The operations `commit` and `rollback` of this class commit or roll back a transaction. A newly created object of

## Part III: Concepts

---

Transaction is linked to the currently open innermost transaction as its child. The operation `getParent` returns the parent transaction. For example, the following Java code illustrates a transaction:

```
Transaction trans = new Transaction();
try {
 // Some actions
 trans.commit();
}
catch (Exception e) {
 if (trans.getParent()!=null) throw e;
 else trans.rollback();
}
```

The `if` statement in the `catch` block checks whether the transaction `trans` is the outermost transaction. If it is not, it has a parent, and the exception should be just passed to the `try-catch` block of that parent transaction. If it is the outermost transaction, it has no parent, and it should roll back.

In C++, the code may look something like this:

```
Transaction* trans = new Transaction(); // Prologue
try {
 // Some actions // Body
 trans->commit(); // Epilogue
}
catch (Exception* e) {
 if (trans->getParent() != 0) throw e;
 else trans->rollback();
}
```

However, because this could be a common idiom with always the same prologue and epilogue code, C++ macros could be defined to make coding easier:

```
#define StartTrans \
{ \
 Transaction* trans = new Transaction(); \
 try { \
 \
#define EndTrans \
 trans->commit(); \
 } \
 catch (Exception* e) { \
 if (trans->getParent() != 0) throw e; \
 else trans->rollback(); \
 } \
}
```

In this way, writing a transaction in C++ becomes very simple:

```
StartTrans
 // Some actions
EndTrans
```

To illustrate once more the difference between FER and BER, and between isolation and fault tolerance, let's look at an example of Java code that uses a transaction just as an isolated group of actions, but with FER instead of BER. It does not roll back the transaction, but tries to recover from the failure and commit the performed actions.

```
Transaction trans = new Transaction();
try {
 // Some actions
 trans.commit();
}
catch (Exception e) {
 // ...Handle e...
 trans.commit();
}
```

A FER without isolation would simply be the following:

```
try {
 // Some actions
}
catch (Exception e) {
 // ...Handle e...
}
```

### Section Summary

- ❑ Execution of an action can experience different kinds of failures that are manifested as inability of the system to accomplish the action. All such failures are notified by exceptions raised by the execution environment.
- ❑ There are two general approaches to recover the system from a failure: *forward error recovery* (FER) and *backward error recovery* (BER).
- ❑ FER tries to overcome the failure by fixing the problem and then to continue the service.
- ❑ BER tries to return the system to a previous consistent state it had had before the failure occurred. BER assumes that certain concepts are defined and special mechanisms are provided by the language or execution environment.
- ❑ In OOIS UML, a *transaction* is an isolated group of actions that inherently supports backward error recovery.
- ❑ If all actions in the transaction complete successfully, the entire transaction *commits*, and the object space moves to the state determined by the effects of the actions. If an exception occurs during the execution of the transaction, the transaction can be *rolled back*, and the object space is left in the state that it had had before the transaction started.



# 14

## State Machines

State machines are often very useful for modeling event-driven behavior of entities whose reaction to an incoming stimulus depends not only on the kind of that stimulus, but also on the history of the previously received stimuli — that is, on the current state of the receiver. In essence, they model lifetimes of entities in terms of states and transitions. This chapter examines hierarchical state machines as supported by OOIS UML.

### Introduction to State Machines

A state machine is a software engineering concept for modeling event-driven behavior of different kinds of entities, such as objects, components, or systems. UML has adopted this concept mostly for specifying behaviors of objects of classes as described in this section.

To allow simplified implementations, and for a shorter learning curve, the definition of state machines and their features from standard UML have been simplified in OOIS UML, although their semantics are simplified and adapted to the domain of business applications. However, their founding principles and usability are preserved.

### Motivation

Let's consider an order processing system with the conceptual model whose excerpt is shown in Figure 14-1. An *Order* has its reference number as its identifier (see Figure 14-1a), and contains a collection of *Order Items*, each of which stores the information about one occurrence of a *Product* in the Order, such as the ordered quantity of the Product, its unit price taken from the price list, and the derived total price of the ordered quantity of that Product. When an Order is being processed, one or more *Shipments* can be generated for it. In most cases, the Order can be fulfilled by only one Shipment, but in some exceptional cases, when some of the ordered Products are not available in the stock, several Shipments might be necessary to fulfill the Order. Similar to the Order, a Shipment consists of a collection of *Shipment Items*, storing the information about the actual quantity of a certain Product shipped with that Shipment.

## Part III: Concepts

---

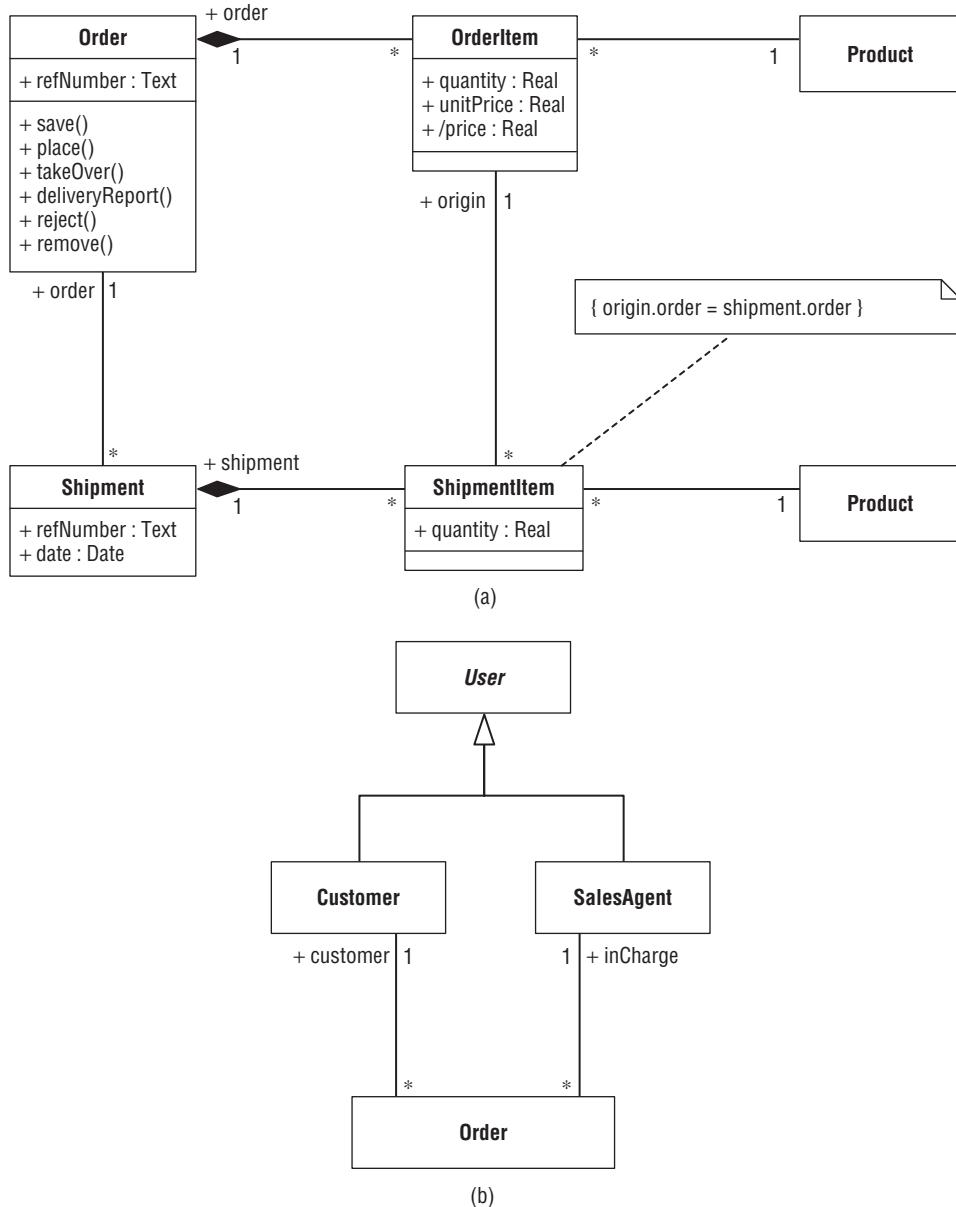


Figure 14-1: An excerpt from the conceptual model of an order processing system. (a) Orders and Shipments are indirectly associated with Products by Order Items and Shipment Items, respectively, which store the information about each particular occurrence of a Product in an Order or Shipment. (b) Customers and Sales Agents are Users of the system. A Customer can place Orders, while a Sales Agent is in charge of processing an Order.

Each Shipment Item is linked to a related Order Item — a Shipment Item is created because a certain quantity of a certain Product is ordered by the corresponding Order. The constraint attached to Shipment Item says that a Shipment Item's Shipment and its original Order Item must have the same initial Order. Note that the Product referred to by a Shipment Item does not need to be the same as the Product referred to by its original Order Item, because a Product that is an adequate replacement can be shipped instead of the ordered Product in some special cases.

Orders are placed by Customers and processed by Sales Agents (see Figure 14-1b), both of whom are the Users of the system. Each Order is assigned to one Sales Agent who is in charge of its processing.

The envisioned behavior of the system looks like this:

- ❑ A Customer can create a new Order and edit it by adding or removing Parts from it — in fact, by creating and deleting Order Items and defining the ordered quantities. Before the Order is placed, the Customer can arbitrarily modify the Order, and can “save” the Order in the system, suspend the work on that Order, and resume it later. Optionally, the Customer can remove the Order from the system if he or she wants to give it up.
- ❑ The Customer can ultimately place the created Order. The Order then goes into a pool of initiated Orders waiting to be processed.
- ❑ A Sales Agent can take one of the initiated Orders from the pool and start its processing. By taking over an Order, the Sales Agent becomes responsible for its processing (playing the role “in charge” of that Order).
- ❑ During processing, the Sales Agent must check the completeness and correctness of the Order. If the agent finds any inconsistency in the Order (according to the business rules that are out of the scope of this simplified example), the Sales Agent can reject the Order. It then becomes available to the Customer who placed it for further editing or possible removal.
- ❑ Otherwise, the Sales Agent creates one or more Shipments for the Order. If the entire Order can be fulfilled from the available stock, only one Shipment will be created. Otherwise, there will be several partial Shipments that will go through the same process.
- ❑ The creation of the Shipment object represents an order for the shipping department that organizes the actual shipment of products. When the delivery report arrives at the company and is recorded in the system (acknowledging the reception of the shipment by the Customer), the Order is declared as completed.
- ❑ Once completed, an Order can be “removed” from the system. “Removing” in this example simply means transferring the Order into a state in which it does not change any more. The Order can be deleted from the system by issuing a Destroy Object action.

This simplified example will not go into further detail for invoicing and other financial aspects of order processing.

Obviously, the given specification describes the *lifecycle* of an object of the class `order`, during which the object changes its *state* according to the incoming *event* and its current state. The event represents an invocation of one of the operations of the class shown in Figure 14-1a. An Order can be in the state *created*, *initiated*, *processing*, or *completed*, while the specification describes how the object should change its state according to the events.

## Part III: Concepts

---

For example, once being *created*, the Order can stay in the same state if it is saved by the Customer, or move to the next state, *initiated*, if the Customer chooses to place it. Similarly, once moved to the stage of *processing*, the Order can be rejected and moved back to the state *created* (and then again saved, removed, or placed), or moved to the state *completed* when the delivery report is received.

Figure 14-2 shows all this in a *state diagram* that shows a *state machine* for the described lifecycle of an Order. It shows the *states* (depicted as rounded rectangular symbols) in which an object resides during its lifetime, as well as *transitions* (shown as arrows between states) annotated with *triggers* that trigger those transitions. The filled circle indicates the *initial state*, and its outgoing transition shows that the Order goes initially to the state *Created* when it is created. The bullseye symbol indicates the *final state*, which represents the completion of the object's state machine by resting in a state in which it does not respond to triggers anymore.

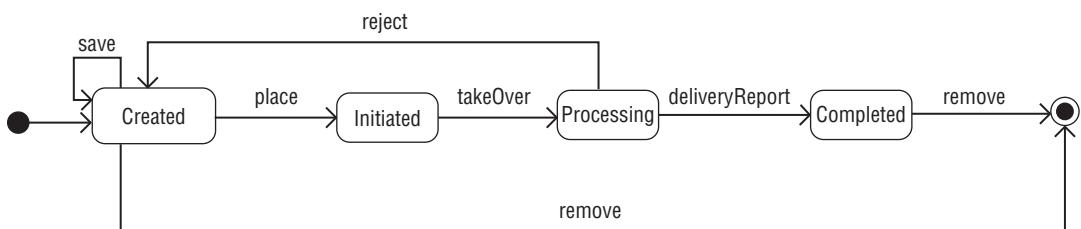


Figure 14-2: State diagram for a simple state machine that defines the lifecycle of an Order

As in other parts of UML, the diagram is only a pictorial representation of a part of the model, while the state machine is a model element that defines the behavior of objects of a certain class (Order, in this case) — that is, the response of the object on the outer stimuli. A state machine defines how the object changes its states upon triggers during its lifetime. State machines are a powerful and expressive technique for specifying behavior of an entity whose response to an outer stimulus not only depends on the kind of that stimulus, but also on the history of previous stimuli that determine the current state of the entity.

For the given example in Figure 14-2, the Order will not react and change its state if its operation *remove* or *place* is invoked when it is in the state *Initiated*, but will react adequately if it is in the state *Created*. Similarly, a created Order can be saved many times, but once it is placed, it will not respond to a *save* trigger. Actually, the user interface can even prevent users from issuing the triggers on which the state machine does not react in the current state. For example, if an Order is in the state *Created*, the user interface can offer only the *save*, *place*, and *remove* options as the triggers. A powerful user interface can do that even generically, using reflection, by consulting the underlying state machine that defines the behavior of an object, and, thus, does not require any coding by the developer when the state machine is redesigned or changed.

In general, state machines are very useful for modeling lifecycles of objects of classes that change their states upon events according to certain rules, and where the behavior or reaction of an object depends on the history of previous events (that is, on the current state of the object). For example, a vending machine reacts significantly differently upon pressing a button for selecting the beverage when enough coins have been inserted, as opposed to when no coins have been inserted.

### Section Summary

- ❑ A state machine can be used to model lifecycle and specifying behavior of an entity whose response to an outer stimulus not only depends on the kind of that stimulus, but also on the history of previous stimuli that determine the current state of the entity.
- ❑ An object whose behavior is modeled with a state machine resides in a state awaiting an event to trigger a transition to another state. The state machine defines the states, triggers, and transitions according to which the object behaves.

## ***State Machines, States, and Transitions***

In standard UML, state machines can be used to express behavior of parts of a system, too. In OOA UML, however, state machines are behavioral model elements owned exclusively by classes. The class that is the owner of a state machine represents the context of that state machine. A class can own at most one state machine, which describes the behavior of its objects in terms of reactions on event occurrences at run-time. Event occurrences are invocations of the object's operations.

As already described, an operation of a class is just a specification of a service that can be requested from each object of that class. An invocation of an operation of an object affects some behavior and ultimately leads to the execution of a set of actions. So far, we have considered only a situation in which a behavior is attached to an operation as its method, directly specifying the effect caused by an invocation of the operation. The contents of the method can be constructed using different techniques (for example, by coding in a specific detail-level language, by creational object structures, by demonstration, and so on). Anyhow, the method is one complex activity that may consist of many actions.

On the other hand, a non-abstract operation of a class does not need to have an explicitly attached method. Instead, the behavior affected by its invocation can be specified by the state machine attached to the operation's class. In that case, an object of the class will react to the invocation of the operation according to that state machine.

In general, a class with an owned state machine can have non-abstract operations with and without methods. The reaction to an invocation of an operation with an explicitly attached method is the activation of that method, while the reaction to an invocation of an operation without the attached method is determined by the class's state machine.

For example, the class `Order` from the order processing example has several non-abstract operations shown in Figure 14-1 that are referred to from the triggers in the class's state machine shown in Figure 14-2. These operations do not have attached methods. However, the same class can have other non-abstract operations, not shown in Figure 14-1, that do have attached methods. For example, there can be an operation `createShipment` with a method that creates a `Shipment` with the corresponding `Shipment Items`, and links the newly created objects appropriately. Because it is a classical creator, the method can be specified using a detail-level language, a creational object structure, or a demonstration, for example.

## Part III: Concepts

---

In OOIS UML, a non-abstract operation must not have an attached method and appear as a trigger in the class's state machine at the same time. Such a model would be considered ill-formed. Static operations cannot be implemented by state machines. They must have explicit methods.

Consequently, non-static operations of a class represent the behavioral interface of that class. They are specifications of the services offered by the objects of that class. A method attached to such an operation represents the behavior that implements that operation only, while a state machine attached to the class implements the behavior of the entire object and has a broader behavioral scope — it usually implements more than one operation. In other words, a state machine defines interdependencies of the reactions to calls of many operations, because the reaction to a call of one operation may depend on the history of invocations of the same or other operations. Although the same reaction of objects can be achieved by defining methods for each of those operations, state machines do it in a more concise and clear way because the interdependencies of different methods are always implicit and subtle.

For example, consider the class `Order` and its state machine. The behavior of objects of `Order` specified by the state machine in Figure 14-2 can be equivalently implemented by introducing a private attribute `state` of an enumeration type and the following explicit methods.

The method for `Order::place` may look like this:

```
if (state==Created) state=Initiated;
```

The method for `Order::reject` may look like this:

```
if (state==Processing) state=Created;
```

In this approach, it is the developer's responsibility to manually implement the bodies of the methods according to the abstract lifecycle model. It can be easily concluded that for more complex lifecycle models, possibly including conditional transitions that will be described shortly, this approach could be extremely tedious and error-prone. State machines provide a more direct and explicit means for specifying object behavior using highly abstract and descriptive models.

A *state* models a situation during which some usually implicit invariant condition holds, such as an object waiting for some external event to occur. As already described, in OOIS UML, external event occurrences are calls of the object's operations. For example, while resting in the state `Created`, an object of the class `Order` is waiting for a call of one of its operations: `place`, `save`, or `remove`. When one of these call events occurs, the object will react accordingly and move to the target state. Until then, all calls of other operations will be simply ignored (there will be no effect on the object and the calls will immediately complete).

A *transition* is a directed relationship between a source vertex and a target vertex in the state machine as a graph. Basically, a vertex can be a state, but as shown later, it can also be a *pseudostate*, a special kind of vertex in a state machine. A transition specifies the response of the state machine to an occurrence of an event of a particular kind.

A transition can have one or more *triggers*. A trigger specifies an event that fires the transition and causes the state machine to move to the target state. In OOIS UML state machines, events are always *call events*, specifying calls of operations. Triggers are specifications of events within the model and, thus, represent model elements that exist at design time. At run-time, there are *event occurrences* that actually trigger the state change and cause the behavioral effects specified by the state machine. In OOIS UML, these event occurrences are actual invocations of operations, ultimately caused by execution of Call Operation actions within the same or other behaviors in the system.

For example, for the transition from `Created` to `Initiated` in Figure 14-2, the trigger refers to the operation place of the class `Order`, and specifies the call event. At run-time, when this operation is invoked for a certain host object, the transition will be fired and the host object will move to the state `Initiated`, if it was in the state `Created`. The transitions going out from the initial states cannot have triggers.

A state machine is a namespace nested within its owning class. A state is also a namespace nested within its owning state machine. Consequently, both state machines and states have names, and can be referred to by their fully qualified names in a usual manner. For example, the state machine named `Lifecycle` of the class `Order` can be referred to by `OrderProcessing::Order::Lifecycle`, assuming that the class `Order` is placed within the topmost package, `OrderProcessing`, while the state `Processing` of this state machine can be referred to by `OrderProcessing::Order::Lifecycle::Processing`.

In OOIS UML, the state an object is currently residing in is stored in an implicit attribute named `state` that exists by default in every class owning a state machine. This attribute is of type `Text` so that its value can be read by usual Read Attribute Value actions and compared using ordinary textual comparison operations in any detail-level or query language. The value of this attribute is a partially qualified name of the current state, starting from the class's state machine. For example, if an object of `Order` resides in the state `Created`, then its `state` attribute has the value "`Created`". This allows easy and flexible search operations and queries for the objects in a certain state. For example, to search for all initiated Orders, you could write the following OQL query:

```
SELECT ord FROM Order ord WHERE ord.state='Initiated'
```

Similarly, to search for all Orders that are not in the state `Processing`, you can write this:

```
SELECT ord FROM Order ord WHERE ord.state<>'Processing'
```

Assuming that the wildcard `%` stands for any sequence of characters, you could even write this:

```
SELECT ord FROM Order ord WHERE ord.state LIKE '%ed'
```

A state machine owned by a specializing class redefines the state machine owned by the generalizing class, meaning that instances of the specializing class behave according to the state machine of the specializing class and not of the generalizing class. Of course, if the specializing class does not define a state machine, it inherits the one from the generalizing class.

### Section Summary

- A *state machine* is a behavioral model element owned by a class, which defines the reaction of objects of the class to calls of non-abstract operations that do not have explicit methods.
- A *State* models a situation during which some (usually implicit) invariant condition holds, such as an object waiting for some external call event to occur.
- A *Transition* is a directed relationship between a source vertex and a target vertex in the state machine as a graph. A vertex can be a state or a pseudostate.

*Continued*

- A *Trigger* specifies a call event that fires the transition and causes the state machine to move to the target state.
- The name of the state an object is currently residing in is stored in an implicit attribute named `state` of type `Text` that exists by default in every class owning a state machine.

### Guards and Effects

The Order lifecycle model considered so far is quite simple and probably insufficient for many real applications. The discussions that follow enhance the lifecycle model by introducing new concepts related to state machines in order to meet extended requirements.

Let's assume that the simple model considered so far must be extended to support the following requirements:

1. When a created Order is placed by the Customer, the system should first check its completeness according to some business rules that are out of the scope of this discussion. For example, the system may check whether the Customer has properly filled in the shipping address, whether a required minimum number of Products is ordered, and so on. Only if the Order is complete does it become initiated. Otherwise, a comment should be given, and the Order should remain in the state `Created`.
2. When a Sales Agent takes over an initiated Order, he or she should become "in charge" of that Order.
3. When a completed Order gets "removed," it should be archived to another backup storage. (It still will exist in the object space, although it will not respond to events anymore.)

To support the given requirements, several extensions have been made in the class `Order`, as shown in Figure 14-3a:

- There is a new helper operation, `isComplete`, that checks the completion of a created Order and returns `true` if the Order is complete. This is a non-abstract operation with a method that does the actual checking according to the business rules.
- Another operation, `setIncompleteComment`, sets the textual attribute `comment` of an Order to the value that gives the explanation for rejecting the Order if it is incomplete. This is a non-abstract operation with a method that is out of the scope of this example.
- The operation `takeOver` accepts an argument of type `SalesAgent` that refers to the agent that takes it over and becomes in charge of it. This is still a non-abstract operation without a method, whose implementation is defined by the state machine.
- There is a new helper operation, `archive`, that performs archiving of an Order to backup storage. This is a non-abstract operation with a method that is out of the scope of this example.

To fulfill requirement 1 given previously, a conditional transition from the state `Created` to the state `Initiated` is obviously needed. Only if `isComplete` returns `true`, should the Order move to `Initiated`; otherwise, it should stay in the same state.

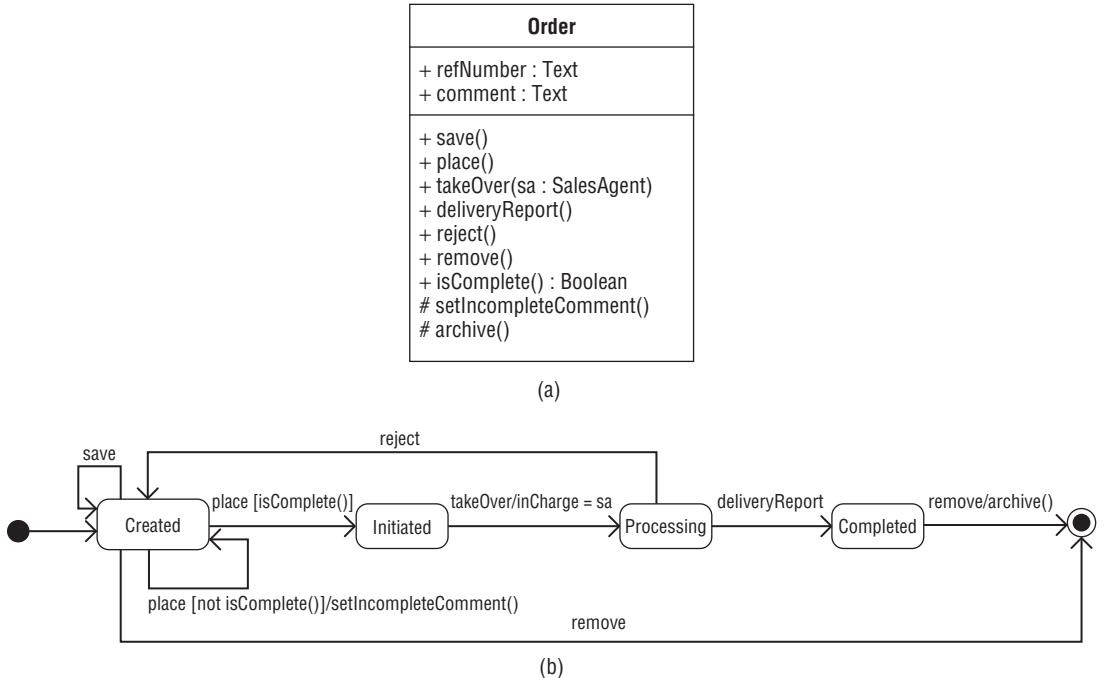


Figure 14-3: (a) The class `Order` with a few more helper operations with methods. (b) State diagram of the lifecycle of an `Order` with guards and effects.

For this purpose, *guards* can be used, as shown in Figure 14-3b. A *guard* is an optional condition attached to a transition that controls the firing of the transition. Only if the guard is satisfied can the transition be fired. In Figure 14-3b, the guard `isComplete` associated with the transition from `Created` to `Initiated` specifies the expression that consists of the Boolean result of the invocation of the operation `isComplete` of the object hosting the state machine and responding to the dispatched call event `place`.

In general, a guard is a constraint attached to a transition. As with any other constraint in the model, it can be specified in OCL or in any other supported language. It is evaluated in the context of the class owning the state machine and, thus, can refer to its members directly because they are in scope. Moreover, a guard is evaluated in the scope of the operation specified by the trigger, and can also refer to the formal parameters of that operation. As with any other constraint, a guard must not have side effects and must return a Boolean value. The transitions going out from the initial states cannot have guards.

The expression of the guard can be shown in the diagram between square brackets `[]` following the trigger, as shown in Figure 14-3b.

To fulfill the rest of the requirements, some actions must be performed when a transition is taken. This can be specified by *effects* of transitions. An *effect* is an optional behavior associated with a transition that is activated when the transition is fired. In Figure 14-3b:

- ❑ When a created `Order` is incomplete, its `comment` attribute is set by calling `setIncompleteComment` as an effect of the guarded transition from `Created` to the same state.

## Part III: Concepts

---

- ❑ When a Sales Agent referred to by the parameter `sa` of the called operation `takeOver` takes over an initiated Order, the property `inCharge` of the hosting Order is set to `sa` by the effect associated with the transition from `Initiated` to `Processing`.
- ❑ When a completed Order is being removed, it is archived by calling the operation `archive`, which is specified as the effect of the transition from `Completed` to the final state.

An effect is a behavior attached to a transition. As with any other behavior in the model, it can be specified in many different ways, but is usually specified in a detail-level language. It is evaluated in the context of the class owning the state machine and, thus, can refer to its members directly because they are in scope. Moreover, in OOA UML, an effect is evaluated in the scope of the operation specified by the trigger, and can also refer to the formal parameters of that operation.

Although an effect can have an arbitrary number of actions, it usually has only one — an action that simply calls a helper operation of the same host object. The method of that operation then does the whole job. Although it is also possible to specify multiple actions directly in the effect of a transition, it is a better practice to encapsulate them into a method of such a helper operation, usually a protected one, as was done in the example in Figure 14-3. This ensures better localization, encapsulation, readability, and reuse of behavior. In some special cases when the effect is quite simple and consists of only one action (such as the action of setting the property `inCharge` to the value of the parameter `sa` in Figure 14-3b), the action can be directly specified as the effect.

The effect written in the target detail-level language can be shown in the diagram following a slash /, as shown in Figure 14-3b. In general, the syntax of the textual notation placed near the transition arrow is as follows:

```
transition ::= trigger-spec guard-specopt effect-specopt
trigger-spec ::= trigger
trigger-spec ::= trigger , trigger-spec
guard-spec ::= [expression]
effect-spec ::= / expression
```

### Section Summary

- ❑ A *guard* is an optional constraint attached to a transition that is evaluated when an event occurrence is dispatched by the state machine. If the guard is true at that time, the transition may be fired; otherwise, it is disabled.
- ❑ An *effect* is an optional behavior attached to a transition that is activated when the transition is taken.
- ❑ Both guard and effect are evaluated in the context of the operation specified by the trigger, and can refer to the formal parameters of the operation, as well as to all elements accessible from the hosting object.
- ❑ The notation is *trigger*[*guard*] /*effect*.

## Semantics

The run-time semantics of state machines in OOIS UML are somewhat simplified, but based on the definition in standard UML. In OOIS UML, state machines have derived semantics as described in this section.

Every class owning or inheriting a state machine has one implicit attribute named `state` of type `Text` and multiplicity 1. The visibility of this attribute is the same as the visibility of the state machine, being a named element owned by the class. The value of this attribute stores the name of the current state of the host object. The value of this attribute should not be modified explicitly by user-defined action code. It is modified by the run-time environment only when a transition is performed. Once the transition is completed and the target state is reached, this attribute value gets frozen. Therefore, the value of this attribute cannot be modified explicitly by the user-defined code unless it is previously unfrozen. However, although possible, such explicit modifications are deprecated.

This attribute gets the initial value that is the name of the target state of the single transition going from the initial pseudostate. The optional effect of that transition is performed after the constructor of the object is completed.

The implementation of each non-abstract operation without an explicit method is defined by the state machine and is equivalent to an implicit method of that operation, which does the following:

- ❑ If the object is in the state that does not have an outgoing transition that has a trigger for this operation, nothing happens and the execution of the operation call is completed.
- ❑ Otherwise, the guards attached to the transitions going out from the current state having the trigger for this operation are evaluated in an arbitrary order. Those transitions whose guards evaluate to `true` or have no guards are considered *enabled*; the others are disabled and will not be taken.
- ❑ If there are no enabled transitions, nothing happens and the execution of the operation call is completed.
- ❑ From the set of enabled transitions, exactly one is arbitrarily selected to be fired, while the others are ignored. This means that the implementation is free to rely on any algorithm for selecting one of the enabled transitions to fire. The algorithm is unknown to the modeler and the model should not rely on that algorithm. The modeler should avoid this kind of ambiguity in models. In other words, the modeler is responsible for defining the guards of the transitions going out from a state so that they have complementary values of which no more than one ever equals to `true`. Similarly, although allowed, it is not a good practice to define more than one transition going out from the same state with the same trigger, but without guards.
- ❑ The selected enabled transition is fired, which means that the following steps are performed one after the other, in this order:
  1. The effect behavior is executed.
  2. When it completes entirely, the current state of the object is changed to the target state of the transition by writing a new value to the `state` attribute.

The following code shows the implementation of several implicit methods of the class `Order` that have equivalent semantics as the one defined by the state machine in Figure 14-3b.

## Part III: Concepts

---

The method for `Order::place()` looks like this:

```
if (state.isEqual("Created")) {
 if (isComplete()) {
 state->unfreeze();
 state="Initiated";
 state->freeze();
 return;
 }
 if (!isComplete()) {
 setIncompleteComment();
 state->unfreeze();
 state="Created";
 state->freeze();
 return;
 }
}
```

The method for `Order::takeOver(sa:SalesAgent)` looks like this:

```
if (state.isEqual("Initiated")) {
 inCharge = sa;
 state->unfreeze();
 state="Processing";
 state->freeze();
 return;
}
```

The method for `Order::remove()` looks like this:

```
if (state.isEqual("Created")) {
 state->unfreeze();
 state="final";
 state->freeze();
 return;
}
if (state.isEqual("Completed")) {
 archive();
 state->unfreeze();
 state="final";
 state->freeze();
 return;
}
```

### Section Summary

- ❑ Every class owning or inheriting a state machine has one implicit attribute named `state` of type `Text` and multiplicity 1, whose value stores the name of the current state of the host object.

- ❑ The implementation of each non-abstract operation without an explicit method is defined by the state machine and is equivalent to an implicit method of that operation that does the following:
  - ❑ The guards attached to the transitions going out from the current state having the trigger for this operation are evaluated in an arbitrary order. Those transitions whose guards evaluate to `true` or have no guards are considered *enabled*. The others are disabled and will not be taken.
  - ❑ From the set of enabled transitions, exactly one is arbitrarily selected to be fired, and the others are ignored.
  - ❑ The selected enabled transition is fired. The effect behavior is executed, and when it completes entirely, the current state of the object is changed to the target state of the transition by writing a new value to the `state` attribute.

## Advanced Concepts

Standard UML supports many other advanced concepts that allow modeling more complex behaviors of state machines. The concepts supported by OOIS UML include composite states, history, entry and exit behaviors, entry and exit points, and submachines.

### Composite States and History

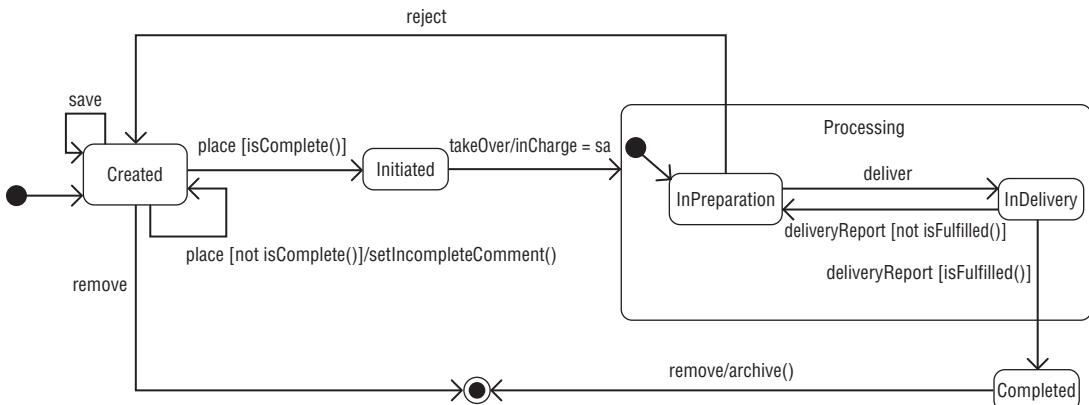
The simplified Order lifecycle model considered so far completely neglects the potentially complex and iterative process of preparation of one or more Shipments for a placed Order, the actual delivery of the products, and reception of a delivery report in the shipping department. It simply abstracts the entire complex process going on in the state `Processing`.

Now, let's refine the lifecycle model so that the system can follow a process like this:

1. The Sales Agent processes the Order by manually preparing a Shipment. The Sales Agent creates a Shipment and Shipment Items for it, according to the current stock of the ordered Products and other business rules.
2. Once the Shipment is created, it is passed to the shipping department and the Order is declared to be *in delivery*. The shipping department delivers the products to the customer and brings back a delivery report. The delivery report is entered into the system manually because it comes in as a paper form signed by the customer.
3. The input of a delivery report is recorded as an event for the Order; if the Shipment fulfilled the Order (for example, by delivering all the ordered Products), the Order then becomes *completed*. Otherwise, the Order goes back to preparation of another Shipment, and so on.

## Part III: Concepts

As you can see, you must *refine* the state Processing of the Order lifecycle model by decomposing that state into substates InPreparation and InDelivery, as shown in Figure 14-4. While an Order is in the state InPreparation, the Sales Agent who is in charge of its processing might reject it if it does not satisfy some business rules. Otherwise, the Sales Agent creates the Shipment for that Order and ultimately moves the Order to the state InDelivery. When the delivery report is entered into the system, if the Order is completely fulfilled (a helper query operation isFulfilled returns true), it is moved to the state Completed. Otherwise, the Order is moved back to InPreparation so that it can be fulfilled in one or more iterations.



**Figure 14-4: State diagram of the lifecycle of an Order with the composite state Processing refined into substates**

The transition from the **Initiated** state, triggered by **takeOver**, targets the **Processing** state, which is a composite state. When fired, this transition actually results in moving the object into the initial pseudostate owned by this target state, and the transition going out from that pseudostate is then fired. This is the transition to the state **InPreparation**, in this case. In general, when a transition targets a composite state, the execution of that transition moves the state machine to the initial pseudostate of the composite state and fires the initial transition going out of that pseudostate.

Consequently, when the state machine is at rest, it is in one of the primitive states not having its substates. That current primitive state, as well as all its direct or indirect owner composite states, are said to be *active* while the state machine resides in the current primitive state. For example, while residing in the **InPreparation** state, active states are **Processing** and **InPreparation**.

Composite states are namespaces and their substates are nested namespaces. All usual naming rules apply here, too. For example, within the scope of the state machine, the **InPreparation** state can be referred to as **Processing::InPreparation**. This naming can be used when querying for objects residing in a state. For example, the following OQL query returns all Orders residing in the **InPreparation** state:

```
SELECT ord FROM Order ord
WHERE ord.state='Order::Processing::InPreparation'
```

Similarly, to search for all Orders that are in any of the substates of **Processing**, you could write the following:

```
SELECT ord FROM Order ord
WHERE ord.state LIKE 'Order::Processing::%'
```

If a state machine has several states that are substates of different composite states, but have the same unqualified name, you could even write the following:

```
SELECT obj FROM SomeClass obj WHERE obj.state LIKE '%::SomeSubstate'
```

Let's now further enhance the lifecycle model of an Order with the following features:

- ❑ As either `Initiated` or `Processing`, an Order can be canceled. Once canceled, the Order can only be removed, with mandatory archiving of it to the backup storage.
- ❑ As either `Initiated` or `Processing`, an Order can be suspended. Once suspended, the Order can either be removed (with mandatory archiving of it to the backup storage), or resumed to exactly the same state it was before its suspension.

Apart from introducing new primitive states `Canceled` and `Suspended`, which are directly implied from these requirements, it is a good idea to introduce some more composite states that will unify some existing states and gather their common reaction to some triggers, as shown in Figure 14-5:

- ❑ Instead of drawing a transition from `Initiated` and every one of the substates of `Processing` to `Canceled` and `Suspended`, a new composite state, `Active`, contains both `Initiated` and `Processing` and defines such a transition.
- ❑ A new state, `Passive`, contains the states `Completed`, `Suspended`, and `Canceled` and defines their common reaction to the `remove` event.

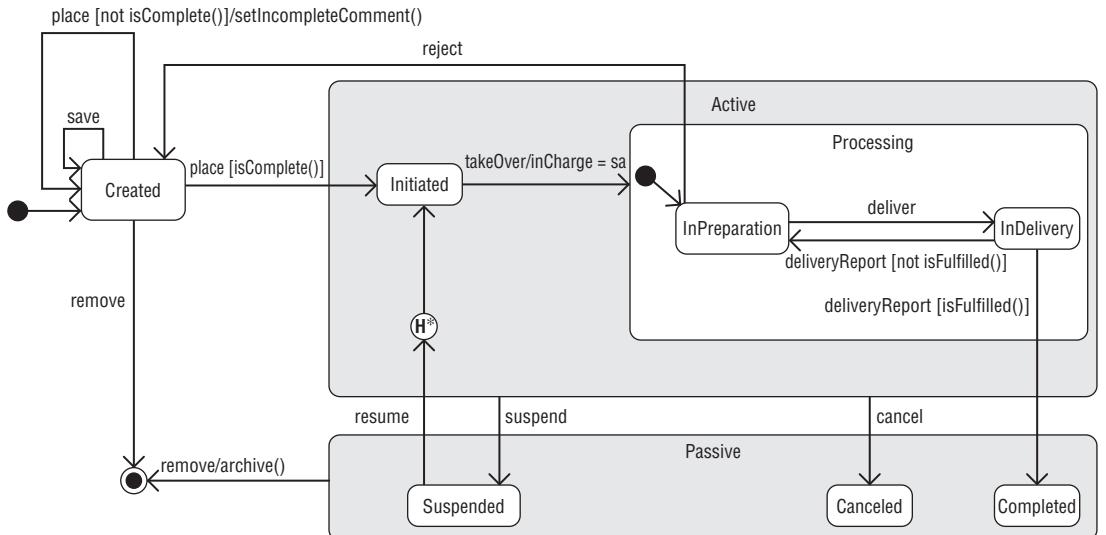


Figure 14-5: State diagram of the lifecycle of an Order with a composite state `Active` having a deep history pseudostate

## Part III: Concepts

---

Such unification of the states that have something in common (for example, represent conceptually similar things, or share some reactions) is often useful because it may simplify queries and classification of objects according to their states. For example, it is now easy to search for all Orders that are in the Initiated or Processing state because you can simply search for the objects residing in the Active state. Notice that composite states are useful essentially as abstraction facilities from which the elimination of repeated transitions is almost invariably a natural side effect.

In general, if the current primitive state does not react to a dispatched event (because there is no transition going out from that state with the corresponding trigger), a transition going out from its owner composite state reacting to that event is searched for. If that state does not have such a transition, its owner composite state is searched for such an outgoing transition, and so on, recursively.

The first such transition encountered in this search fires. If there is no such transition, the event is simply ignored and the state machine does not react. For the state machine in Figure 14-5, if the current state is Active::Processing::InDelivery and the event suspend is dispatched, the state machine moves to the state Suspended because the transition going out from the indirectly enclosing state Active is fired. In this way, a transition going out from a composite state defines a common reaction to an event for all its substates, eliminating redundant specifications of transitions, whereas any of the substates can still override that general reaction specification by defining its own outgoing transition with the same trigger.

Another question is how to return a suspended Order to exactly the same state in which it had been when it was suspended. The concept of a *history pseudostate* provides a solution. If a transition terminates on a shallow history pseudostate, denoted with a circled H symbol, directly owned by a composite state S, the active substate becomes the most recently active substate of the composite state S, unless the most recently active substate is the final state or if this is the first entry into S. In the latter two cases, the *default history state* is entered. This is the substate of S that is the target of the transition originating from the history pseudostate. If no such transition is specified, the model is ill-formed and the behavior is undefined.<sup>1</sup> If the active substate determined by history is a composite state, then it proceeds with its default entry (transition going out from its initial pseudostate).

In this example, an Order has to be returned to the complete set of composite states and the ultimate primitive state that were most recently active. This is handled by the *deep history pseudostate*, denoted with a circled H\* symbol, as shown in Figure 14-5. The rule here is the same as for shallow history, except that the process of entering the most recently active substate is applied recursively to all levels in the composite state hierarchy below this one.

For the example in Figure 14-5, if the Order is suspended from the state Active::Processing::InPreparation, the transition from Passive::Suspended triggered by resume moves the machine back to the state Active::Processing::InPreparation. Similarly, if the Order is suspended from the state Active::Processing::InDelivery, it is resumed to that state. If the transition from Passive::Suspended triggered by resume is fired but the machine has never been in the state Active before (which, in this example, can never happen because of the specific construction of the state machine — it can never come to the state Passive::Suspended but from the state Active), the machine is move to the state targeted by the transition going out from the history pseudostate H\* (that is, to the state Active::Initiated).

---

<sup>1</sup>A history state does not have to have a transition to a default history state defined. This is only required if there is a possibility that a transition to history is executed before any history is recorded.

If a shallow history pseudostate ( $H$ ) were used instead of the deep history pseudostate ( $H^*$ ), an Order suspended from `Active::Processing::InDelivery` would resume to the state `Active::Processing::InPreparation` because the history would reactivate only the first-level substate (`Processing`, in this case) of the composite state owning the shallow history pseudostate, while its initial substate (and its initial substates recursively) would become active (`InPreparation`, in this case).

### Section Summary

- ❑ A state can be *composite* (owning nested substates) or *primitive* (having no substates).
- ❑ Composite states are namespaces and their owned substates are nested namespaces.
- ❑ When the state machine is at rest, it resides in exactly one primitive state. That current primitive state, as well as all its direct or indirect owner composite states, are said to be *active*.
- ❑ If the active primitive state does not react to a dispatched event (because there is no transition going out from that state with the corresponding trigger), a transition going out from its owner composite state reacting to that event is searched for. If that state does not have such a transition, its enclosing owner composite state is searched for such an outgoing transition, and so on, recursively. The first such transition fires. If there is no such transition, the event is simply ignored and the state machine does not react.
- ❑ If the fired transition terminates on a composite state, the active substate becomes the initial pseudostate of that composite state, and the initial transition going out from that pseudostate is fired.
- ❑ If the fired transition terminates on a *shallow history* pseudostate (denoted with a circled  $H$ ), the active substate becomes the most recently active substate of the composite state owning the history pseudostate, unless the most recently active substate is the final state or if this is the first entry into this state. In the latter two cases, the *default history state* is entered. This is the substate that is the target of the transition originating from the history pseudostate. If the active substate determined by history is a composite state, then it proceeds with its default entry.
- ❑ The rule for a transition terminating on a *deep history* pseudostate (a circled  $H^*$ ) is the same as for shallow history, except that the rule of entering most recently active substates is applied recursively to all levels in the state hierarchy below this one.

### Pseudostates and Final State

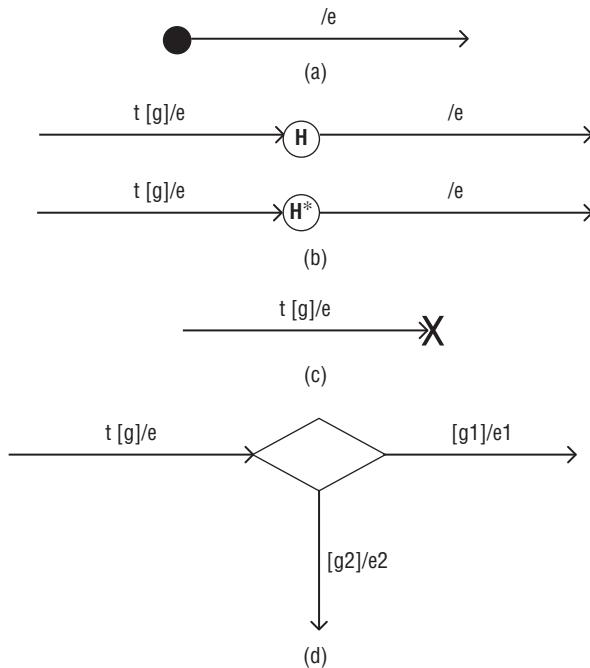
A state machine is a graph whose vertices are states and pseudostates, and whose edges are transitions that connect vertices. Pseudostates are not true states because the state machine never rests in a pseudostate during non-transitional intervals. Instead, a pseudostate is just a transient vertex that affects

## Part III: Concepts

---

the sequence of guards evaluated, and actions taken, for a traversal from the source to the ultimate target state. They are used to connect multiple transitions into more complex state transitions paths. The transitions going out from pseudostates may not have a trigger.

Figure 14-6 summarizes all kinds of pseudostates supported by OOIS UML and shows their symbols.



**Figure 14-6: Notation for pseudostates.** The elements attached to transitions indicate allowed kinds of elements for the transitions going into or out from the pseudostates. (a) Initial pseudostate. (b) Shallow and deep history pseudostates. (c) Terminate pseudostate. (d) Choice pseudostate.

An *initial* pseudostate (see Figure 14-6a) has a single outgoing transition to the *default* substate of a composite state. When a transition that targets a composite state is fired, the outgoing transition from the sole initial pseudostate owned by that composite state is then fired. There can be, at most, one initial pseudostate in a composite state. An initial pseudostate can have, at most, one outgoing transition. That outgoing transition may have an effect, but not a trigger or a guard.

Reaching a *shallow history* pseudostate (see Figure 14-6b) represents entering the most recently active substate of its containing composite state and the default substate of that substate. A composite state can have, at most, one shallow history pseudostate. At most, one transition may originate from the history pseudostate to the *default* shallow history substate. This transition is taken in case the owner composite state was never active before, or its most recently active substate was the final state. Entry behaviors (described later in this chapter) of states entered on the path to the state represented by a shallow history are performed.

Reaching a *deep history* pseudostate (see Figure 14-6b), on the other hand, means entering the most recently active substate of its containing composite state, and the most recently active substate of that substate, and so on, recursively. The rest of the rules are the same as for the shallow history pseudostate.

Although there can be, at most, one initial, deep, and shallow history pseudostate (as a model element) in a composite state, it may appear more than once in a diagram for graphical convenience, as is usual in UML.

Reaching a *terminate* pseudostate, denoted with an X symbol (see Figure 14-6c), is equivalent to invoking a Destroy Object action. The state machine is terminated and its context object is destroyed. No exit actions (described later in this chapter) other than those associated with the transition leading to the terminate pseudostate are executed.

A *choice* vertex (see Figure 14-6d) is a kind of usual conditional branch box in the classical flowcharts. When a choice vertex is reached, the guards of its outgoing transitions are dynamically evaluated. These guards may use the results of prior actions performed in the same transition process.

For example, the guards  $g_1$  and  $g_2$  in Figure 14-6d may depend on the results and side effects of the behavior  $e$  of the incoming transition. If only one of the guards evaluates to `true`, its outgoing transition is taken. If more than one of the guards evaluates to `true`, an arbitrary one is selected. If none of the guards evaluates to `true`, then the model is considered ill-formed and the behavior is undefined (that is, it is implementation-dependent). To avoid this, it is recommended that one outgoing transition with the predefined `else` guard be defined for every choice vertex. This guard evaluates to `true` if none of the other guards evaluates to `true`. In a complete state machine, a choice vertex must have at least one incoming and one outgoing transition.

A choice or a history pseudostate connects its incoming and outgoing transitions into a *compound transition*, a notion that is referred to in the descriptions of the transition execution semantics given later. A compound transition represents a complete path made of one or more transitions, originating from one true state (as opposed to a pseudostate), joined via one or more pseudostates, and targeting a true state (as opposed to a pseudostate). As a special case, the originating and the target state of a compound transition can be the same state for self-transitions. In addition, a compound transition can be composed of one simple transition connecting two states, without pseudostates.

As opposed to the described pseudostates, the *final state* is a special kind of a true state, signifying, when reached, that its enclosing composite state or state machine is completed. If the enclosing owner is a state machine, then it means that the entire state machine (as a behavior) is completed. Otherwise, if the enclosing owner is a composite state, when the final state is reached, a *completion event* is dispatched, signifying completion of that composite state. Such an event fires a *completion transition* going out from that composite state. This is a transition that does not have an explicit trigger, although it may have a guard.

A completion transition (that is, a transition exiting a state without explicit triggers) is, thus, implicitly triggered by a completion event (that is, when a composite state enters its final substate). In case of a primitive state, a completion event is dispatched immediately upon entering the state. The completion event is handled before any other events, and has no associated parameters. If multiple completion transitions are defined for a state, then they should have mutually exclusive guard conditions. Otherwise, the model is ill-formed and the result is undefined.

In OOIS UML, the final state is as any other state and can be referred to by its name `final`. A state machine residing in its final state does not react on any event. The object whose state machine has reached

## Part III: Concepts

---

its final state is not implicitly destroyed, as is the case with the terminate pseudostate. For the order processing example and the shown state machine, if the final state were replaced with a terminate state, a “removed” Order would be implicitly destroyed (physically removed from the system’s object space). With a final state, the object still survives, although it does not respond to event occurrences and rests in that state.

### Section Summary

- ❑ A pseudostate is not a true state in which the state machine rests between operation invocations, but is just a transient vertex that affects the sequence of guards evaluated and actions taken at a transition path from the source to the ultimate target state.
- ❑ An *initial* pseudostate has a single outgoing transition to the *default* substate of a composite state. When a transition that targets a composite state is fired, the outgoing transition from the sole initial pseudostate owned by that composite state is then fired.
- ❑ Reaching a *shallow history* pseudostate represents entering the most recently active substate of its containing composite state and the default substate of that substate.
- ❑ Reaching a *deep history* pseudostate represents entering the most recently active substate of its containing composite state, and the most recently active substate of that substate, and so on, recursively.
- ❑ Reaching a *terminate* pseudostate is equivalent to the Destroy Object action for the owner object.
- ❑ A *choice* pseudostate is a kind of usual conditional branch box in the classical flowcharts. A choice vertex, when reached, results in the dynamic evaluation of the guards of its outgoing transitions and the selection of one whose guard evaluates to *true*.
- ❑ A *compound transition* represents a complete path made of one or more transitions, originating from one true state, joined via one or more pseudostates, and targeting a true state.
- ❑ A *final state* is a special kind of a true state, signifying, when reached, that its enclosing composite state or state machine is completed.

## Entry and Exit Behaviors

Sometimes some actions have to be performed whenever a state is entered or exited. For example, in the order processing system, you want to freeze some or all attributes of an Order and its associated Order Items, Shipments, and Shipment Items whenever the Order enters the `Passive` state. This is to ensure that the Order is “hibernated” whenever it is suspended, canceled, or completed, so that it (or some of its parts) cannot be modified by users. Similarly, when the Order exits the `Passive` state and resumes, it should be unfrozen.

Let's assume that helper operations `freeze` and `unfreeze` of the class `Order` provide the necessary actions for this. It follows that the `freeze` operation should be called whenever the `Passive` state is entered, and the `unfreeze` operation should be called whenever that state is exited.

Such a request can be fulfilled using the concepts described so far: the effects of all transitions entering the `Passive` state should be defined to include the call of `freeze`, while all transitions exiting that state should be defined to include the call of `unfreeze`. However, this approach would have several disadvantages:

- ❑ The (possibly different) behaviors of transitions must be artificially extended to include these common actions.
- ❑ A conceptually localized aspect (that something must be done whenever a state is entered or exited) is spread across many loosely coupled transitions, which leads to redundancy, and implies poor readability of the model and bad localization of design decisions.
- ❑ Incorporating such needs for nested states may be more complicated for the transitions entering or exiting composite enclosing states.

Instead of this, such behaviors can be optionally associated with the very state in the form of *entry* and *exit behaviors*, as shown in Figure 14-7. An *entry behavior* is an optional behavior that is executed whenever the state is entered, regardless of the transition taken to reach the state. Similarly, an *exit behavior* is an optional behavior that is executed whenever the state is exited, regardless of the transition taken to exit the state. For the example in Figure 14-7, the Order will be frozen whenever it is suspended, completed, or canceled, and unfrozen whenever it is resumed or removed.

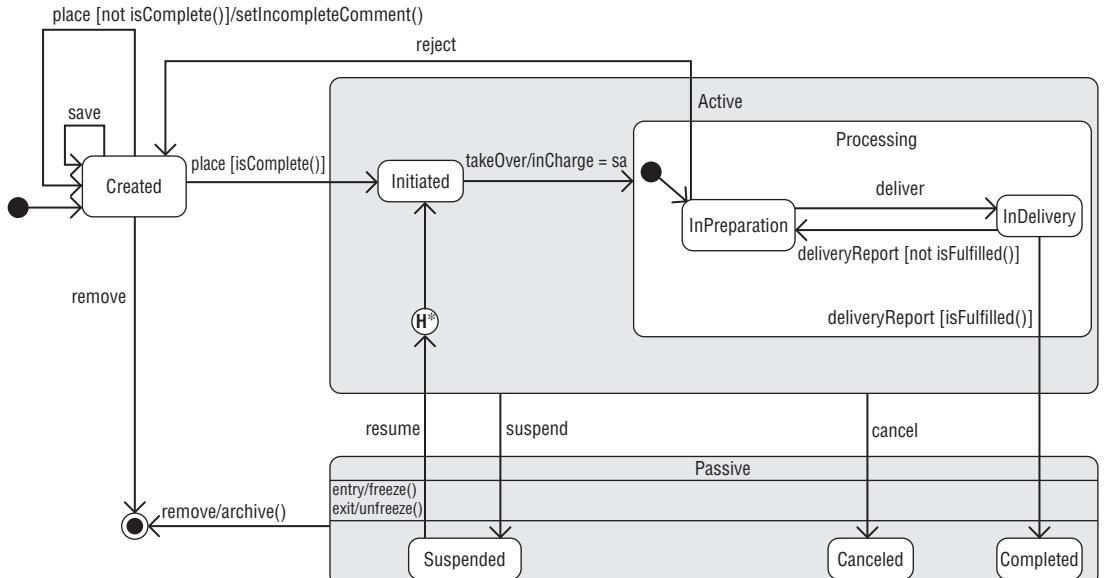


Figure 14-7: Entry and exit behaviors

Entry and exit behaviors are executed whenever the state is entered and exited by any transition, including self-transitions. However, a transition can be *internal*. An internal transition is a transition from one to the same state that executes without exiting or re-entering the state in which it is defined. This is true

## Part III: Concepts

---

even if the state machine is in a nested state within this state. In diagrams, internal transitions are not drawn as lines, but are rather listed in a separate compartment of the state, similar to the entry and exit behaviors, using the textual notation for transitions (*trigger[guard]/effect*).

A transition originating from a composite state is called a *high-level* or *group* transition. If triggered, a high-level transition results in exiting of all the substates of its originating composite state and executing their exit behaviors starting with the innermost active state outward. Such a transition with a target outside its source composite state will include the execution of the exit action of the composite state, too, whereas one with a target inside the composite state will not.

In general, as also shown in Figure 14-7, a state symbol can have multiple compartments:

- A name compartment (showing the state's name)
- An internal activities compartment (listing the entry and exit behaviors)
- An internal transitions compartment (listing the internal transitions)
- A decomposition compartment for composite states (showing their substates)

Optionally, the state symbol may have an attached name tab, which is a rectangle placed on top of a state symbol and containing the name of that state.

### Section Summary

- An *entry behavior* is an optional behavior that is executed whenever the state is entered, regardless of the transition taken to reach the state.
- An *exit behavior* is an optional behavior that is executed whenever the state is exited, regardless of the transition taken to exit the state.
- An *internal transition* is a transition from one to the same state that executes without exiting or re-entering the state in which it is defined.
- A transition originating from a composite state is called a *high-level* or *group* transition. If triggered, a high-level transition results in exiting of all the substates of its originating composite state, and executing their exit behaviors starting with the innermost active state outward.

## Semantics

This section summarizes the semantics of state machines as behaviors of classes in OOIS UML. The semantics follow the standard UML specification [UML2], with some simplifications applied in OOIS UML.

During execution, a state of a state machine becomes *active* when it is entered as a result of some transition, and becomes *inactive* when it is exited as a result of a transition. In a hierarchical state machine with composite states, more than one state can be active at the same time. If a simple state is active, then all

the composite states that either directly or indirectly contain the active simple state are also active. Such a set of currently active states is called the *active state configuration*. Except during transition execution, if a composite state is active, at most one of its substates is active.

While there are no incoming call event occurrences, the state machine rests in the active state configuration. When there are one or more incoming call event occurrences, the following processing is performed, in order:

1. One of the incoming call event occurrences is dispatched and processed. Multiple waiting call event occurrences are possible because of multiple concurrent threads invoking operations on the same object while it is busy handling an existing invocation.
2. Among the *enabled transitions* for the active state configuration, one is selected for firing. If there are no enabled transitions for the active state configuration, the event is discarded (that is, consumed without any effects).
3. The states in the active state configuration that the fired transition leaves are exited.
4. The effects and dynamic guards along the transition path are executed.
5. The states in the target active state configuration of the fired transition are entered.
6. The active state configuration is updated.

All these steps will be discussed in greater detail in the discussions that follow. The sample state machine shown in Figure 14-8 will be used to illustrate some parts of the described semantics.

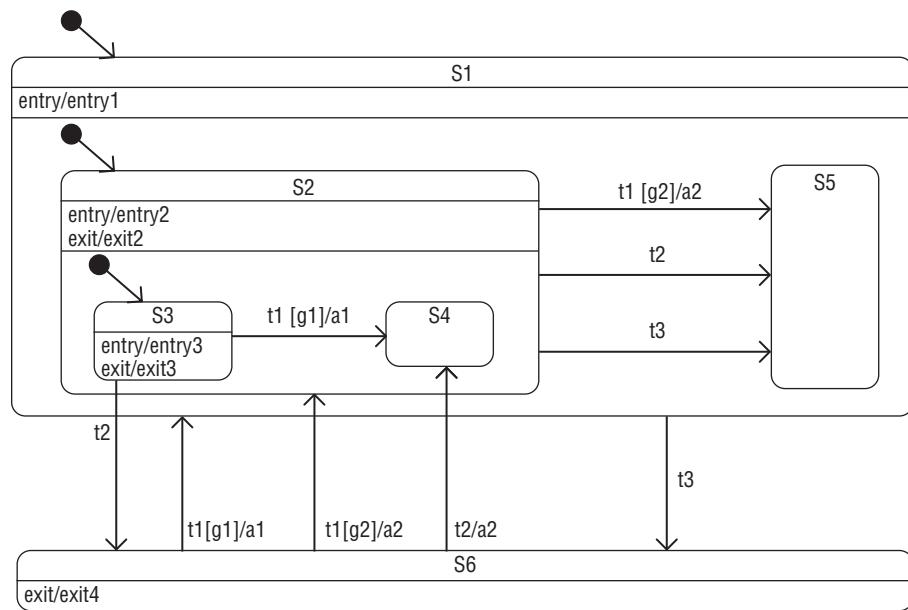


Figure 14-8: Sample state machine

### ***Dispatching Events and the Run-to-Completion Step***

Call event occurrences are dispatched and processed by the state machine one at a time, in an arbitrary (implementation-dependent) order. The dispatched event occurrence is processed fully and completely, until the state machine reaches its target active state configuration, before the next event occurrence is dispatched and processed. The next event occurrence can be dispatched only if the processing of the previous occurrence is fully completed. In other words, processing of one event occurrence cannot interrupt the ongoing processing of another event occurrence, but must wait until the latter is fully completed. Multiple waiting call event occurrences are possible because of multiple concurrent threads invoking operations on the same object while it is busy handling an existing invocation. This is called the *run-to-completion* semantics, whereas the processing of a single event occurrence is referred to as a *run-to-completion step*.

The run-to-completion step represents the transient period during execution of a state machine (that is, the passage between two stable active state configurations). Before starting and after completing a run-to-completion step, a state machine is in a stable state with all activities performed during the transition period completed. During the run-to-completion step, the state machine and its owner object can be in an intermediate and possibly inconsistent condition. Therefore, an event occurrence will never be processed while the state machine is in such a condition. Such semantics allows the state machine to safely complete its run-to-completion step, avoiding concurrency conflicts internal to the implementation of the state machine. In other words, the modeler does not have to be concerned about race conditions possibly caused by concurrent executions of the activities on different transitions of the same state machine because such transitions are always executed sequentially.

However, it should be noted that run-to-completion does not protect from race conditions that occur because of simultaneous concurrent access to shared data (such as the attributes of the host object) by two or more state machines, or calls of the host object's operations with explicit methods running in separate threads. The calls of the host object's operations with explicit methods are not treated as event occurrences for the state machine. Run-to-completion protects only the internal state of the host object (that is, the host object's state attribute), as well as the entire object space, from race conditions caused by concurrent handling of different event occurrences of its state machine.

When an event occurrence is dispatched, none, one, or more transitions can be enabled for firing. If no transition is enabled, the event occurrence is discarded and the run-to-completion step is completed. If more than one transition is enabled, exactly one is selected for execution. When the machine has finished executing the selected enabled transition, the current event occurrence is fully consumed, and the run-to-completion step is completed.

During the execution of a transition selected for the dispatched event occurrence, a number of actions may be executed. If such an action is a synchronous operation call on another object also having a state machine, then the transition step of the caller is not completed until the invoked object's state machine completes its run-to-completion step fired by that call event occurrence. In other words, the caller's run-to-completion step blocks until the synchronous operation call regains control from the method activated by the call. If the method is implied from the state machine of the invoked object, it returns control when it completes its run-to-completion step.

In OOIS UML, run-to-completion is implemented by isolating the processing of an event occurrence — that is, by exclusively locking the implicit `state` attribute of the host object at the beginning of the run-to-completion step. Consequently, a thread calling an operation of the host object

cannot proceed (that is, is blocked) until the previous one completes the run-to-completion step and unlocks the state attribute. In addition, the entire run-to-completion step is an isolated group of actions.

## Transition Selection

When an event occurrence is dispatched, some transitions can be enabled for firing. A transition is *enabled* if and only if all of the following are true:

- Its source state is in the active state configuration.
- One of the triggers of the transition is satisfied by the event of the current occurrence, meaning that the event of the occurrence is the one specified by the trigger.
- Its guard condition is `true`. Transitions without guards are treated as if their guards are always `true`.

For a simple transition with a guard, the guard is evaluated before the transition is triggered. For a compound transition involving a chain of multiple simple transitions and their guards (connected via choice pseudostates), only the guard of the first simple transition that targets the first choice point in the compound transition path is evaluated. Guards downstream of any choice point along the path are evaluated if and when the choice point is reached. The order of evaluation of the guards of the candidate transitions going out from the same state or choice pseudostate is undefined. This is one of the reasons why guards should not have side effects. Models that violate this rule are considered ill-formed, although they might be acceptable (that is, the violations ignored) by the model compiler.

As already noted, it is possible that more than one transition is enabled within a state machine by the same event occurrence. If that happens, then such transitions may be in *conflict* with each other. For example, consider the case of two transitions originating from the same state, triggered by the same event, but with different guards, like the transitions  $t1[g1]/a1$  and  $t1[g2]/a2$  having  $S6$  as their source state in Figure 14-8. If that event occurs and both guard conditions are `true`, the transitions are conflicting, but only one will fire. In general, in case of conflicting transitions, only one of them will fire in a single run-to-completion step — being enabled is a necessary (but not sufficient) condition for the firing of a transition.

Two enabled transitions may also conflict with each other even though they do not have the same state as their common source, but they both exit the same state directly or indirectly. For example, the transitions triggered by  $t2$  going out from the states  $S2$  and  $S3$  in Figure 14-8 are in conflict when the machine resides in  $S3$ , because both transitions are enabled when an event  $t2$  occurs and both exit  $S3$  as well as  $S2$ . An internal transition in a state conflicts with transitions that cause an exit from that state.

When there are conflicting transitions, some (but not necessarily all) of the conflicts can be resolved using the *implicit* priority that is based on the relative position of the conflicting transitions in the state hierarchy — more precisely, of their source states. By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing composite states. For example, the transition going out from  $S3$  triggered by  $t2$  has higher priority than the transition going out from  $S2$  triggered by the same trigger in Figure 14-8. This similarly holds true for the transitions triggered by  $t1$  going out from these states, if both their guards result to `true`.

This selection based on the implicit priority can be easily implemented with a straightforward traversal of the active state configuration, starting from the inner-most simple active state outward and searching for the enabled transition (that is, the one whose guard evaluates to `true`). The first such encountered

## Part III: Concepts

---

transition is the one with the highest priority. If all conflicts are resolved by this implicit priority (that is, if one and only one enabled transition has the highest priority), that transition is selected to fire. Otherwise, one of the encountered enabled (and conflicting) transitions with the highest priority is selected arbitrarily. For the example in Figure 14-8, the transition that will fire for each of the states and triggers is given in the following table. (States are shown by their unqualified names for brevity.)

| Active State Configuration | Event | Under Condition   | Fired Transition        |
|----------------------------|-------|-------------------|-------------------------|
| S1, S2, S3                 | t1    | Not g1 and not g2 | None                    |
| S1, S2, S3                 | t1    | g1                | t1[g1]/a1 from S3 to S4 |
| S1, S2, S3                 | t1    | Not g1 and g2     | t1[g2]/a2 from S2 to S5 |
| S1, S2, S3                 | t2    | Always            | t2 from S3 to S6        |
| S1, S2, S3                 | t3    | Always            | t3 from S2 to S5        |
| S1, S2, S4                 | t1    | Not g2            | None                    |
| S1, S2, S4                 | t1    | g2                | t1[g2]/a2 from S2 to S5 |
| S1, S2, S4                 | t2    | Always            | t2 from S2 to S5        |
| S1, S2, S4                 | t3    | Always            | t3 from S2 to S5        |
| S1, S5                     | t1    | Always            | None                    |
| S1, S5                     | t2    | Always            | None                    |
| S1, S5                     | t3    | Always            | t3 from S1 to S6        |
| S6                         | t1    | Not g1 and not g2 | None                    |
| S6                         | t1    | g1 and not g2     | t1[g1]/a1 from S6 to S1 |
| S6                         | t1    | Not g1 and g2     | t1[g2]/a2 from S6 to S2 |
| S6                         | t1    | g1 and g2         | One of the previous two |
| S6                         | t2    | Always            | t2/a2 from S6 to S4     |
| S6                         | t3    | Always            | None                    |

### Exiting States

Every transition (except internal transitions) causes the exiting of one or more states and the entering one or more states. The *least common ancestor* (LCA) state of a (compound) transition is the state that is the first common (direct or indirect) owner state of the source and target states of the (compound) transition. For example, the LCA state of the states S3 and S4 in Figure 14-8 is the state S2. This is also the LCA state of the transition t1[g1]/a1 from S3 to S4. The LCA of the transition triggered by t2 from S3 to S6 is the state machine itself, which may be considered as an implicit topmost state. If one of the states is a direct or indirect owner of the other, then that owner state is the LCA state of the two.

When the selected enabled transition is fired, unless it is an internal transition, all states starting from the innermost state in the active state configuration up to the LCA state of the transition's source and target states (except that LCA) are exited, in that order. Whenever a state is exited, its exit behavior is executed

as the step prior to leaving the state. This means that the exit behaviors are executed in sequence, starting with the innermost active state outward, until (but not including) the LCA state.

The following table shows the states exited (in order) for every transition in Figure 14-8. (States are shown by their unqualified names for brevity.)

| Transition              | LCA State     | States Exited (In Order)       |
|-------------------------|---------------|--------------------------------|
| t1[g1]/a1 from S3 to S4 | S2            | S3                             |
| t1[g2]/a2 from S2 to S5 | S1            | S3 or S4, S2                   |
| t2 from S2 to S5        | S1            | S3 or S4, S2                   |
| t3 from S2 to S5        | S1            | S3 or S4, S2                   |
| t2 from S3 to S6        | State Machine | S3, S2, S1                     |
| t1[g1]/a1 from S6 to S1 | State Machine | S6                             |
| t1[g2]/a2 from S6 to S2 | State Machine | S6                             |
| t2/a2 from S6 to S4     | State Machine | S6                             |
| t3 from S1 to S6        | State Machine | (S3 or S4, S2, S1) or (S5, S1) |

### **Transition Execution**

When the selected transition is fired, its effect behavior is executed. In case of a compound transition, the effects are executed one after the other along the transition path in the following manner. Whenever a choice pseudostate is reached, the guards on its outgoing transitions are evaluated in an arbitrary order, prior to selecting one of the outgoing transitions and executing its effect. If more than one of the guards evaluates to `true`, an arbitrary one is selected. If none of the guards evaluates to `true`, then the model is considered ill-formed and the result is undefined. The `else` guard evaluates to `true` if none of the other guards evaluates to `true`. The effect of the selected transition down the stream is then executed, and so on.

### **Entering States**

Whenever a state is entered, its entry behavior is executed. When a composite state is entered, one of the following occurs happens:

- ❑ **Default entry** — If the fired transition terminates on the very composite state, the entry behavior of the composite state is executed, and then the transition going out from the initial pseudostate contained in that state is fired, executing the behavior associated with that initial transition. Graphically, this is indicated by an incoming transition that terminates on the outside edge of the composite state symbol, such as the transition `t1[g1]/a1` from `S6` to `S1` in Figure 14-8.
- ❑ **Explicit entry** — If the fired transition terminates on a direct or indirect substate of the composite state, then the entry behavior of that composite state is executed first, and then all entry behaviors of the nested states down to the target state are executed in sequence, from the composite state inward. The target substate becomes active and, if it is a composite state, its default entry is executed as described in the previous bullet. Otherwise, its entry behavior is executed in the end. This is the case with the transition `t2/a2` from `S6` to `S4` in Figure 14-8.

## Part III: Concepts

---

- ❑ **Shallow history entry** — If the transition terminates on a shallow history pseudostate, the entry behavior of the composite state is executed first, and then the most recently active substate prior to this entry is entered and becomes active. If the most recently active substate is the final state, or this is the first entry into this composite state, the default history state is entered. This is the substate that is the target of the transition originating from the history pseudostate. If no such transition is specified, the model is ill-formed and the result is undefined (that is, is implementation-dependent). If the substate determined by these rules is a composite state, then it proceeds with its default entry, as already described. Otherwise, it is a primitive state and its entry behavior is executed in the end.
- ❑ **Deep history entry** — If the transition terminates on a deep history pseudostate, the rule is the same as for shallow history, except that the rule of entering the most recently visited substate is applied recursively.

The following table shows the states entered (in order) for every transition in Figure 14-8. (States are shown by their unqualified names for brevity.)

| Transition              | LCA State     | States Entered (In Order) |
|-------------------------|---------------|---------------------------|
| t1[g1]/a1 from S3 to S4 | S2            | S4                        |
| t1[g2]/a2 from S2 to S5 | S1            | S5                        |
| t2 from S2 to S5        | S1            | S5                        |
| t3 from S2 to S5        | S1            | S5                        |
| t2 from S3 to S6        | State Machine | S6                        |
| t1[g1]/a1 from S6 to S1 | State Machine | S1, S2, S3                |
| t1[g2]/a2 from S6 to S2 | State Machine | S1, S2, S3                |
| t2/a2 from S6 to S4     | State Machine | S1, S2, S4                |
| t3 from S1 to S6        | State Machine | S6                        |

In any of these cases, if the ultimately reached active primitive state is not a final state, the special *completion event* of that state is dispatched and processed within the same continuous run-to-completion step. If the target state is a final state, however, the completion event of the enclosing composite state or state machine is dispatched and processed within the same continuous run-to-completion step. The completion event enables only outgoing transitions without specified triggers. If there are no such transitions, the completion event is simply discarded.

### Section Summary

- ❑ While there are no incoming call events, the state machine rests in the active state configuration. When one or more incoming call events occur, the following processing is performed:

- One of the incoming call event occurrences is dispatched and processed.
- Among the *enabled transitions* for the active state configuration, one is selected for firing. If there are no enabled transitions for the active state configuration, the event occurrence is discarded (that is, consumed without any effects).
- The states in the active state configuration that the fired transition leaves are exited.
- The effects and dynamic guards along the transition path are executed.
- The states in the target active state configuration of the fired transition are entered.
- The active state configuration is updated.
- The semantics of event occurrence processing is based on the *run-to-completion* semantics, meaning that an event occurrence can only be dispatched if the processing of the previous occurrence is fully completed.

## Entry and Exit Points

As explained earlier, a composite state is a form of abstraction representing the fact that an object may reside in it for a period of time, exhibiting a particular behavior as defined by the outgoing transitions and substates of that composite state. This is why those transitions that cross the boundaries of composite states (that is, transitions from nested substates to states outside the composite states or vice versa) break the composite states' encapsulation in some way. In addition, sometimes the implicit order of execution of exit activities, transition effects, and then entry activities is not what is needed. Sometimes it is useful to define a precise interface of a composite state in terms of *entry* and *exit* points so that transitions do not cross the composite state boundaries, but instead go in and out from those points.

As shown in Figure 14-9a, entry and exit points are pseudostates usually drawn on the state's outlines. Graphically, an entry point is represented with a small circle drawn on the border of the state machine diagram or composite state, with its name written in the vicinity of the circle. Quite similarly, an exit point is represented with a small circle with a cross. Optionally, entry and exit points may be placed within or outside the border of the state machine diagram or composite state.

When a composite state is encapsulated in such a way, it can be shown in diagrams with its contents suppressed, as in Figure 14-9b, with the intention of hiding the complex contents from the viewer as less relevant. It should be noted that this kind of encapsulation is only presentational and does not have any semantic impact on the contents — the contents are still accessible according to the usual namespace visibility rules.

In general, an *entry point* is a pseudostate within a state machine or composite state, having one and only one outgoing transition to a state or pseudostate within the same state machine or composite state. When a fired transition reaches an entry point of a state machine or composite state, this state machine or composite state is entered, and the single outgoing transition of the entry point is then fired.

## Part III: Concepts

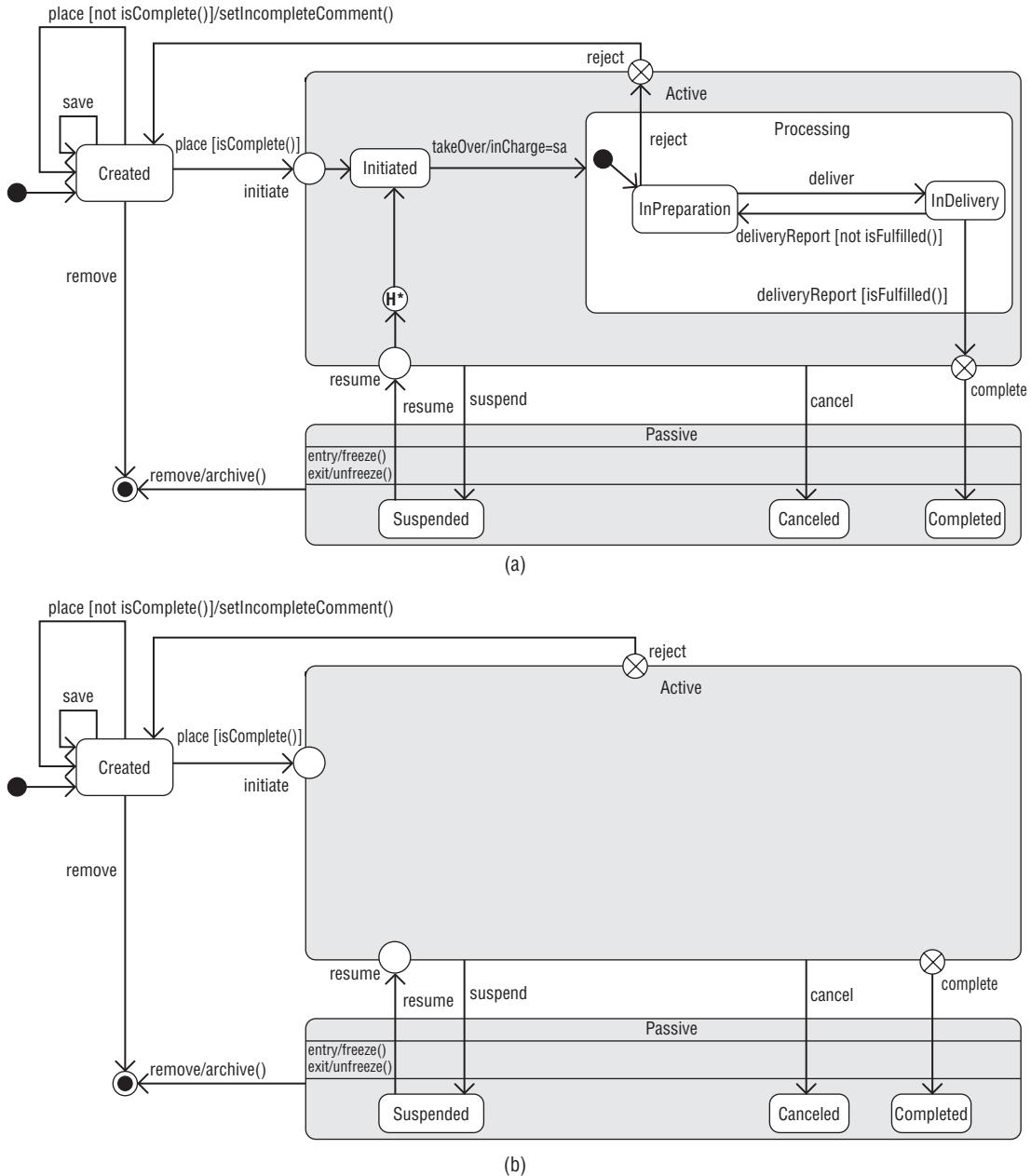


Figure 14-9: Entry and exit points. (a) A complete state diagram of the Order lifecycle with entry and exit points of the state **Active**. (b) A state diagram of the same state machine with the contents of the composite state **Active** suppressed.

An *exit point* is a pseudostate within a state machine or composite state. When a fired transition reaches an exit point within a composite state or state machine referenced by a submachine state (explained later in this chapter), this composite state or submachine state is exited, and the single outgoing transition of the exit point in the state enclosing the submachine or composite state is then fired.

As for all other pseudostates, transitions going out from entry and exit points may not have a trigger.

Entry and exit points are pseudostates that split compound transitions into parts, implying execution of their effects sequentially, one after the other. They slightly change the implicit default order of execution of entry and exit activities and transition effects. If, in a composite state, the exit occurs through an exit point pseudostate, the exit behavior of the composite state is executed *after* the effect behavior associated with the transition coming in to the exit point. This similarly holds true for the effects of the transitions going out from entry points.

### Section Summary

- An *entry point* is a pseudostate within a state machine or composite state, having one and only one outgoing transition to a state or pseudostate within the same state machine or composite state. When a fired transition reaches an entry point of a state machine or composite state, this state machine or composite state is entered, and the single outgoing transition of the entry point is then fired.
- An *exit point* is a pseudostate within a state machine or composite state. When a fired transition reaches an exit point within a composite state or state machine referenced by a submachine state, this composite state or submachine state is exited, and the single outgoing transition of the exit point in the state enclosing the submachine or composite state is then fired.

## Submachines

Once a composite state is isolated from its environment by its entry and exit points, it becomes more easily replaceable by another implementation. Let's consider the order processing example and assume that another strategy of processing an initiated Order is wanted — automatic processing. This means that the system will automatically create a Delivery for an initiated Order if it can be fulfilled; otherwise, the Order will be rejected.

Such replaceable implementations of composite states are supported in UML by *submachine states*. A submachine state is a kind of a state that actually refers to another existing state machine definition.

Figure 14-10a shows the state diagram of the state machine for the Order lifecycle model. In that machine, the state named `HandleOrder` is a submachine state referring to another state machine named `Active`, whose state diagram is shown in Figure 14-10b. The entry and exit points of that submachine state are actually also references to the corresponding connection points of the referenced state machine.

## Part III: Concepts

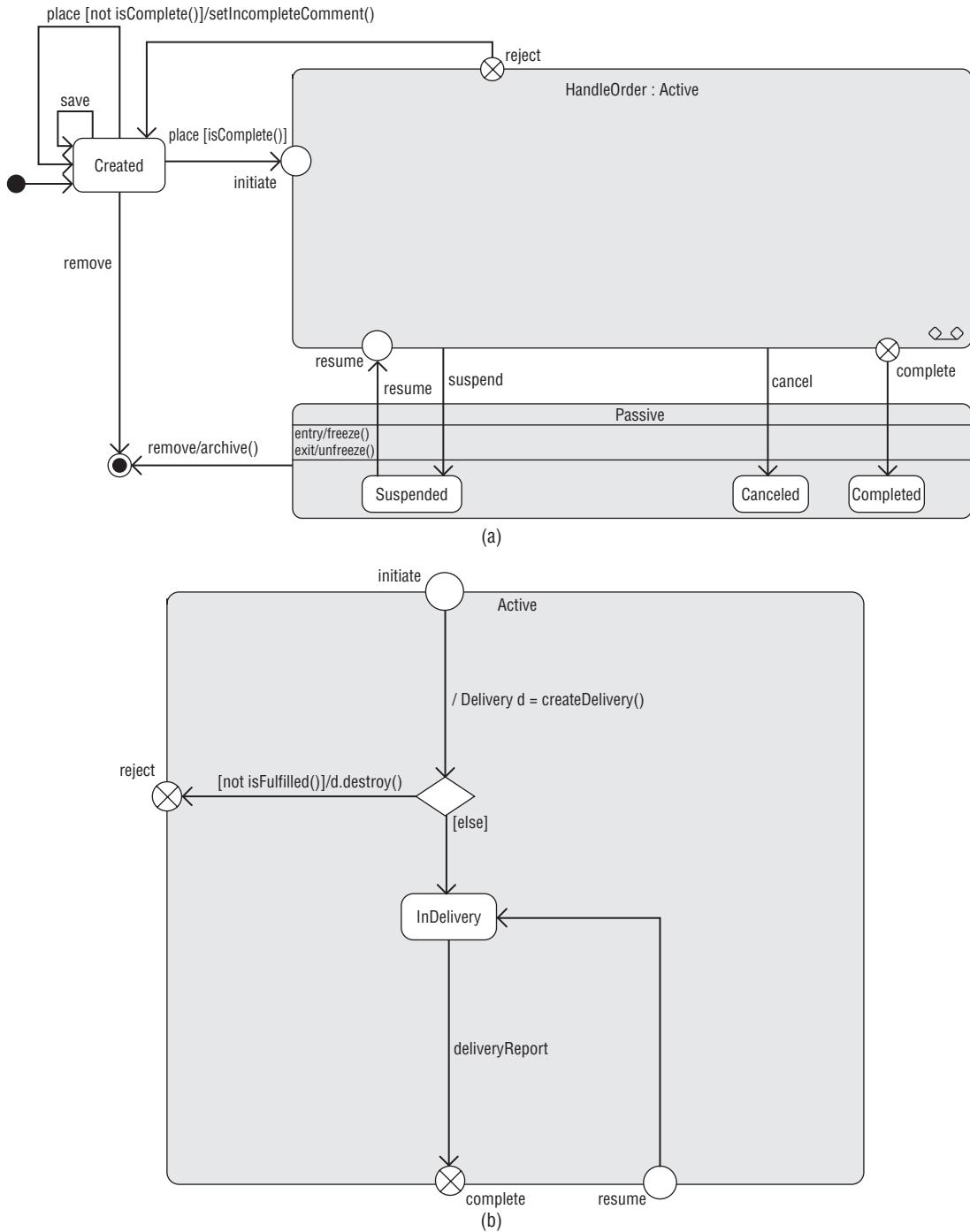


Figure 14-10: Submachines. (a) A state diagram of the Order lifecycle with a submachine state referencing the state machine `Active`. (b) The definition of the referenced state machine `Active`.

A submachine state is logically equivalent to the insertion of the referenced state machine as a composite state in the place of the submachine state. Consequently, every entrance to a submachine state is equivalent to the corresponding entrance to the inserted (referenced) composite state. In particular, it can be entered through its initial pseudostate (as any other composite state), or through one of its entry points (when the entry behavior of the composite state is executed prior to the effects of the transition go out from the entry point).

Similarly, every exit from a submachine state is equivalent to the corresponding exit from the inserted composite state. It can be exited as a result of reaching its final state, by a “group” transition that applies to all substates in the submachine composite state, or through one of its exit points. When it is exited through its exit point, the effect of the transition targeting the exit point is executed first, followed by the exit behavior of the composite state. Of course, if the submachine composite state is to be entered through its initial pseudostate or if it is to be exited as a result of reaching its final state or through a “group” transition that applies to all its substates, it is not necessary to use entry and exit points. The transitions can simply have the submachine state as its target or source state, respectively.

In diagrams, the submachine state is depicted as a normal state, with an optional icon shown in the bottom-right corner of the submachine state in Figure 14-10a, where the text string in the name compartment has the following syntax:

```
submachine_state_name : name_of_referenced_state_machine
```

The symbols for entry and exit points on the outline of a submachine state actually represent references to the corresponding entry and exit points of the referenced state machine. Their notation is the same as for normal entry and exit point pseudostates. The names of these pseudostates are the names of the referenced entry/exit points.

Another motivation for using submachines is better decomposition — factoring of common behaviors and their reuse. When the same composite state implementation is to be reused at several places, it is usually a better idea to refer to that state machine as a separate model element from different enclosing state machines, than it is to copy it several times. The purpose of defining submachine states is to decompose and localize repetitive parts because the same state machine can be referenced from more than one submachine state, even in the same containing state machine.

### Section Summary

- A *submachine state* is a kind of a state that actually refers to another defined state machine.
- A submachine state is logically equivalent to the insertion of the referenced state machine as a composite state at the place of the submachine state.

## Design Considerations

State machines are an efficient technique for modeling lifetime behaviors of objects. Because they are fully formal, state machine models are executable. This allows for some important design and user interaction implications.

## Part III: Concepts

---

First, the generic run-time environment can provide a GUI that enables triggering state machine behaviors of objects in a generic way. This might be very useful during the development and testing of the system. For one selected object, the environment can do the following:

- Display the current state of the object by reading the value of its `state` attribute.
- By means of reflection, offer the options for triggering the object's state machine. For example, the GUI can offer the triggers that the active state configuration react to, along with the ultimate simple target states that would be reached if the trigger is selected. The options can be rendered as radio buttons in a dialog, for example.
- The user can select and trigger one transition, when the objects changes its state, and so on.

The GUI can even visualize the behavior by animating state machine diagrams.

A specific application can take a similar approach, by offering the triggers as actions available in a certain dialog (conversation with the user) for the object being operated on. The dialog can offer other opportunities, such as editing the values of the object's attributes, while the triggers can be offered as actions on the object to take when the dialog is finished (for example, "save," "submit," "reset," "discard," and so on).

Because triggers are usually issued from the GUI in such applications, it is a good idea to encapsulate triggering of a state machine in a command. Such a command can simply make a call event for an object that is an input parameter of that command, or it can also do some other things (such as updating the values of the target object's attributes). For example, in the order processing application, the conversation with the sales agent can be conducted over one or more dialogs/forms/Web pages, where the agent can specify some attribute values of the processing Order for internal use, and ultimately "upload" those values and trigger a state transition with the same command.

### Section Summary

- State machines are an efficient technique for modeling lifetime behaviors of objects. Because they are fully formal, state machine models are executable. This allows for some important design and user interaction implications (such as triggering state machines in a generic way).
- Because triggers are usually issued from the GUI, it is a good idea to encapsulate triggering of a state machine in a command.

# 15

## Collaborations and Interactions

Collaborations represent the vehicle for sketching, designing, illustrating, and documenting the structure of collaborating entities and their communication paths, along with the way they communicate in order to accomplish a certain common task. Interactions are behavioral elements that describe or specify how participants in a collaboration communicate by exchanging messages. This chapter discusses collaborations and interactions in OOIS UML.

## Collaborations and Interactions

The behaviors described so far (that is, methods and state machines) represent formal and executable specifications of how individual instances react to outer stimuli (that is, invocations of their operations). However, they are just small building blocks of the system's overall behavior. Although such internal behaviors are embodied in their classifiers, they do not work in isolation. Instead, instances interact by interchanging messages (that is, calling operations) over certain communication paths, which include links of associations or references provided in a method's arguments or local variables. In particular, during the execution of a method of one instance, operations of other instances may be invoked and their methods executed. The system's behavior emerges from such synergistic cooperation of entities. The vehicle for sketching, designing, illustrating, and documenting the structure of collaborating entities and their communication paths, along with the way they communicate in order to accomplish a certain common task, are known as *collaborations*.

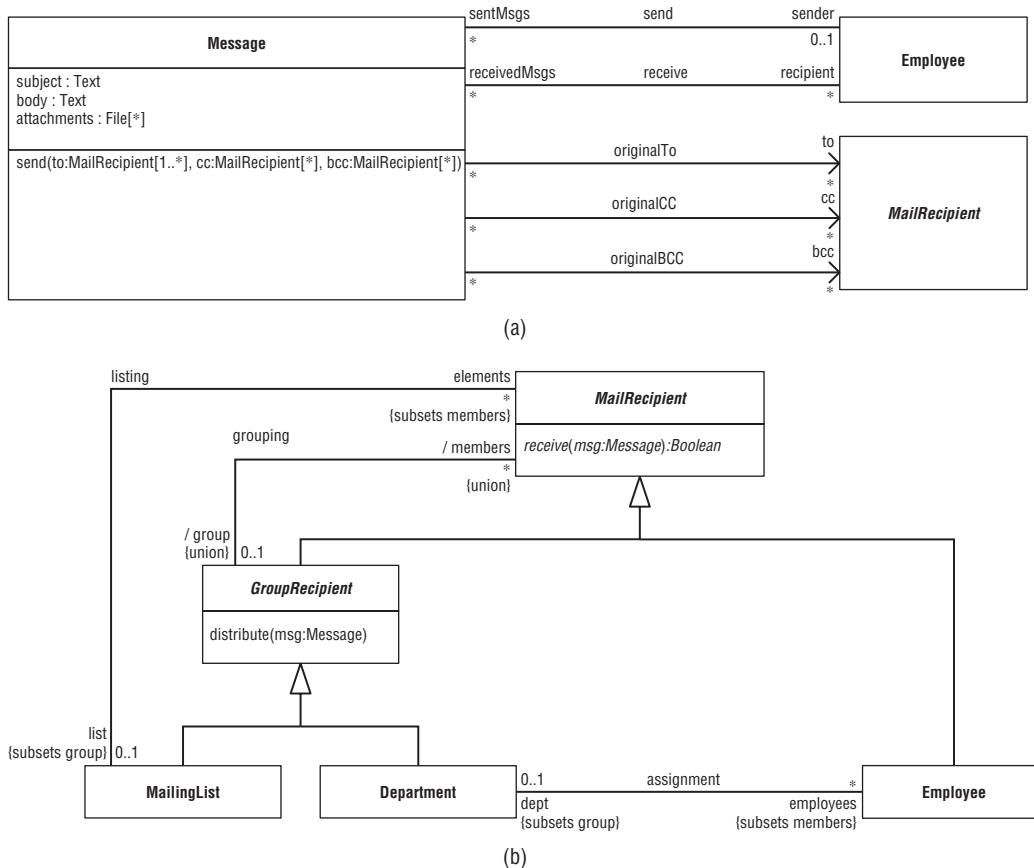
### Motivation

For an illustrative example, let's consider an enterprise information system that incorporates an internal messaging and notification subsystem. Instead of (or in addition to) using standard e-mail infrastructure or other outer communication protocols for transporting traditional e-mail messages,

## Part III: Concepts

---

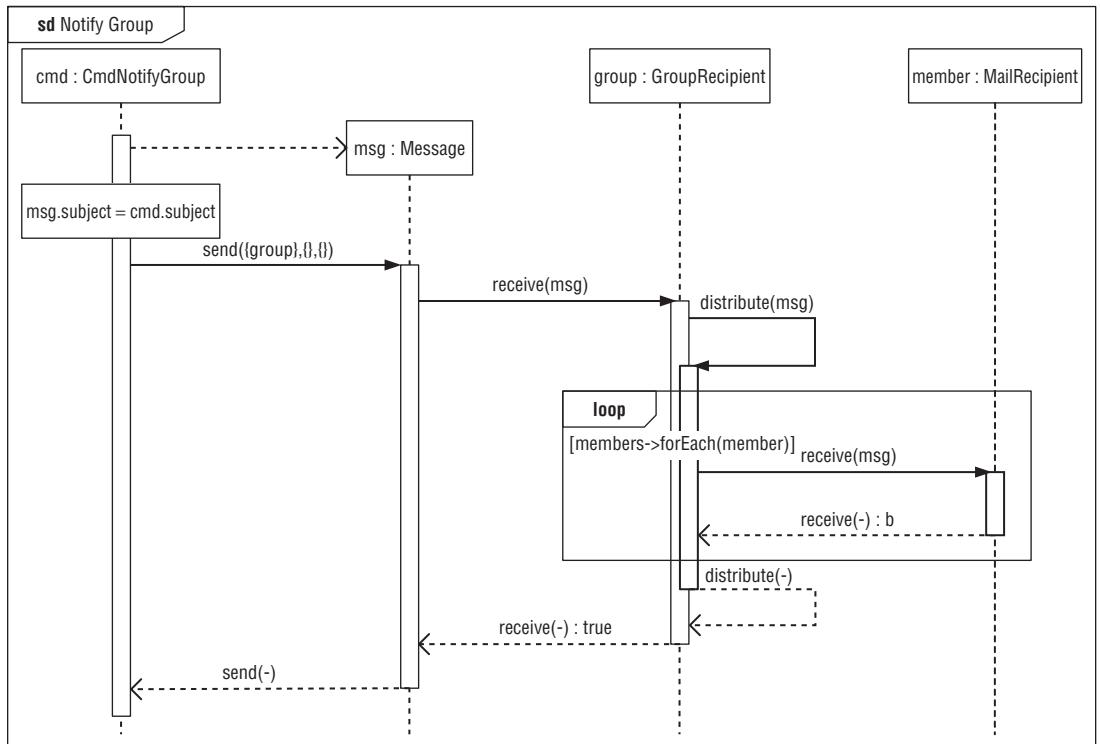
the system manages messages as regular objects in its object space. As shown in Figure 15-1a, a Message has a subject, body, and set of optional attachments. A Message is linked to its sender and ultimate recipients, which are Employees of the enterprise. However, a Message can be originally sent to an abstract recipient such as a mailing list. The concept of an abstract recipient is modeled with Mail Recipient, which can be either a Group Recipient, or an individual Employee, as shown in Figure 15-1b. In addition, a Group Recipient can be an ad-hoc Mailing List, which consists of an arbitrary number of Mail Recipients, thus allowing inclusion of other individual or Group Recipients (such as other Mailing Lists) into mailing lists.



**Figure 15-1: The conceptual model of an enterprise messaging and notification system. (a) A Message is linked to its sender and recipients, which are individuals employed in the company (Employees). (b) The mechanism for organizing Message Recipients into hierarchical structures. Group Recipients can be either ad-hoc Mailing Lists or the company's Departments.**

On the other hand, in order to align the grouping of mail recipients with the company's internal organizational structure, a Department is also a Group Recipient, where its members are its employees. That way, a Message can be distributed to all members of a Group Recipient, be it a Mailing List or a Department, using the same mechanism of passing the Message to each member of the Group.

Figure 15-2 shows the described mechanism. The figure shows a *sequence diagram* that illustrates the way in which objects communicate by making operation calls in order to accomplish a certain task. The task here is to create a notification Message and distribute it to all members of a target Group Recipient.



**Figure 15-2: The sequence diagram for the mechanism for distributing a notification Message to a Group Recipient**

The sequence diagram depicts an interaction among the objects participating in the collaboration. The participants are depicted as rectangles lined up in the upper part of the diagram. The communication between the participants is denoted with horizontal arrowhead lines spanning between the vertical dashed lines that are called *lifelines*. Events in the diagram are ordered chronologically, from the top downward.

The interaction is initiated by an object of a command class, `CmdNotifyGroup`, that is assumed to exist in the model and that is responsible for implementing the functionality of notification and its activation from the user interface. The diagram actually shows the implementation of its `execute` operation. It first creates a new object of the class `Message`. The event of creation is depicted as the dashed line with an open arrowhead pointing to the rectangle representing the new `Message` object. The new object is named and referred to as `msg` throughout the interaction. The command then sets the `subject` attribute of the new `Message` by copying the value of the `subject` attribute of the command, which is assumed to be set from the user interface. The command then calls the operation `send` of the `Message`, providing the reference to the `Group Recipient` named `group` through the argument `to` of the operation.

## Part III: Concepts

---

The executions of methods are rendered as vertical bars along the vertical dashed lifelines of the participants. The method of `send` simply delegates the call to the polymorphic operation `MailRecipient::receive` of the target Group Recipient. The method of this operation defined in `GroupRecipient` delegates the call to the `distribute` operation. The method of this operation distributes the Message to all members of the group in a loop by calling their `receive` operation. The loop is specified by the rectangle with a pentagon in the upper-left corner labeled with `loop`, along with the condition of the loop given within square brackets. Depending on the concrete type of the target Mail Recipient, the method of `receive` will either recursively distribute the Message to its members (when the Mail Recipient is a Group Recipient), or simply link the Message to the `receivedMsgs` property (when the Mail Recipient is an Employee). Finally, the returns of control from the completed methods are rendered as dashed horizontal lines with open arrowheads labeled with the operation names.

In general, a means for sketching, designing, illustrating, and documenting the way that some entities communicate in order to accomplish a certain task is often needed. In UML, this can be done with *interactions*. In addition, the concept for describing the structure of the participating entities and their communication paths is *collaboration*. Their primary purpose is to explain how one particular functionality or mechanism incorporated in the system is (or is to be) implemented. Therefore, they are typically focused on a particular aspect of how the system works, and, thus, incorporate only those details that are deemed relevant to the explanation, while the others are suppressed. Figure 15-3 shows the collaboration that explains the described mechanism.

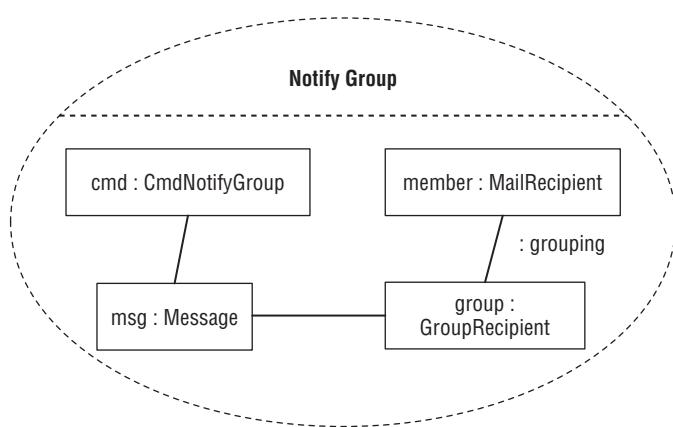


Figure 15-3: The collaboration describing the mechanism for notifying a Group Recipient

In diagrams, a collaboration can be shown as a dashed-outline ellipse. It contains a set of *roles* that describe the participants in the collaboration. In the case of the collaboration `Notify Group` shown in Figure 15-3, the roles are the participants of the described interaction. In addition, the collaboration diagram shows the communication paths that exist between the participants. These are called *connectors*, and are depicted as solid lines connecting the roles. Some of the connectors may represent links of associations. Such is the connector between the `group` and `member` roles, labeled with the name of the association `grouping` preceded by a colon. The other connectors do not represent links of associations, but other kinds of connections between objects that allow their communication, such as referencing from a local variable in a method (`cmd` refers to `msg` over a local variable in its method for `execute`) or a parameter of an operation (`msg` refers to `group` over the parameter `to` of its operation `send`).

### Section Summary

- ❑ *Collaborations* are the vehicle for sketching, designing, illustrating, and documenting the structure of cooperating entities and their communication paths, along with the way they communicate in order to accomplish a certain common task.
- ❑ *Interactions* are descriptions of behavior emerging from passing of information between participating instances.

## Collaborations

A *collaboration* is a kind of classifier that defines a set of *roles* as cooperating entities that are played by concrete instances at run-time, as well as a set of *connectors* that define communication paths between the participating instances. Each of the participating roles performs its specialized function, but the collaboration emphasizes the cooperation of these roles that collectively accomplish certain functionality.

In UML, a collaboration is a classifier. The roles of a collaboration are its properties. They can represent arbitrary or specific instances of classifiers, referred to by the properties of other classifiers or parameters of an operation whose implementation is described by the collaboration. Similar to properties of classes and parameters of operations, roles have names, types, and multiplicities. The types of roles can be arbitrary classes or data types. Roles specify how instances take part in the collaboration, while the classifiers that are their types specify the features of these instances that are required in the collaboration (that is, that concrete instances have to have in order to take part in the collaboration when playing certain roles). The default multiplicity of a role is 1..1. If the role is multi-valued, then it represents a collection of instances playing the same role.

The relationships between the roles relevant for the given collaboration are modeled as connectors between the roles. The connectors specify what communication paths do (or must) exist between the participating instances in order to take part in the collaboration. The connectors can represent links of associations, but can also represent other ways of referencing instances, such as through local variables of methods or parameters of operations. Of course, at run-time, the same concrete instance can play more than one role in one or more collaborations.

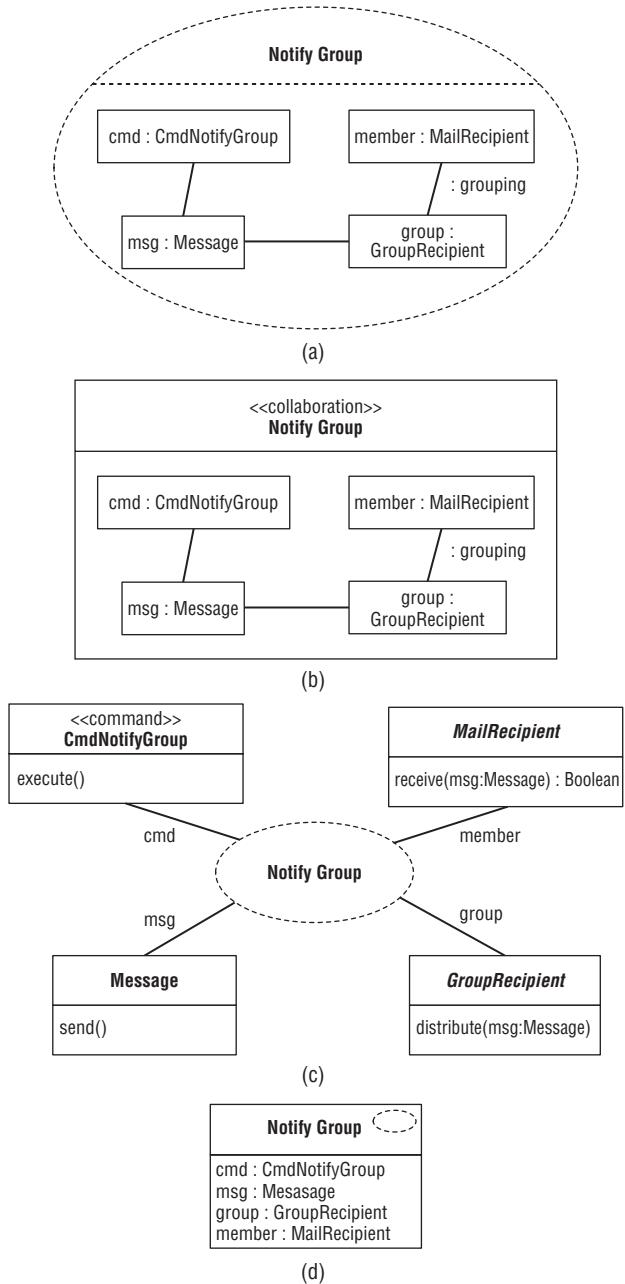
The way the roles cooperate is specified by one or more behaviors attached to the collaboration. It is usually an interaction, but it can be specified in any other way available in UML.

Although collaborations describe what happens at run-time, they do not have executable semantics or any other run-time implications in OOS UML. In other words, they do not represent normative specifications of how the system should work. Instead, they simply illustrate how the system works or is supposed to work. In fact, the behaviors of collaborations are those that provide the illustration. Collaborations themselves provide a context for those illustrations of behavior.

A collaboration is a simple grouping concept that provides the environment for the modeling entities that synergistically work together to accomplish a certain task — the classifiers, the roles played by instances of these classifiers, the connectors representing the communication paths between these instances, and the attached behaviors that describe the cooperation. A collaboration is not directly instantiated, and is not executable. Instead, the cooperation described by the collaboration is a possibly selective view of a concrete situation of actual cooperation that happens at run-time between concrete instances.

## Part III: Concepts

---



**Figure 15-4: Alternative notations for collaborations and their roles:** (a) disclosing the internal structure of the collaboration (that is, showing the connectors), (b) using the general classifier symbol and disclosing the structure of the collaboration, (c) suppressing the internal structure, but showing the actual classifiers of the roles, and (d) using the general classifier symbol with properties shown as text.

In diagrams, a collaboration is shown as a dashed-outline ellipse. Its internal structure (that is, its roles and connectors) may be shown inside the ellipse, as depicted in Figure 15-4a.

Although the dashed ellipse is commonly used in UML textbooks, it is not a practical notation because the geometrical constraints of the ellipse take up far too much useful drawing space. An alternative, most practical, and recommended notation for collaborations is available to all classifiers, as shown in Figure 15-4b. A collaboration can be drawn as a rectangle box with its structure (roles and connectors) contained inside a box (that is, inside a separate compartment) within the rectangle. The keyword `<<collaboration>>` is then usually written above the name of the collaboration.

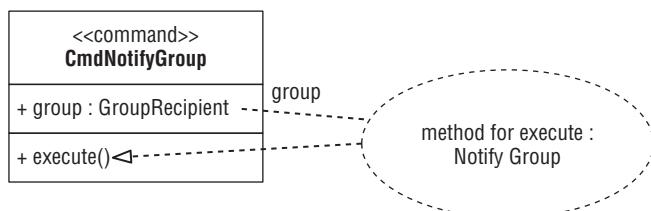
Alternatively, a line may be drawn from the collaboration symbol to each of the symbols of classifiers that are the types of roles in the collaboration, as shown in Figure 15-4c. The name of the role is written close to each line end at the classifier. In this manner, a diagram can show more details of the classifiers whose instances take part in that particular collaboration. Finally, the generic symbol for classifiers can be used for a collaboration, too, while its roles may be shown as usual properties in the textual notation (see Figure 15-4d). The small dashed ellipse in the upper-right corner of the rectangle indicates the kind of the classifier (that is, the collaboration).

Although a collaboration can own other nested classes and data types (for example, the types of its roles), it usually does not own any. Instead, a collaboration simply refers to these classifiers.

As a classifier, a collaboration can be directly owned by a package. This is usually the case when the collaboration describes an important mechanism of a system or a protocol of communication between some participants.

In addition, a collaboration can be owned by another enclosing classifier (such as a class). In that case, the collaboration specifies the implementation of that classifier or some of its operations. In particular, a collaboration is a suitable means for describing the implementation of a method of an operation; actually, the collaboration in Figure 15-4 is a description of the method for the operation `CmdNotifyGroup::execute`.

In such a case, the collaboration is related to the class or the operation by a *collaboration use*. A collaboration use is a named model element that refers to a collaboration and relates the roles in the collaboration to the properties of the classifier or the parameters of the operation. For example, in Figure 15-5, there is a collaboration use named `method for execute` that refers to the collaboration `Notify Group`. It appears as a dashed ellipse with the optional name of the collaboration use, and the name of the referenced collaboration following a colon. For each binding of a role, there is a dashed line connecting the collaboration use and the element that plays that role with the name of the role written close to the element, as shown in the figure. In addition, the collaboration can be related to the operation or the class it realizes by a realization dependency, as is the case in Figure 15-5.



**Figure 15-5: A collaboration use**

There can be many different collaboration uses referring to the same collaboration, each providing a different binding of its roles. This option is particularly useful for documenting an application of design

## Part III: Concepts

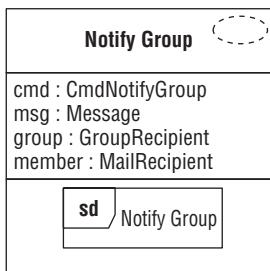
patterns. Namely, a design pattern can be often described using a collaboration. Such a collaboration describes the pattern in terms of the participating roles, their classifiers, features, and relationships in an abstract manner. Those classifiers can even be owned by the collaboration. A use of such collaboration then represents a concrete application of the pattern in terms of binding concrete participants to abstract roles in the pattern.

### Section Summary

- ❑ A *collaboration* is a kind of classifier that defines a set of *roles* as cooperating entities (which are played by concrete instances at run-time), as well as a set of *connectors* (which define communication paths between the participating instances).
- ❑ Collaborations do not have formal semantics, and are not executable in OOS UML.
- ❑ A *collaboration use* is a named model element that refers to a collaboration and relates the roles in the collaboration to the properties of a classifier, or the parameters of an operation whose implementation is described by the collaboration.

## Interactions

*Interactions* are units of behavior of their enclosing classifiers. For the ongoing example, the interaction Notify Group can be a behavior contained by the collaboration Notify Group, as shown in Figure 15-6. An interaction describes the behavior emerging from the cooperation of parts of its enclosing classifier.

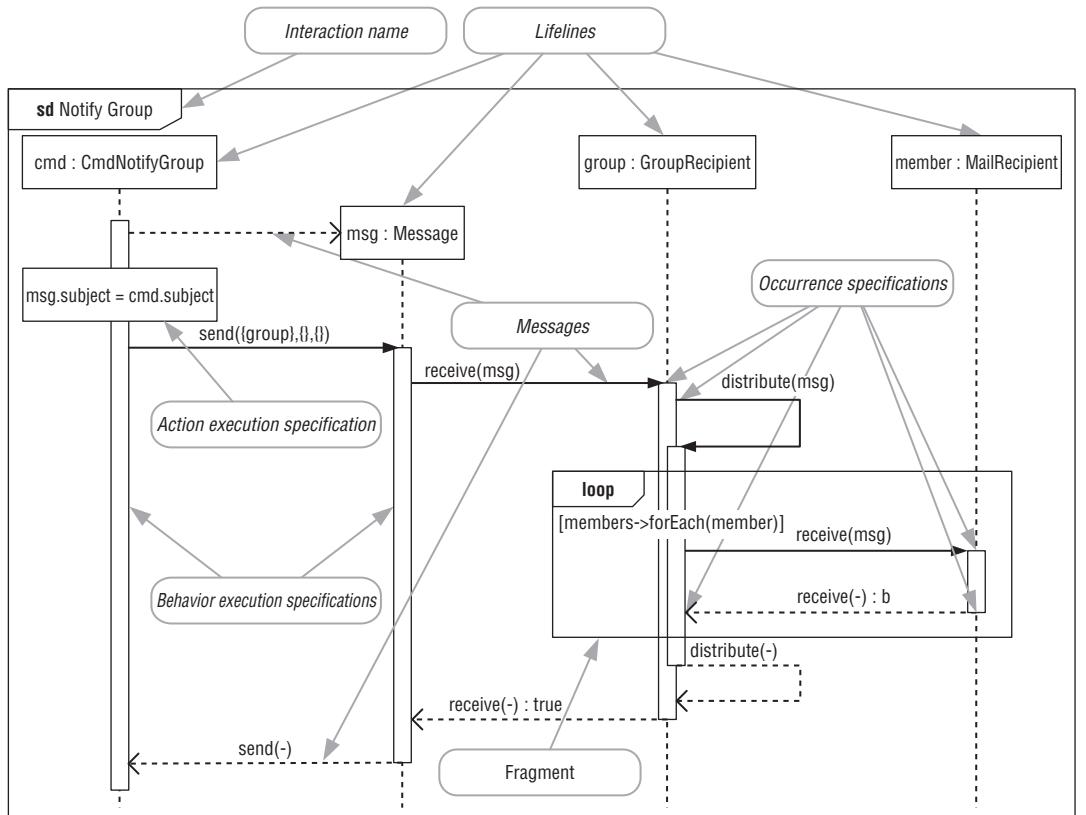


**Figure 15-6: Interaction as a unit of behavior in its enclosing classifier (collaboration in this case)**

An interaction can be shown in a kind of interaction diagram, like the one shown in Figure 15-7<sup>1</sup>, called a *sequence diagram*, because it conveys the sequencing of messages interchanged between the participants

<sup>1</sup>The rounded callouts, along with their arrowhead lines, in this figure are not part of the UML notation and serve as annotations only.

of the interaction. The elements of a diagram are framed in a rectangle with a pentagon in its upper-left corner stating the interaction name preceded by the keyword `sd`.



**Figure 15-7: Basic elements of interactions. The rounded callouts along with their arrowhead lines are not part of the UML notation and serve as annotations only.**

The participants of the interaction are called *lifelines*. A lifeline represents one entity taking part in the interaction. A lifeline may (but need not) have a name. A lifeline may represent an instance of a classifier, or even a classifier, when invocations of static operations must be modeled by messages targeting that lifeline. A lifeline can (but need not) refer to a property of its enclosing classifier (such as the role of a collaboration or property of a class) or to a parameter of an operation. In addition, a lifeline can represent an instance created within a method or referred to by a local variable of a method.

In interaction diagrams, a lifeline is rendered as a solid-outlined rectangle with a vertical dashed line below. Its optional name is written inside the rectangle. If the lifeline represents an instance of a classifier, its type is shown following the colon, as in Figure 15-7. If the lifeline refers to a multi-valued multiplicity element (such as a property or parameter), the lifeline can have a “selector” that specifies which of the elements in the referred collection is represented by the lifeline. The selector is given as an arbitrary expression enclosed in square brackets following the lifeline name. For example, a lifeline text can look like this:

```
member[i] : User
```

## Part III: Concepts

---

If the selector is omitted, then the lifeline represents an arbitrary element in the referred collection.

An interaction can be owned by a class or data type, and can illustrate an implementation of its method. In that case, a lifeline named `self` represents the host object of the classifier owning the method. For the current example, the shown interaction could have been owned by the class `CmdNotifyGroup`, and the lifeline `cmd` could have been renamed to `self`.

In sequence diagrams, the rectangles representing lifelines are distributed horizontally. They have attached vertical dashed lines that represent their lifetimes.

Interactions focus on the *messages* between the lifelines and their sequencing, which is deemed important for the understanding of the interactions. The data that the messages convey, and that the lifelines store, may also be important and shown in the diagrams, but manipulation of data is not the central point in interactions. Messages are rendered as arrows between lifelines.

Different kinds of messages and their meanings are described later in this chapter. Most of the messages in Figure 15-7 represent calls of operations. As a result of a call, the method of the invoked operation is executed. Executions of methods or other behaviors are symbolized with vertical white or gray bars along the vertical dashed lines of the lifelines.

Some of the elements in an interaction, such as emission or reception of a message, or starting and ending of a behavior execution, represent specifications of occurrences of special importance to the interaction. These are called *occurrence specifications*.

Executions of actions within a method or other behavior executions can be modeled as *action execution specifications* attached to lifelines. For the example in Figure 15-7, such is the specification shown in the rectangle below the head of the `cmd` lifeline.

Another basic element of an interaction is an interaction *fragment*, which can represent a loop or an *if-then-else* construct, for example. In Figure 15-7, the fragment is a loop fragment, representing an iterative repetition of the sequence of occurrences given within the fragment. In general, a fragment is a piece of an interaction that is conceptually like an interaction by itself, except that it is not self-contained, and is always incorporated in an interaction.

Most of these basic elements, as well as some more advanced ones, are described in more detail later in this chapter.

### Section Summary

- ❑ *Interactions* are units of behavior of their enclosing classifiers. An interaction describes the behavior emerging from the cooperation of parts of its enclosing classifier.
- ❑ A *lifeline* represents one participating entity. A lifeline can (but need not) refer to a property of its enclosing classifier, or to a parameter of an operation.
- ❑ Interactions focus on the *messages* between the lifelines and their sequencing.
- ❑ Executions of actions within a method (or another behavior execution) can be modeled as *action execution specifications* attached to lifelines.

- An interaction *fragment* is a piece of interaction that is conceptually like an interaction by itself, except that it is not self-contained, and is always incorporated in an interaction.

## Semantics of Interactions

Interactions do not have fully formal semantics, and are, thus, not executable in OOA UML. They are not normative specifications of behavior, and do not have any run-time implications on the system. Instead, they are simple illustrations of what can (or cannot) happen at run-time, but typically do not describe all possible situations that may happen at run-time. There are normally many other legal and possible situations exhibited by the running system that are not documented through the modeled interactions. Some projects may, however, request that all possible traces of a system or its parts be documented through interactions. This is, however, most often impractical and unnecessary.

The basic semantic interpretation of an interaction is that it describes a set of possible or legal *traces of events* that can occur at run-time, whereby the trace means a sequence of event occurrences that are manifestations of occurrence specifications given in the interaction. For example, the interaction shown in Figure 15-8a<sup>2</sup> is interpreted as a statement that, during the execution of the system, some (or particular) instances  $x$ ,  $y$ , and  $z$ , of type  $x$ ,  $y$ , and  $z$ , respectively, can interact and cause the following outwardly visible sequence of occurrences:

$x1, x2, y1, y2, z1, z2, y3, y4, y5, y6, z3, z4, y7, y8, x3, x4$

This is the only possible trace that can be inferred from this interaction. However, it does not mean that this is the only trace that can result from the running system. The others can be specified by other interactions, or do not have to be specified at all.

Two basic semantic rules define the relationships of chronological ordering between occurrences specified in an interaction:

- Two occurrence specifications,  $x_i$  and  $x_j$ , which belong to the same lifeline, where  $x_j$  is below  $x_i$ , are chronologically ordered. This means that an occurrence specified by  $x_j$  may happen after (or at the same time as) an occurrence specified by  $x_i$ , denoted with  $x_i \leq x_j$ . For example, in Figure 15-8a,  $x1 \leq x2 \leq x3 \leq x4, y1 \leq y2 \leq \dots \leq y7 \leq y8$ , and  $z1 \leq z2 \leq z3 \leq z4$ .
- The message reception  $r$  may happen after (or at the same time as) the message sending  $s$  of the same message, denoted with  $s \leq r$ . For example, in Figure 15-8a,  $x2 \leq y1, y2 \leq z1, z2 \leq y3, y6 \leq z3, z4 \leq y7$ , and  $y8 \leq x3$ .

These two basic rules will be extended or redefined by the semantics of some other elements of interactions, such as fragments.

Applying the two given rules to the example in Figure 15-8a, the following can be concluded, which defines exactly the trace given previously:

$x1 \leq x2 \leq y1 \leq y2 \leq z1 \leq z2 \leq y3 \leq y4 \leq y5 \leq y6 \leq z3 \leq z4 \leq y7 \leq y8 \leq x3 \leq x4$

---

<sup>2</sup>The rounded callouts, along with their arrowhead lines, in this figure are not part of the UML notation and serve as annotations only.

## Part III: Concepts

Simply put, this means, for example, that  $x$  starts the execution of its behavior first, then it calls the operation  $op1$  of  $y$ , then  $y$  calls  $op2$  of  $z$ , and so on. Note that action and behavior executions have occurrence specifications that represent their start and completion. For example, the execution of  $action1$  can be considered to take some or no time ( $y4 \leq y5$ ). Reception of an operation call message and activation of its method are usually represented by the same occurrence specification, although they do not have to be.

The semantics of interaction are exactly what is stated by the two given basic rules — no more and no less can be concluded about the ordering of occurrences. Consequently, there can be occurrence specifications that cannot be chronologically related to one another.

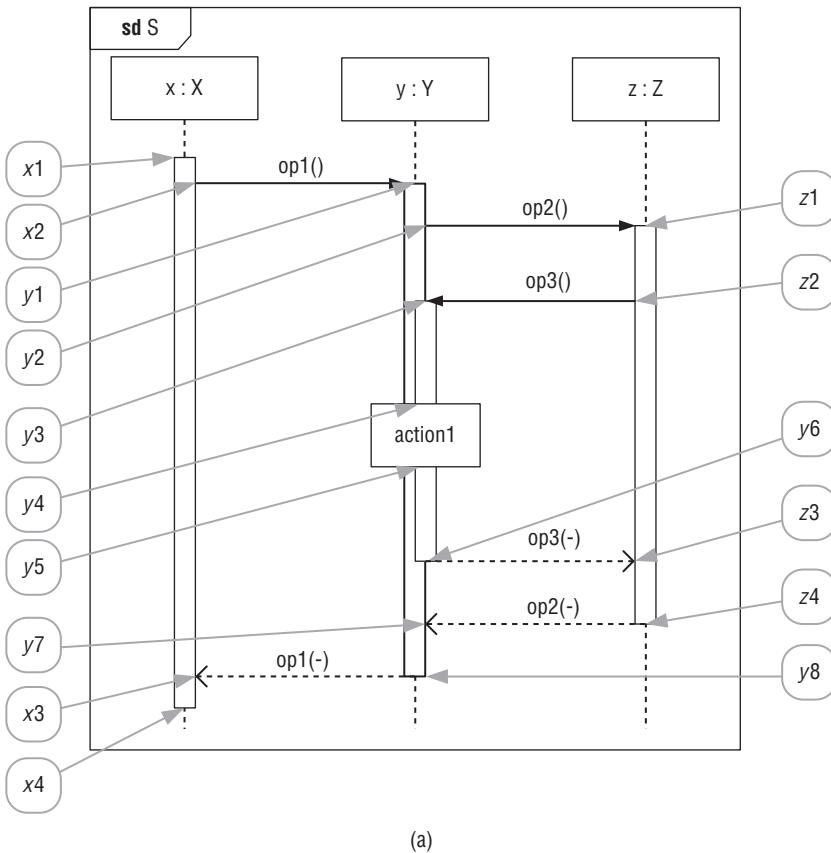


Figure 15-8: Examples of ordered and partially ordered sets of occurrences implied from the semantics of interactions. The basic semantic rules of ordering occurrences cause that the occurrences in the interaction shown in (a) are completely ordered, while some of the occurrences in (b) are ordered and some are unrelated. The rounded grayed callouts are annotations that enable you to refer to the occurrence specifications, and are not part of the UML notation.

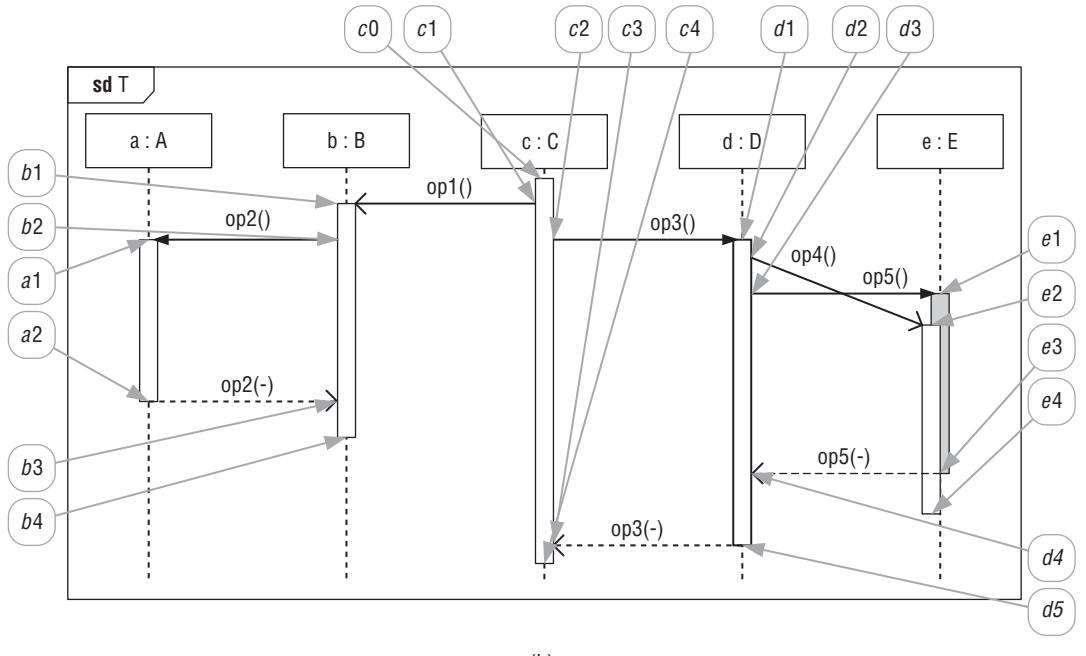


Figure 15-8: continued

Consider the example in Figure 15-8b. The first event that comes about is specified by  $c_0$  and is followed by  $c_1$ . But  $c_1$  is an asynchronous operation call of  $b.op1()$ , denoted with an open arrowhead line. Because it is asynchronous, the execution of the behavior of  $c$  continues, and  $c$  issues a synchronous message call of  $d.op3()$ . From then on, the methods of  $b.op1$  and  $d.op3$  execute concurrently, and nothing can be concluded about the chronological ordering of occurrences in these two threads of control. Applying the basic semantic rules, the following can be concluded:

$$c_0 \leq c_1 \leq b_1 \leq b_2 \leq a_1 \leq a_2 \leq b_3 \leq b_4, \text{ and}$$

$$c_0 \leq c_1 \leq c_2 \leq d_1 \leq d_2 \leq d_3 \leq d_4 \leq d_5 \leq c_3 \leq c_4, \text{ and}$$

$$c_0 \leq c_1 \leq c_2 \leq d_1 \leq d_2 \leq d_3 \leq e_1 \leq e_2 \leq e_3 \leq e_4, \text{ and}$$

$$c_0 \leq c_1 \leq c_2 \leq d_1 \leq d_2 \leq e_2 \leq e_3 \leq d_4 \leq d_5 \leq c_3 \leq c_4, \text{ and so on.}$$

Note, however, that nothing can be concluded about the chronological ordering between, for example,  $b_2$  and  $c_2$ ,  $b_3$  and  $c_4$ ,  $b_4$  and  $c_3$ ,  $d_5$  and  $e_4$ ,  $b_4$  and  $e_2$ , and so on. These occurrences are unrelated, and are taken to be executed concurrently, meaning that they can occur in any order in a trace.

In fact, interactions apply the *interleaving* semantics of such unrelated sequences. Interleaving means the merging of two or more traces such that the events from different traces may come in any order in the resulting trace, while events within the same trace retain their order. For that reason, the interaction

## Part III: Concepts

---

in Figure 15-8b can be interpreted as a set of all traces obtained by interleaving of traces of the two concurrent threads, such as the following:

$c0, c1, b1, b2, a1, a2, b3, b4, c2, d1, d2, d3, e1, e2, e3, d4, d5, c3, c4, e4$

or

$c0, c1, c2, d1, d2, d3, e1, e2, e3, e4, d4, d5, b1, b2, a1, a2, c3, c4, b3, b4$

and so on.

The interaction in Figure 15-8b shows another interesting case that can be modeled — *message overtaking*. Although  $d$  issued an asynchronous operation call of  $e.op4()$  before the call of  $e.op5()$  (that is,  $d2 \leq d3$ ), the latter overtook the former, and the reception of the call of  $op5$  occurred before the reception of the call of  $op4$  (that is,  $e1 \leq e2$ ). The intention of the modeler here was simply to emphasize that such a case could possibly happen at run-time, and that the interaction illustrated how the system was expected to behave.

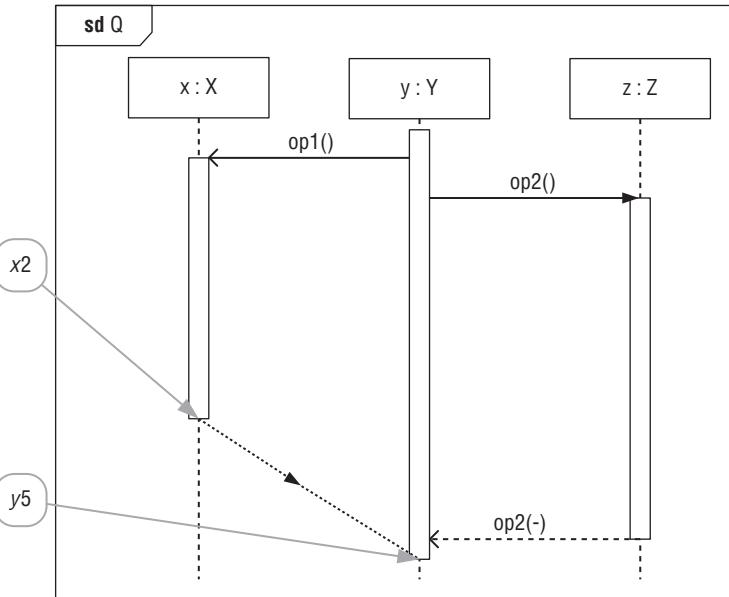
In some special cases, there is a need to restrict the set of possible interleaved traces by imposing an explicit ordering of occurrence specifications where this is not imposed by the basic semantic rules. For the example in Figure 15-9<sup>3</sup>, the basic semantic rules do not define any ordering between occurrences  $x2$  and  $y5$ . However, it may be necessary to model the requirement that  $x2$  must happen before  $y5$ . For that reason, there is a *general ordering* relationship, which is a binary relationship between two occurrence specifications that describes that one occurrence must happen before the other in any valid trace. A general ordering relationship may have a name and may appear anywhere in an interaction, but only between occurrence specifications. It is shown by a dotted line connecting the two occurrence specifications. An arrowhead placed somewhere in the middle of the dotted line indicates the direction of the precedence relation, and points from the preceding to the succeeding occurrence specification.

An interaction is a generalizable and redefinable model element. An interaction can specialize another interaction owned by the same or any other classifier. Specializing an interaction simply means extending the set of traces defined by the generalizing interaction with the set defined by the specializing interaction. In addition, an interaction owned by a specializing classifier can redefine the one from the generalizing classifier. Redefining simply means replacing the redefined interaction from the generalizing classifier.

In general, interactions can be used for different purposes. They can be used by modelers to document or communicate an idea of a desired behavior. They can also be used in later design phases to specify the system's key mechanisms and implementations of various behavioral parts of the system. During testing, the expected traces of the system for a set of tests can be described as interactions, and compared with those produced in the actual test. Interactions can be produced by humans, or can be generated from automated tools that trace executions of the running system and represent a recorded trace by an interaction. Specialized testing tools can even use interactions for some more rigorous purposes. For example, they can report violations of recorded traces against those specified by interactions.

---

<sup>3</sup>The rounded callouts, along with their arrowhead lines, in this figure are not part of the UML notation and serve as annotations only.



**Figure 15-9: General ordering between occurrence specifications. The rounded grayed callouts are annotations that provide references to the occurrence specifications, and are not part of the UML notation.**

In any case, in OOSI UML, interactions do not have normative, executable meaning for the system under construction. The behavior of the system cannot be constructed from interactions, nor is it constrained by the interactions defined in the model.

### Section Summary

- ❑ Interactions do not have fully formal semantics, and are not executable in OOSI UML. The behavior of the system cannot be constructed from interactions, nor is it constrained by interactions in the model.
- ❑ The basic semantic interpretation of an interaction is that it describes a set of possible or legal *traces of events* that can occur at run-time.
- ❑ Following are two basic semantic rules that define the relationships of chronological ordering between occurrences specified in an interaction:
  - ❑ Two occurrence specifications,  $x_i$  and  $x_j$ , which belong to the same lifeline, where  $x_j$  is below  $x_i$ , are chronologically ordered. This means that an occurrence specified by  $x_j$  may happen after (or at the same time as) an occurrence specified by  $x_i$ .
  - ❑ The reception of a message may happen after (or at the same time as) the sending of the same message.

### Messages

A message in an interaction specifies a particular communication of a certain kind between the sender and the receiver lifelines. A message relates two occurrence specifications — one sending and one receiving. If the sending and the receiving occurrence specifications of the same message are on the same lifeline, the sending one must be placed before (that is, above) the receiving one. A message can optionally refer to the connector over which it is sent.

Graphically, a message is a line from the sender lifeline to the receiver lifeline, possibly having multiple segments. Every segment must be either horizontal or downwards when traversed from the send to the receive end. As already noted, the send and receive ends may both be on the same lifeline.

In OOIS UML, the following subset of sorts of messages available in standard UML are used (see Figure 15-10):

- An *asynchronous message* models the communication that emerges from an asynchronous operation call action executed by the sender instance. Asynchronous messages are solid lines with open arrowheads, as shown in Figure 15-10 for the call of op2.

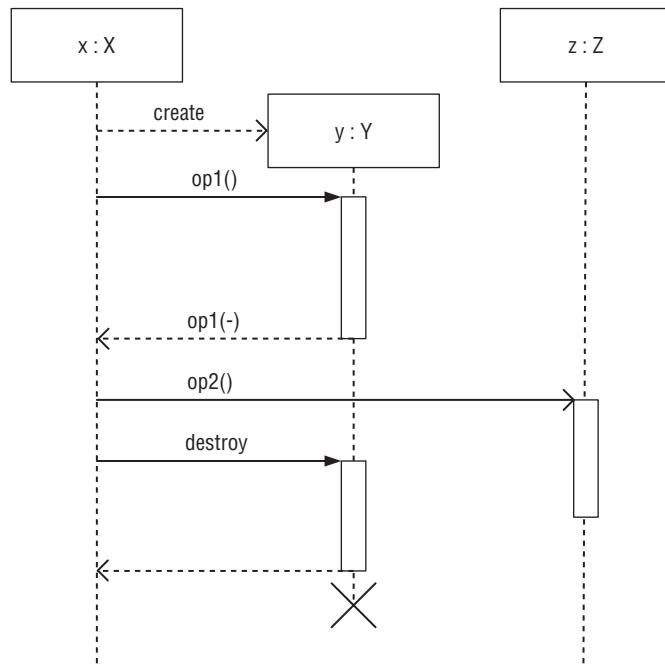


Figure 15-10: Sorts of messages used in OOIS UML

- A *synchronous message* models the communication that emerges from a synchronous operation call action executed by the sender instance. Synchronous messages are solid lines with filled arrowheads, as shown in Figure 15-10 for the call of op1.
- A *reply message* models the communication that happens when a method activated by a synchronous operation call has completed and the control (along with optional output arguments

and return value) is returned to the caller. Reply messages are drawn as dashed lines with open arrowheads, as shown in Figure 15-10 for the return from op1. For each message representing a synchronous operation call, there will normally be a reply message from the called lifeline back to the calling lifeline before the behavior of the calling lifeline proceeds. UML does not forbid occurrence specifications on the caller's behavior execution specification (the thick bar on the lifeline in the diagram) between the synchronous call and the return occurrence specifications. In OOIS UML, however, although they're allowed, such occurrence specifications make no sense because the caller's behavior is blocked during the synchronous call.

- ❑ A *create message* models the communication that emerges from a Create Instance action executed by the sender instance. Create messages are drawn as dashed lines with open arrowheads pointed to the lifeline created by that action, as shown in Figure 15-10 for the creation of y. The keyword `create` can be written near the line. Optionally, to model the execution of the constructor of the created instance, a behavior execution specification can be drawn as a thick bar immediately below the receiver lifeline head, with an optional reply message to the caller modeling the return from the constructor.
- ❑ An operation call message (synchronous or asynchronous) can model the invocation of the *destroy* operation of an object, as shown for the destruction of y in Figure 15-10. The behavior whose activation is modeled at the receiver's side is then its destructor. Because the recipient object ceases to exist after the destructor is completed, its destruction is modeled with a *destruction event* at its lifeline. A destruction event represents the destruction of the instance described by the lifeline containing it. The symbol for the destruction event is a cross ( $\times$ ) at the bottom of a lifeline. There can be no more occurrence specifications on that lifeline below the destruction event. The optional reply message from the destructor can start from the same point, or immediately above the destruction event, as shown in Figure 15-10.

Each of these kinds of messages can carry optional arguments of the call (input arguments) or reply (output arguments and return value). The arguments of the message should conform to the parameters of the called operation, including constructor or destructor. The name of the called operation and the arguments are shown on the message line as a signature of the call or reply, as shown in the previous sequence diagrams. The syntax for the signature is as follows:

```

message-signature ::= prop-assignmentopt operation-name arg-signatureopt ret-specopt
prop-assignment ::= property-name =
arg-signature ::= (arg-listopt)
ret-spec ::= : return-value

arg-list ::= argument
arg-list ::= argument , argument-list

argument ::= param-specopt argument-value
argument ::= property-name = output-parameter-name arg-valopt
argument ::= -

param-spec ::= parameter-name =
arg-val ::= : argument-value

```

The argument value (*argument-value*) and return value (*return-value*) can be any concrete value (that is, a literal). It can also be a symbolic value referring to an instance from the scope of the interaction, or representing an abstract, arbitrary value. Actually, it can be an arbitrary expression that is interpreted by the human readers, and not by the tools and model compilers. A return value (*return-value*) and

## Part III: Concepts

---

property assignment (*prop-assignment*) are used only for reply messages. A property assignment is shorthand for including the action that assigns the return value, or the value of an output argument to a receiver's property. If the argument list contains only argument values without names, they are matched in order either by a value or by a dash (-) that represents an omitted argument. If the argument list uses parameter names to identify the arguments, then arguments may freely be omitted and ordered. Omitted arguments get unknown values.

The following table shows several illustrative examples with their meanings.

| Example                                | Meaning                                                                                                                                                                                |
|----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| notify(listener, -, 5)                 | Parameters are matched in order. The second argument is undefined.                                                                                                                     |
| notify(count=5, who=listener)          | Parameters are matched by name. The unlisted arguments are undefined.                                                                                                                  |
| myTime = getTime() : t                 | This is a reply message with an assignment of the return value <i>t</i> to the property <i>myTime</i> of the sender.                                                                   |
| assign(myToken=assignedToken:5) : true | This is a reply message returning the value <i>true</i> , with an assignment of the value 5 of the output parameter <i>assignedToken</i> to the property <i>myToken</i> of the sender. |

### Section Summary

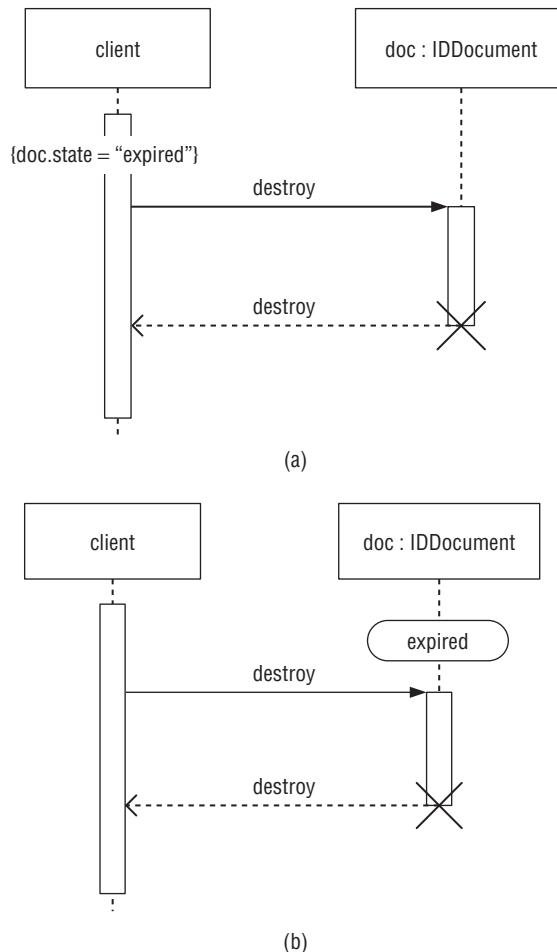
- ❑ A message in an interaction specifies a particular communication of a certain kind between the sender and the receiver lifelines.
- ❑ In OOIS UML, several sorts of messages are used:
  - ❑ An *asynchronous message* models an asynchronous operation call.
  - ❑ An *synchronous message* models a synchronous operation call.
  - ❑ A *reply message* models a return from an operation.
  - ❑ A *create message* models creation of an instance.
- ❑ A message can carry optional arguments of the call (input arguments) or reply (output arguments and return value).
- ❑ A *destruction event* represents the destruction of the instance described by the lifeline containing it.

## Fragments

*Interaction fragments* are pieces of interactions that are conceptually like interactions themselves, but that, in most cases, extend or redefine the basic semantics of interactions described so far. In fact, the semantics

of an entire interaction are built from its constituting fragments. One such general extension is that an interaction defines a set of legal (valid) traces *and* a set of illegal (invalid) traces. The union of these two sets does not have to cover the entire universe of traces — that is, there can be traces whose validity cannot be determined from the considered interaction. In other words, interactions can also be used to describe illegal or prohibited traces.

One simple kind of fragment is a *state invariant*. A state invariant is a constraint placed on one life-line that defines a condition or restriction on the participants of the interaction that must hold at a certain moment at run-time. More precisely, the trace defined by an interaction is valid only if the constraint evaluates to `true` when the interaction reaches its point. It may specify, for example, a constraint on values of attributes or variables, on states of objects, and so on. For example, the constraint that the object of type `IDDocument` must be in the state `expired` before it is destroyed can be shown as in Figure 15-11a.



**Figure 15-11: State invariant, depicted (a) using the notation with curly braces and (b) using the state symbol**

## Part III: Concepts

---

In general, the constraint defined by a state invariant is assumed to be evaluated immediately prior to the execution of the occurrence specification that immediately follows the invariant on the same lifeline, such that all actions that are not explicitly modeled have been executed before the constraint is evaluated. If the constraint evaluates to `true`, the trace is valid; otherwise, it is invalid.

In diagrams, the constraint attached to a state invariant is shown as a text in curly braces on the lifeline, as depicted in Figure 15-11a. Alternatively, a state invariant can be shown as a note associated with an occurrence specification, or in an oval state symbol, as depicted in Figure 15-11b. The state symbol then represents the constraint that inspects the state of the object represented by the lifeline. This could be the internal state of the object's state machine, or some external abstract state based on a "black-box" view of the lifeline.

A more complex form of fragment is a *combined fragment*, which defines an expression of one or more nested interaction fragments that are its operands. With combined fragments, a (possibly unlimited) number of traces can be specified concisely. Combined fragments are described in the discussions that follow.

### Section Summary

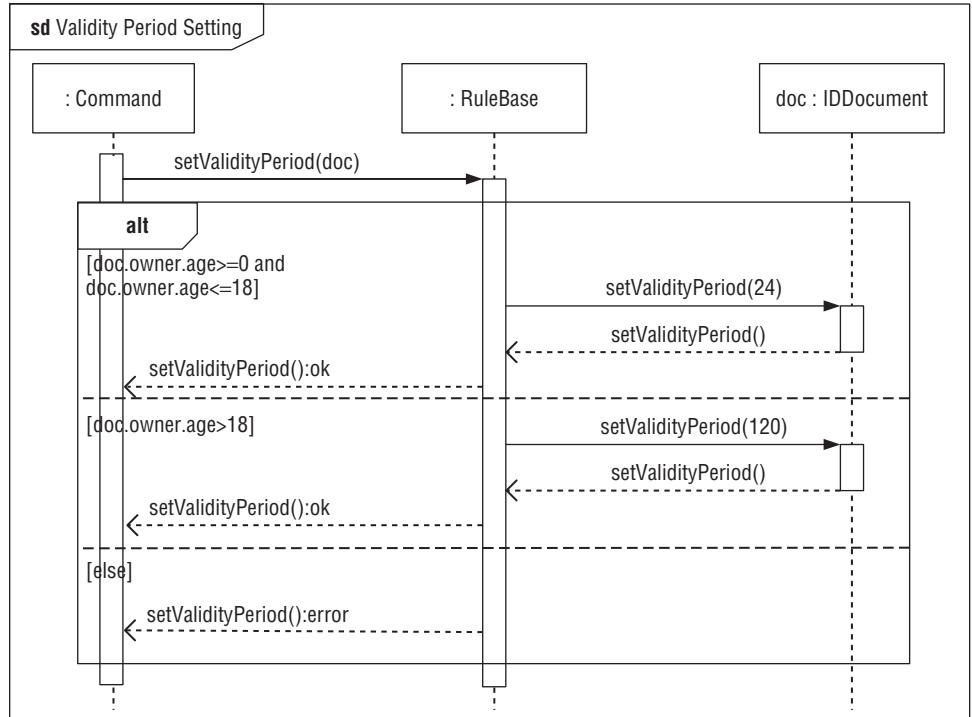
- ❑ A *state invariant* is a constraint placed on one lifeline that defines a condition or restriction on the participants of the interaction that has to hold at a certain moment at run-time. All traces in which the state invariant evaluates to `false` at the time the interaction reaches its point are considered invalid.
- ❑ A *combined fragment* defines an operator and one or more nested operand interaction fragments.

## Alternatives and Options

The combined fragment with the `alt` operator designates a choice of one of its operand fragments. As shown in Figure 15-12, one to an arbitrary number of operand fragments of the combined fragment are separated by horizontal dashed lines, while the entire combined fragment is framed in a rectangle having a pentagon with the keyword `alt` in the upper-left corner. Each of the operand fragments can have an explicit or implicit guard expression that evaluates to a Boolean value. The guard is shown in square brackets at the beginning of the operand fragment. An implicit `true` guard is assumed if the operand has no explicit guard. The keyword `else` designates a guard that is true when none of the remaining guards are true. If the guard expression should be evaluated in the context of a lifeline, it must overlap that lifeline. In some cases, it may be evaluated on multiple lifelines, in which case it is sufficient for it to appear on one of them.

In brief, the `alt` combined fragment represents a selection of one of the alternatives, depending on the Boolean guards of the alternatives. In each trace defined by the enclosing interaction, the guards of the operand fragments are interpreted as being evaluated at the point when the trace reaches the combined fragment. Then, only one of the operand fragments contributes to the resulting trace of the enclosing interaction fragment. This is the operand whose guard evaluates to `true` at this point in the interaction.

If none of the operands has a guard that evaluates to `true`, none of the operands contributes to the resulting trace.



**Figure 15-12:** The `alt` combined fragment is a choice of behavior. No more than one of the operands contributes to the resulting trace, depending on the results of the guard expressions of the operands.

A combined fragment with the `opt` operator is a special case of the `alt` fragment. It can have only one operand with a guard. It designates a choice where either the operand is performed or ignored. In other words, its result is the trace defined by its operand if its guard evaluates to `true`, or an empty trace otherwise. Consequently, an `opt` fragment is semantically equivalent to an `alt` fragment having one operand equal to the operand of the `opt` fragment, and another empty `else` operand.

A combined fragment with the `break` operator has only one operand fragment with a guard, and should always cover all lifelines of its enclosing interaction fragment. When the guard of the `break` operand evaluates to `true`, the operand is chosen, and the rest of the enclosing interaction fragment is ignored. When the guard evaluates to `false`, the operand fragment is ignored, and the rest of the enclosing interaction fragment is chosen. This kind of fragment can be used to model exceptional cases and their handling, for example. In the case where an exceptional situation is encountered in the enclosing interaction (modeled by the guard of the `break` fragment), it is handled as modeled in the `break` fragment, while the main interaction is aborted.

### Section Summary

- ❑ The combined fragment with the `alt` operator represents a choice of one of its operand fragments. Only one of the operand fragments contributes to the resulting trace defined by the enclosing interaction fragment. This is the operand whose guard evaluates to `true` at this point in the interaction.
- ❑ The combined fragment with the `opt` operator is semantically equivalent to an `alt` fragment having one operand equal to the operand of the `opt` fragment, and another empty `else` operand.
- ❑ A combined fragment with the `break` operator designates that the operand is performed and the remainder of the enclosing interaction fragment is ignored, when the operand's guard evaluates to `true`.

### Parallelizing and Sequencing

A combined fragment with the `par` operator represents a parallel merge of the traces of its operands. The traces of a `par` combined fragment are obtained by interleaving the traces defined by its operands in any way that preserves the ordering imposed by each operand. For example, a parallel merge of the first two operands of the `par` combined fragment in Figure 15-13a results in a set of all traces where the occurrences are ordered in any order, provided that the sending of the message `send(msg1)` is before its reception, which is before the sending of the message `accept(msg1)`, which is before its reception, which is before the sending of the message `ack(msg1)`, and so on. For example, a legal trace could be `send(msg1), accept(msg1), send(msg2), ack(msg1), accept(msg2), ack(msg2), ack(msg1)`.<sup>4</sup>

Very often, a parallel combined fragment is limited to the occurrences or other nested fragments on a single lifeline, indicating that their order is insignificant. There is a notational shorthand for such common cases of parallel combined fragments in the form of a *coregion* placed on one lifeline, as shown in Figure 15-13b. A coregion placed on a lifeline designates that all the fragments or occurrence specifications that are directly contained by the coregion are considered separate operands of a parallel combined fragment. For example, the coregion placed on the lifeline `c` in Figure 15-13b specifies that the `send(msg1)` and `send(msg2)` messages can be received in any order, although the former was sent prior to the latter. In other words, this interaction illustrates that the messages can be received in the same order they were sent, or they can overtake each other.

However, sometimes it is necessary to specify that the traces defined by a certain fragment of interaction should not be interleaved with the traces of other fragments, even though such interleaving may be implied from the enclosing combined fragments. Such interaction fragments that should be treated as isolated are called *critical regions*.

For example, the interaction in Figure 15-13a documents that when the Channel receives an urgent message through its operation `sendUrgent`, it must be sure to forward that message to the Recipient, and to reply with an acknowledgment to the Sender once it receives an acknowledgment from the Recipient,

---

<sup>4</sup>Every item in this list represents an ordered pair of the send and receive event occurrences of the corresponding operation call message shown in the diagram.

before (or instead of) doing anything else. The same holds true for the Recipient. For that purpose, a combined fragment with the **critical** operator has been introduced.

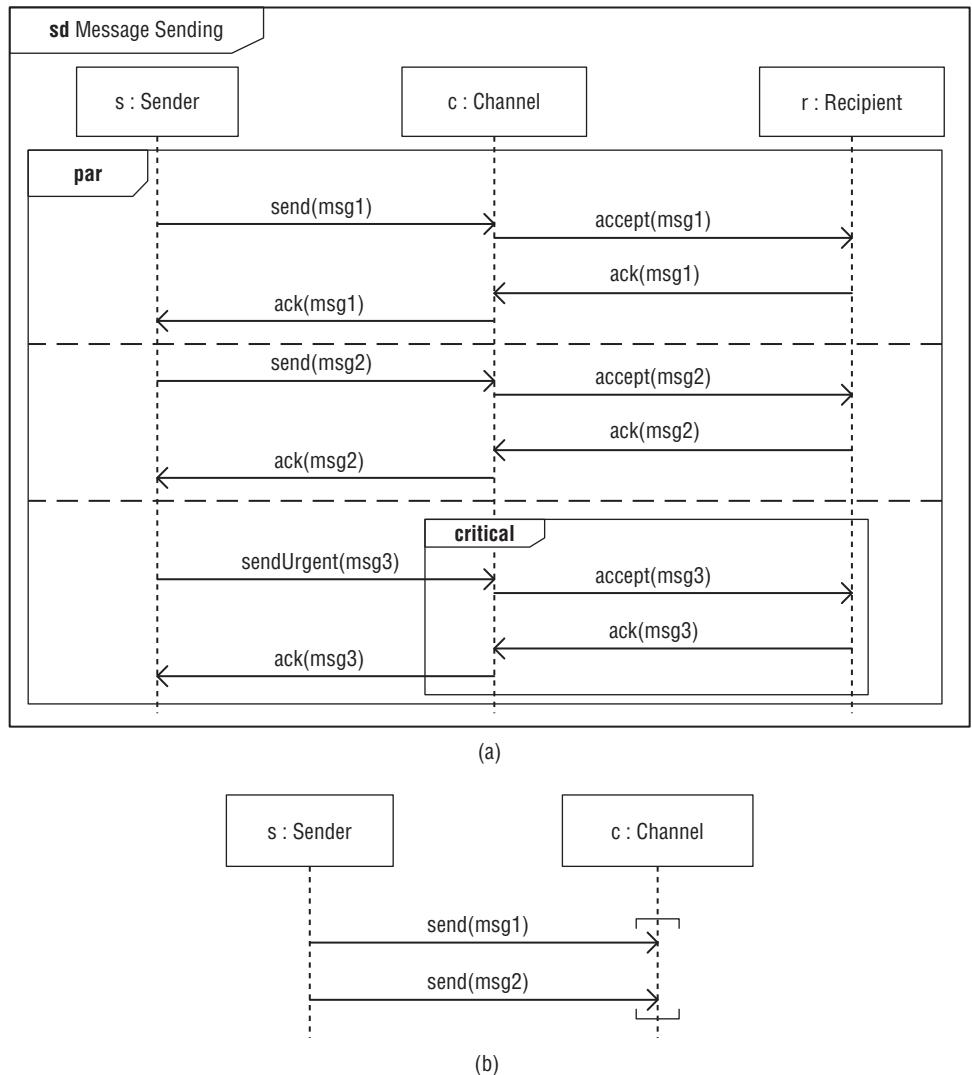


Figure 15-13: a) The **par** and **critical** combined fragments. b) A coregion

The **par** combined fragment allows interleaving of occurrences from different operand fragments even when they are on the same lifeline, and, thus, redefines the basic ordering rule. Sometimes exactly the opposite is needed — to restrict the ordering of occurrence specifications from different lifelines, even though they would be unrelated by the default ordering rules. For that purpose, combined fragments for *strict sequencing* can be used.

A combined fragment with the **strict** operator represents a strict ordering of the traces defined by the operands. That is, a trace defined by a subsequent operand fragment can come only after a complete trace

## Part III: Concepts

defined by the preceding operand fragment, and cannot be interwoven with it. For the example shown in Figure 15-14<sup>5</sup>, the only valid trace is  $s1, r1, s2, r2$ . Note that the default semantics, if there were no `strict` combined fragment in this interaction, would also allow the trace  $s1, s2, r1, r2$ .

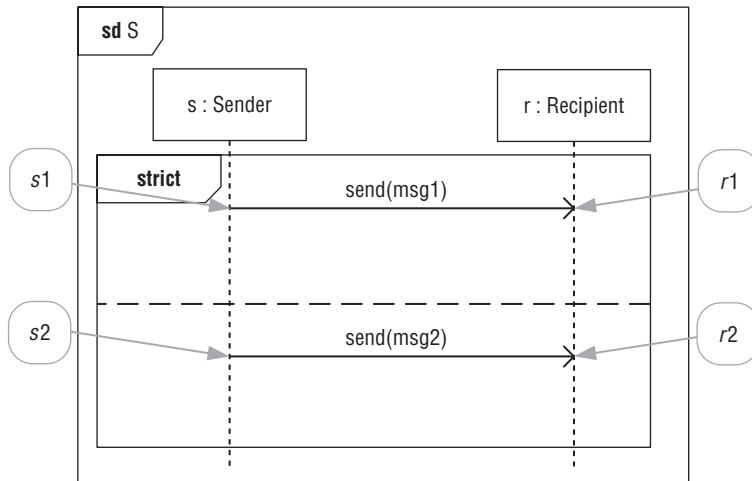


Figure 15-14: The `strict` combined fragment. The rounded gray callouts are annotations that provide references to the occurrence specifications and are not part of the UML notation.

### Section Summary

- ❑ The traces of a `par` combined fragment are obtained by interleaving the traces defined by its operands in any way that preserves the ordering imposed by each operand.
- ❑ A combined fragment with the `critical` operator represents a *critical region*, meaning that the traces of the region cannot be interleaved by other occurrence specifications on those lifelines covered by the region.
- ❑ A combined fragment with the `strict` operator represents a strict ordering of the traces defined by the operands.

## Loops

A combined fragment with the `loop` operator represents a repetition of its sole operand, as shown in Figure 15-7. In the traces implied from the fragment, the traces implied from its sole operand will be repeated a number of times. The operand may include expressions for the lower and the upper number

<sup>5</sup>The rounded callouts, along with their arrowhead lines, in this figure are not part of the UML notation and serve as annotations only.

of iterations of the loop, as well as a Boolean guard expression. The loop will iterate at least the lower number of times, and at most the upper number of times. The loop will terminate after the lower number of iterations has executed and the Boolean guard expression evaluates to `false`, but not later than the upper number of iterations has executed.

The semantics of repetition are that the operand is iteratively appended to the interaction the given number of times, as if it has been explicitly modeled in the interaction. This is not equivalent to iterative strict sequencing of the loop operand because the event occurrences from different iterations on different lifelines may interleave as long as the ordering of the occurrences from different iterations on the same lifeline is satisfied.

The lower and the upper bound of the loop are written in the pentagon of the loop frame after the `loop` keyword in the following syntax:

```
loop_spec ::= loop
loop_spec ::= loop (minint)
loop_spec ::= loop (minint , maxint)
```

The `maxint` can be an expression or an asterisk (\*), which means an infinite upper bound. If only `minint` is present, this means `minint=maxint`. If there is no `minint` and `maxint`, this means `minint=0` and `maxint` is infinite, so the loop is executed while the guard expression evaluates to `true`. For example:

- ❑ `loop (3)` — Executes exactly three times, regardless of the Boolean guard
- ❑ `loop (3,5)` — Executes minimum three times and, then, while the guard evaluates to `true`, at most five times
- ❑ `loop (3, *)` — Executes minimum three times, and, then, while the guard evaluates to `true`
- ❑ `loop (0, *)` — Executes while the guard evaluates to `true`
- ❑ `loop` — Executes while the guard evaluates to `true`

The Boolean guard expression is depicted as a constraint before the first occurrence specification in the loop operand, as shown in Figure 15-7. The placement of the guard is restricted in the same way as for the `alt` operand case — that is, it must overlap the appropriate lifeline. If there is no Boolean expression, the guard is taken to evaluate to `true`.

In OOIS UML, the iteration can also be specified using the iteration constructs of this language (for example, `forEach`), as already shown in Figure 15-7. In that case, the iteration variable refers to a lifeline in the interaction.

### Section Summary

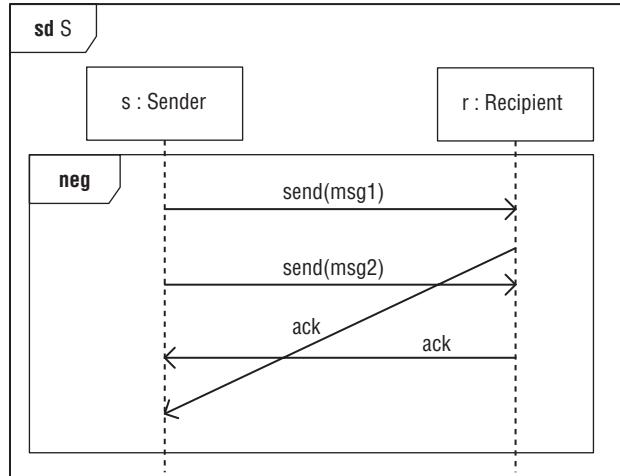
- ❑ A combined fragment with the `loop` operator represents a repetition of its sole operand. The loop will iterate, the lower number of times and, at most, the upper number of times. After the minimum number of iterations has executed and the Boolean expression is `false`, the loop will terminate.

## Part III: Concepts

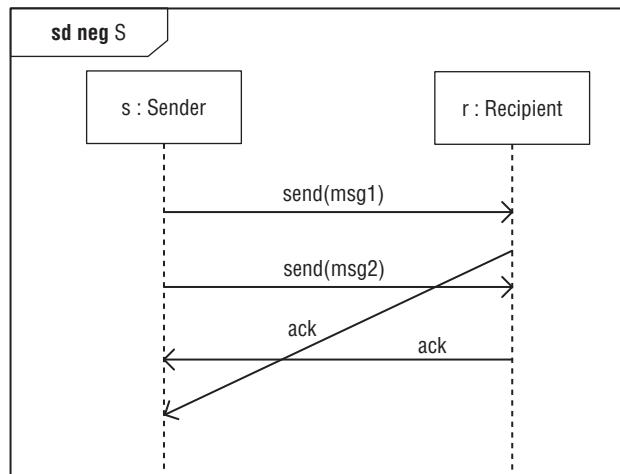
### Negative Traces

A combined fragment with the operator neg designates invalid traces, that is, the traces that should not happen during execution. The set of invalid traces defined by a neg combined fragment is equal to the set of traces defined by its sole operand.

Figure 15-15a shows an example of a neg combined fragment. It illustrates that the reception of an ack message should not be delayed so long as it comes after the reception of another ack for a later send.



(a)



(b)

Figure 15-15: The neg combined fragment

Figure 15-15b shows an alternative diagram for the same interaction. In general, more than one operator may be shown in the pentagon descriptor. This is shorthand for nesting combined fragments. This means, for example, that `sd strict` in the pentagon descriptor is the same as a `strict` combined fragment nested into an `sd` interaction, while an `sd neg` is the same as a `neg` fragment nested in an `sd`.

### Section Summary

- ❑ A combined fragment with the operator `neg` defines invalid rather than valid traces.

## Interaction References

Similar to all other behaviors, interactions can be specified for reuse at different places. This follows the principle of procedural decomposition and localization of common behavioral parts to be reused in different contexts.

For example, the interaction shown in Figure 15-16a<sup>6</sup> represents a general case of distributing a Message to a Group Recipient. It is designed for reuse. Hence, it has formal arguments, similar to operations. In general, interactions can have formal arguments that have similar meaning and are shown in the title of the interaction using the same syntax as for operations. In addition, this interaction has its *formal gates*, which are connection points for the messages that lead from and to the interaction, as shown in Figure 15-16. Gates are named elements, and are represented in diagrams as end points of messages placed at the interaction frame.

An interaction defined in that way can then be reused in other interactions by *referencing*. Figure 15-16b shows another interaction that illustrates the implementation of the execution of the `CmdNotifyGroup` command. In that interaction, the `Notify Group` interaction is used (that is, referred to) from the `ref` interaction fragment. The use provides the actual arguments for the referenced interaction, as well as the actual gates. The actual gates are points to which the messages from the enclosing interaction are anchored. By that mechanism, the same `Notify Group` interaction can be reused in other contexts without actual copying. In fact, an interaction use allows multiple interactions to reference an interaction that represents a common subpart of their specifications.

In general, an interaction use is semantically equivalent to copying the contents of the referenced interaction at the place of the use, with formal parameters substituted with actual arguments and formal gates connected with the actual ones.

<sup>6</sup>The rounded callouts, along with their arrowhead lines, in this figure are not part of the UML notation and serve as annotations only.

## Part III: Concepts

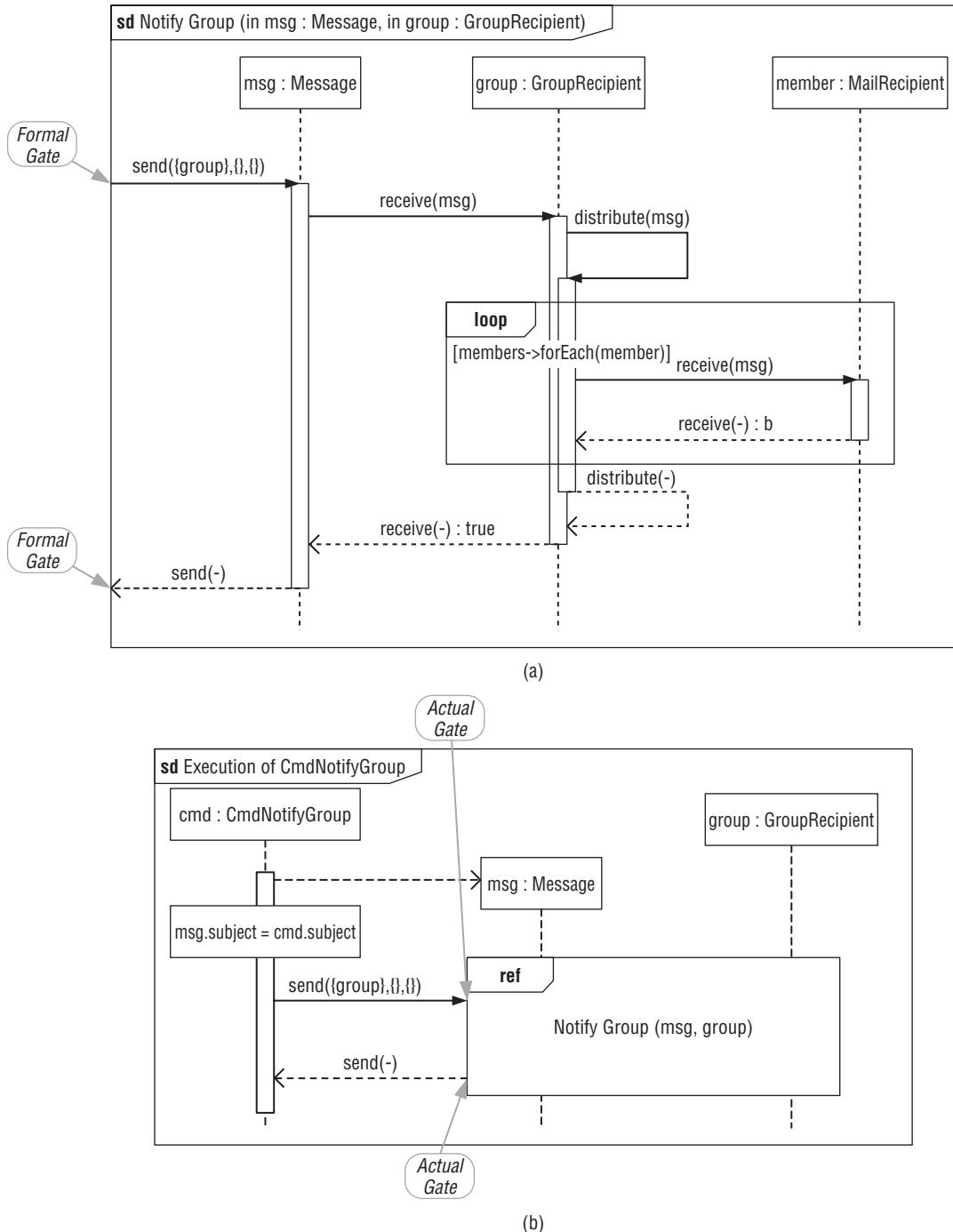


Figure 15-16: Interaction referencing. (a) The definition of an interaction with formal parameters and gates. (b) A use of interaction with the provided actual arguments and gates. The rounded grayed callouts are annotations that provide references to the elements and are not part of the UML notation.

The interaction use (as a fragment) must cover all lifelines of the enclosing interaction that appear within the referenced interaction. The actual arguments of the interaction use should match the formal parameters of the referenced interaction. They can be constant expressions, parameters of the enclosing interaction, or properties of the classifier owning the enclosing interaction. The syntax for the actual arguments of an interaction use is the same as for messages. The actual gates of an interaction use must match the formal gates of the referenced interaction. For drawing convenience, there is a convention (which is not formally part of the UML notation but is widely adopted) whereby lifelines that do not participate in an interaction use can be shown as crossing over the interaction use, to show that they are not involved in it.

### Section Summary

- As a behavior, an interaction can have its formal parameters, similar to operations.
- An interaction can have its *formal gates*. Gates are connection points of the messages leading from and to the interaction.
- An interaction can be reused in other interactions by referencing. An interaction use is a fragment that refers to another interaction, providing actual arguments and connecting gates. It is semantically equivalent to copying the contents of the referenced interaction at the point of the use.



# 16

## Commands, Presentation, and Architecture

Chapter 6 provided an introductory overview of the core concepts and mechanisms related to the customizable presentation and interaction in OOIS UML. This chapter provides more detailed descriptions of these concepts and mechanisms available for GUI development in OOIS UML.

One of the key concepts is that of *commands*. Commands are classes whose objects represent requests for services from the system, issued interactively from the GUI or from outer systems interoperating with the given system. This chapter describes a command as a modeling concept and its features. It also describes the built-in commands of OOIS UML.

In addition, this chapter discusses the architecture of the application's presentation layer, as well as the entire application in different deployment strategies.

### Commands

A *command* is a class whose objects represent requests for a service from the system, issued interactively from the GUI or from outer systems interoperating with the given system. When a command is issued, an instance of the command class is created and its `execute` operation is invoked. The behavior of the command is thus specified within the method of this operation. It can invoke an operation of an object, for example, or it can incorporate arbitrary code in the detail-level language.

The *generic commands* (also known as *built-in commands*) are commands that embody certain UML actions upon the object space. They are accessible from the application because they are part of the OOIS UML model library.

*Domain-specific commands* represent entry points to the implementation of the specific functionality of the system, not directly supported by the execution environment and generic commands. They are introduced into the model by the modeler.

### Class Command

The abstract class `Command`, which is part of the OOIS UML model library, is a generalization of all generic and domain-specific commands. It gathers the common features and mechanisms of all commands, including pins, execution, and prototyping, as described in this section.

Figure 16-1 shows the `Command` class and its main features. This class depends on a few other types from the OOIS UML model library, as you can see from the definition of this class. These types generalize modeling and run-time concepts so that commands can work with any of their kind. An `Element` is an abstract generalization of all kinds of model and run-time elements that are supported by the implementation and that can be represented in the GUI as items (classes, properties, objects, slots, and so on). It allows application of commands to both model and run-time elements as parameters (input pins). For example, the generic command `Create Object` is applicable to a class (which is a model element), while the command `Destroy Object` is applicable to an object of a class (which is a run-time element). The type `Slot` abstracts an instance of a property (attribute or association end). An instance of the type `Slot` refers to an instance of a property inside an object of a class. Through operations of `Slot`, a command can access the values of the referred property instance.

| <i>Command</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| + name : Text = "New Command"<br>+ description : Text<br>+ isPrototype : Boolean = false<br><br>+ getOutputPins() : Slot[*]<br>+ getInputPins() : Slot[*]<br>+ getInputPins(e : Element) : Slot[*]<br>+ isApplicableTo(e : Element) : Boolean<br>+ isApplicableTo(e1 : Element, e2 : Element) : Boolean<br>+ isApplicableToAll(e : Element[*]) : Boolean<br>+ getFavoritePinForFirstParam(e : Element) : Slot<br>+ getFavoritePinForSecondParam(e : Element) : Slot<br>+ setInputPin(e : Element)<br>+ setInputPins(e1 : Element, e2 : Element)<br>+ checkInputPinsCoverage() : Boolean<br>+ clone() : Command<br>+ handle()<br># execute()<br>+ applyTo(e : Element)<br>+ applyTo(e1 : Element, e2 : Element)<br>+ applyToAll(e : Element[*]) |

**Figure 16-1:** The OOIS UML built-in class `Command` and its main features. The `Element` type is a generalization of all classes and data types, as well as model elements accessible through reflection.

The first subset of features of the `Command` class is related to the identification of a command by the system, as well as by users and modelers. The `name` attribute stores a short name of a command, while the `description` attribute may store a more explanatory description of a command. These attributes are usually redefined so that they have specific default values adequate for a concrete command. The generic GUI relies on these attributes to display commands, especially during dynamic customization of behavior.

The second important subset of features is related to the capability to have prototypes of commands, as predefined, preconfigured objects of certain specialized commands, and their cloning in a generic way, according to the Prototype design pattern [Gamma, 1995].

To be able to distinguish the preconfigured prototypes of certain commands from other instances of commands that might have been persistently stored in the system as logs of execution, the system (for example, the generic GUI) relies on the value of the `isPrototype` attribute. The preconfigured prototypes of commands have `true` as the value of this attribute. When a clone of a prototype is created over the `clone` operation, it gets the default value `false` of this attribute. Depending on the needs and kind of command, such a clone can be deleted from the system after its execution, or retained persistently in a log of execution. Because prototypes and retained clones of a command are just ordinary objects of the same class, they can be distinguished, if necessary, over the value of the `isPrototype` attribute.

A command can have *input pins*, which are properties of the command tagged with `<<inputPin>>`. Input pins serve as input parameters of the command provided from the GUI or programmatically. The `getInputPins` operation returns a collection of all slots that are instances of input pins of the command. The `getInputPins(e:Element)` operation returns a collection of slots that are instances of input pins, and that can accept the given model or run-time element.

Matching of a provided element to an input pin is done by the operation `getFavoritePinForFirstParam`. It returns the input pin that can accept the given element and is the “favorite” for the element as the first parameter of the command. From all input pins returned by `getInputPins` that can accept the given element, it returns the first encountered one that has the name (case-insensitive, in this precedence) “this” or “self,” then “source” or “src,” then “input” or “input1,” and then any other name.

This similarly holds true for `getFavoritePinForSecondParam`. It returns the input pin that can accept the given element and is the “favorite” for the element being the second parameter of the command. From all input pins returned by `getInputPins` that can accept the given element, it returns the first encountered one that has the name (case-insensitive, in this precedence) “target” or “tar,” then “destination” or “dest” or “dst,” then “input2,” then any other name, but is not the favorite pin for the first parameter.

Using these two operations, the generic GUI can supply the parameters to a configured command (for example, at drag-and-drop). For that reason, domain-specific commands should conform to the naming rules of input pins for the source and target parameters, if modelers want them to be easily recognized by the generic GUI on a drag-and-drop configuration (that is, if the modeler wants them to be applicable upon drag-and-drop).

Although input pins are properties of command classes and thus, can be manipulated like all other properties (programmatically or interactively), there is a slight difference between properties that are input pins and those that are not. When the generic GUI handles a user’s action and is about to execute a command, it takes into consideration only the properties that are input pins in order to pass the values that are parameters of the user’s action. For that reason, a general rule for modeling properties as input pins is to do so if the value of that property should be provided dynamically, at the very moment of command issuance, as the direct parameter of that action (for example, as the item clicked on, or as the two items taking part in a drag-and-drop).

On the other hand, the other properties that also influence the execution of a command but are not input pins are usually those whose values are preconfigured in command prototypes so that the command is specialized or configured for a certain task. In other words, the values of the properties that are not input

## Part III: Concepts

---

pins are usually already provided in the command prototypes, while the values of the input pins are provided at the moment of executing a command.

For example, the built-in command `CmdCreateObjectAndLinkToObject` has a property named `className` that is not an input pin and that determines the class whose object will be created. The value of this property is usually preconfigured for a specific command prototype. The same command, however, has an input pin that determines the object to which the newly created one will be linked. That object is usually provided as the parameter of each particular execution of the command. Of course, if a command is issued programmatically, all properties can be treated the same way, so that even the properties that are not input pins can be set at the moment of command dispatching.

When a command is about to be executed, the `checkInput Pins Coverage` operation is called to check whether all necessary input pins have their values as needed by the command. The default implementation of this operation simply checks whether each input pin has at least one value. However, this operation is often overridden in specific commands. For example, a command that operates on a slot, like the command `CmdCreateObjectAndLinkToObject`, which links a newly created object to the provided slot that is an association end instance, can determine the slot in one of two available ways:

- ❑ The slot can be provided to an input pin of the command, so that the user can issue the command on one slot as its direct dynamic parameter.
- ❑ The slot can also be determined by the name of the property specified in a textual attribute of the command, and by the object on whose slot of that property the command will operate, and which is provided to an input pin of the command. That way, the user can issue the command on an object as its dynamic parameter.

In other words, the command `CmdCreateObjectAndLinkToObject` will have two input pins, one of type `AssocEndInstance` and one of type `ObjectOfClass`. But these pins will be used alternatively. Only one of them will need a value. For that reason, that command class (as with many others) will override the `checkInput Pins Coverage` operation.

The two operations `isApplicableTo` and `isApplicableToAll` are also used by the generic GUI to determine whether a selected command is (or which commands are) applicable to the given element, two elements (as the source and the target parameters), and to each of the elements in the given collection, respectively.

When a command prototype is applied to one element (`applyTo(e:Element)`) or to two elements (`applyTo(e1:Element, e2:Element)`), the following is performed:

- ❑ If the command is not applicable to the given element(s), nothing is done.
- ❑ The command is cloned using the operation `clone`.
- ❑ The input pin(s) of the clone that accepts (accept) the given element(s) is (are) set to the value(s) using the operation `setInputPin` (`setInputPins`). The operations `setInputPin` and `setInputPins` rely on the operations `getFavoritePinForFirstParam` and `getFavoritePinForSecondParam` to match the given values to the input pins.
- ❑ The clone is dispatched for execution by invoking its `handle` operation.

A command prototype can also be applied to each of the elements in the given collection over the operation `applyToAll`, when the described procedure is applied to each element in the collection.

Handling of command execution is the responsibility of the operation handle. This operation should be called whenever a command instance is to be dispatched for execution. The method of this operation is not, however, responsible for the specific execution of the command. Instead, it is a wrapper around the execution that provides much more of the generic behavior common for all commands. In other words, the method of handle provides a fixed set of ordered steps of execution of every command, only one of them being specific for a specialized command. That very specific step is embodied in the protected operation execute that should be overridden in subclasses. That way, the handle operation provides a skeleton of the common algorithm of execution of all commands, while some of the steps of that execution (provided by other elementary operations not shown here) can (or cannot be) modified in specializing classes. This is the application of the *Template Method* design pattern [Gamma, 1995].

Depending on the concrete implementation of OOIS UML, the needs of the concrete system, architecture, and platform, the set of steps embodied in handle may vary, but may in general include the following:

- ❑ **Transaction mechanism** — The entire following processing can be enclosed in a transaction, to support isolation and backward error recovery.
- ❑ **Checking of complete coverage of all input pins by calling checkInput PinsCoverage** — If any of the necessary values on the input pins are missing, the GUI can ask the user to provide them.
- ❑ **Processing and checking of parameters** — For example, the arguments may be provided from outer or remote systems as serialized streams of characters (for example, in XML), so that they have to be unmarshaled prior to command execution. Additionally, a command may need to check the semantic correctness of the parameters before it accepts them for execution.
- ❑ **Authorization (access right checking)** — This determines whether the user who issued the request has the right to do it.
- ❑ **Explicit locking** — Certain parts of the object space may have to be explicitly locked, if necessary (because they are not implicitly locked by the actions, but have to be according to the specific logic of the command).
- ❑ **Checking of other specific preconditions, if any** — A specific command can have domain-specific preconditions for its execution. If a precondition is not satisfied, the command may refuse to execute or undertake other alternative activities.
- ❑ **Performing other preparation activities, if any** — A specific command can perform other activities prior to its “core” execution, if necessary.
- ❑ **Concrete execution of the command** — This is the call of the execute operation that does the “core” execution of the command.
- ❑ **Performing other post actions** — This might, for example, involve clean-up or other activities that a specific command might need to do after its “core” execution.
- ❑ **Checking other specific postconditions, if any** — A specific command can have domain-specific postconditions that its execution must meet upon completion. These serve as a kind of acceptance criteria for the proper completion of the command. If a postcondition is not satisfied, an exception can be raised, and the transaction can be rolled back.
- ❑ **Logging (persistence) of the command** — Logging may simply mean not deleting the command object after its execution, so that it remains in the persistent storage. It may, however, mean other activities, such as creating separate log objects, serialization and writing to log files, and so on.
- ❑ **Unlocking of explicitly locked objects, if necessary** — If any part of the object space has been locked explicitly in one of the first steps, the locks can be released now.

## Part III: Concepts

---

- ❑ **Exception handling** — If any of these steps failed, because of preconditions, postconditions, transaction, or locking failures, the raised exception can be caught and possibly linked to the command object as an evidence of failure for later inspection, and the transaction can be rolled back.

Most of these steps are implemented in separate protected operations that may provide some default (often empty) behavior, which can be overridden in specializing classes. Such operations are usually referred to as the *hook* methods. By enclosing the entire common algorithm in the `handle` template method, the developer is spared from having to take care of these maintenance tasks separately for each specific command (possibly repeatedly) and from making the mistake of omitting some of them. In the simplest and most common case, the specialized command class just needs to override the `execute` operation.

Finally, a command may have its *output* pins that serve as output or result parameters of the command execution. For example, a command that creates an object or an object structure may put a reference to one of these objects on its output pin for later use from the GUI or other parts of the system. Output pins are properties tagged with `<outputPin>`. The operation `getOutputPins` returns the collection of all output pins. As one of its final steps of execution, the method of `execute` may set the output pins to the appropriate values.

### Section Summary

- ❑ The abstract class `Command`, which is part of the OOIS UML model library, is a generalization of all generic and domain-specific commands. It gathers the common features and mechanisms of all commands:
  - ❑ Identification of commands by the system and the users
  - ❑ Prototyping and cloning
  - ❑ Input pins and features for accessing them (including matching of provided values to pins)
  - ❑ Output pins and features for accessing them
  - ❑ Applying a command to one or two given values, or to each value in the provided collection
  - ❑ Handling and execution of the command

## Built-In Commands

The OOIS UML model library contains a package with a set of predefined commands that embody some fundamental UML structural actions. By means of those actions, the GUI or the back-end application can perform some basic structural manipulations, taking full advantage of the generic command execution features (such as isolation, authorization, fault tolerance, logging, and so on).

The built-in commands will be described in separate discussions that follow, classified according to their kind and scope of work.

Commands that work with one or more slots (instances of attributes or association ends) as their parameters offer two alternatives for specifying each slot:

- ❑ **By directly referring to the slot over an input pin** — The input pin is usually named `input`, `source`, or `target`, and is of type `AttrValue`, or `AssocEndInstance`, or `Slot` (the generalization of the two).
- ❑ **By indirectly specifying the slot over its owner object and the name of its property (attribute or association end)** — The owner object is provided to an input pin that is usually named `srcObj` or `tarObj` and is of type `ObjectOfClass`. The fully qualified name of the property is provided in another attribute of the command, usually named `propName`, `attrName`, `assocEndName`, or something similar.

Consequently, such a command can be applied either directly to a slot, or to an object, in which case it will affect the specified slot of that object. For example, the `CmdCreateLink` command can be applied (for example, by a drag-and-drop) to a source object supplied to its input pin `source` and to either a target association end instance supplied to its input pin `target`, or to a target object supplied to its input pin `tarObj`. By overriding the operation `checkInputPinsCoverage`, such commands ensure that they can be executed in any of these cases (that is, even when some of the input pins do not have values). This mechanism allows reusability of the same command class that basically does the same work for different combinations of input parameters.

As an additional opportunity to customize a command prototype for specific properties or types of objects they work on, such commands have attributes that specify the allowed type of the object on whose slot they operate. Such an attribute is usually named `type`, `srcType`, or `tarType`, and specifies the fully qualified name of that type. This attribute is optional, and does not have to have a value (it is of multiplicity `0..1`). If it has a non-empty textual value, the command will be applicable only to the slot owned by an object of that type (in the first case), or on the object of that type (in the second case).

If such a command is issued in the first way (on a slot as its input pin), the attribute specifying the name of the property may also have a restrictive purpose. If that attribute has a non-empty value, the command will be applicable only to the slot that is an instance of the property specified in that attribute by the fully qualified name. Note that the property does not need to be owned by the same classifier specified in the attribute `type` (or similar) — they can be different classifiers, usually from the same generalization/specialization hierarchy.

### **Object Commands**

The sections that follow examine object commands and their properties.

#### **Class CmdCreateObject**

`CmdCreateObject` creates a new object of the class supplied to its input pin.

Properties of this command are shown in the following table.

| Property                                                    | Description                            |
|-------------------------------------------------------------|----------------------------------------|
| <code>&lt;inputPin&gt; input : Class[0..1]</code>           | The class whose object will be created |
| <code>&lt;outputPin&gt; output : ObjectOfClass[0..1]</code> | The created object                     |

## Part III: Concepts

---

### Class CmdCreateObjectOfClass

CmdCreateObjectOfClass creates a new object of the class whose fully qualified name is given in its attribute that is not an input pin. A prototype can be configured for a specific class and does not need any value on its input pins to be executed (because it does not have any). This command can be attached to buttons or other GUI controls that are enabled independently of the selection of any element in the GUI. For example, there can be a button or a hyperlink reading “Create a new Department” or “Create a new Person” configured to issue such a command.

Properties of this command are shown in the following table.

| Property                                 | Description                                                        |
|------------------------------------------|--------------------------------------------------------------------|
| className : Text                         | The fully qualified name of the class whose object will be created |
| «outputPin» output : ObjectOfClass[0..1] | The created object                                                 |

### Class CmdDestroyObject

CmdDestroyObject destroys the object supplied to its input pin. If its attribute `type` has a non-empty textual value, the command can be applied only to objects of the type with the fully qualified name given in that attribute.

Properties of this command are shown in the following table.

| Property                               | Description                                                                                     |
|----------------------------------------|-------------------------------------------------------------------------------------------------|
| «inputPin» input : ObjectOfClass[0..1] | The object to be destroyed                                                                      |
| type : Text[0..1]                      | The fully qualified name of the type of the object accepted at the input pin <code>input</code> |

## Slot Commands

Commands of this category generally work on slots, be they instances of attributes or association ends. Note that for all of the following discussions, the `Value` type is a generalization of objects of classes and instances of data types. All these commands rely on the corresponding actions on slots.

### Class CmdClearSlot

CmdClearSlot clears the given slot (that is, removes all values from it).

Properties of this command are shown in the following table.

| Property                                | Description                                                                    |
|-----------------------------------------|--------------------------------------------------------------------------------|
| «inputPin» input : Slot[0..1]           | The slot to be cleared (all values removed)                                    |
| «inputPin» target : ObjectOfClass[0..1] | The target object whose slot is to be cleared                                  |
| type : Text[0..1]                       | The fully qualified name of the type of the object whose slot is to be cleared |
| propName : Text[0..1]                   | The fully qualified name of the property of the slot to be cleared             |

### Class CmdRemoveValueFromSlot

CmdRemoveValueFromSlot can remove one or more occurrences of a value from the given slot:

- If the value of the command's attribute `removeFirst` is `true` and the slot's property is ordered, only the first value from the slot will be removed. The value in the input pin `source` is ignored and not needed in that case.
- Otherwise, if the value of the command's attribute `removeLast` is `true` and the slot's property is ordered, only the last value from the slot will be removed. The value in the input pin `source` is ignored and not needed in that case.
- Otherwise, if the value of the command's attribute `removeAt` is non-negative and the slot's property is ordered, only the value at that position from the slot will be removed. The value in the input pin `source` is ignored and not needed in that case.
- Otherwise, the value given in the input pin `source` will be removed. In the case where the slot's property is non-unique, if `removeAll` is `true`, all occurrences of that value will be removed. Otherwise, only the first one will be removed.

Properties of this command are shown in the following table.

| Property                                   | Description                                                                                                      |
|--------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| «inputPin» target : Slot[0..1]             | The slot from which a value is to be removed                                                                     |
| «inputPin» tarObj : ObjectOfClass[0..1]    | The target object whose slot is to be changed                                                                    |
| type : Text[0..1]                          | The fully qualified name of the type of the object whose slot is to be changed                                   |
| propName : Text[0..1]                      | The fully qualified name of the property of the slot to be changed                                               |
| «inputPin» source : Value[0..1]            | The value to be removed from the slot                                                                            |
| removeAll : Boolean = <code>false</code>   | If <code>true</code> , all occurrences of the given value in the slot (in case it is non-unique) will be removed |
| removeFirst : Boolean = <code>false</code> | If <code>true</code> , the first value in the slot (in case it is ordered) will be removed                       |

*Continued*

## Part III: Concepts

---

(continued)

| Property                     | Description                                                                                                   |
|------------------------------|---------------------------------------------------------------------------------------------------------------|
| removeLast : Boolean = false | If true, the last value in the slot (in case it is ordered) will be removed                                   |
| removeAt : Integer = -1      | If greater than or equal to 0, the value in the slot (in case it is ordered) at this position will be removed |

### Class CmdSetSlot

CmdSetSlot sets the new value of the given slot.

Properties of this command are shown in the following table.

| Property                                 | Description                                                                |
|------------------------------------------|----------------------------------------------------------------------------|
| «inputPin» source : Value[0..*]{ordered} | The new value that is to be copied to the target slot                      |
| «inputPin» target : Slot[0..1]           | The target slot that is to be set                                          |
| «inputPin» tarObj : ObjectOfClass[0..1]  | The target object whose slot is to be set                                  |
| type : Text[0..1]                        | The fully qualified name of the type of the object whose slot is to be set |
| propName : Text[0..1]                    | The fully qualified name of the property of the slot to be set             |

### Class CmdAddValueToSlot

CmdAddValueToSlot adds the value given in the input pin source to the target slot. If the slot's property is ordered, the value is inserted at the position given in addAt. A negative value of addAt indicates adding to the last position.

Properties of this command are shown in the following table.

| Property                                | Description                                                                     |
|-----------------------------------------|---------------------------------------------------------------------------------|
| «inputPin» source : Value[0..1]         | The value to be added to the target slot.                                       |
| «inputPin» target : Slot[0..1]          | The slot to which the value is to be added.                                     |
| «inputPin» tarObj : ObjectOfClass[0..1] | The target object whose slot is to be changed.                                  |
| type : Text[0..1]                       | The fully qualified name of the type of the object whose slot is to be changed. |

| Property               | Description                                                                                                  |
|------------------------|--------------------------------------------------------------------------------------------------------------|
| propName : Text [0..1] | The fully qualified name of the property of the slot to be changed.                                          |
| addAt : Integer = -1   | If greater than or equal to 0, the value will be added to the slot (in case it is ordered) at this position. |

### **Class** CmdReplaceValueInSlot

`CmdReplaceValueInSlot` replaces one or more occurrences of the value given in `replaceVal` with the value given in the input pin `source` in the target slot. The properties `replaceAll`, `replaceFirst`, `replaceLast`, and `replaceAt` have the same meaning as the corresponding properties of `CmdRemoveValueFromSlot`.

Properties of this command are shown in the following table.

| Property                                                   | Description                                                                                                     |
|------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| <code>&lt;inputPin&gt; source : Value[0..1]</code>         | The value to replace with.                                                                                      |
| <code>&lt;inputPin&gt; target : Slot[0..1]</code>          | The slot whose value is to be changed.                                                                          |
| <code>&lt;inputPin&gt; tarObj : ObjectOfClass[0..1]</code> | The target object whose slot is to be changed.                                                                  |
| <code>type : Text [0..1]</code>                            | The fully qualified name of the type of the object whose slot is to be changed.                                 |
| <code>propName : Text [0..1]</code>                        | The fully qualified name of the property of the slot to be changed.                                             |
| <code>replaceVal : Value[0..1]</code>                      | The value to be replaced.                                                                                       |
| <code>replaceAll : Boolean = false</code>                  | If true, all occurrences of the given value in the slot (in case it is non-unique) will be replaced.            |
| <code>replaceFirst : Boolean = false</code>                | If true, the first value in the slot (in case it is ordered) will be replaced.                                  |
| <code>replaceLast : Boolean = false</code>                 | If true, the last value in the slot (in case it is ordered) will be replaced.                                   |
| <code>replaceAt : Integer = -1</code>                      | If greater than or equal to 0, the value in the slot (in case it is ordered) at this position will be replaced. |

### **Class** CmdReorderValuesInSlot

`CmdReorderValuesInSlot` works on an ordered slot. It has no effect on an unordered slot. It exchanges the places of the two values in the slot at the positions given in the attributes `pos1` and `pos2`.

## Part III: Concepts

---

Properties of this command are shown in the following table.

| Property                                | Description                                                                 |
|-----------------------------------------|-----------------------------------------------------------------------------|
| «inputPin» input : Slot[0..1]           | The (ordered) slot whose two values will be swapped                         |
| «inputPin» target : ObjectOfClass[0..1] | The target object on whose slot the command will operate                    |
| type : Text[0..1]                       | The fully qualified name of the type of the object having the operated slot |
| propName : Text[0..1]                   | The fully qualified name of the (ordered) property of the operated slot     |
| pos1, pos2 : Integer = 0                | The positions of the two values to swap                                     |

### Class CmdCopySlot

`CmdCopySlot` copies the value from one slot to another. It can be very convenient in the GUI for operations that involve copying values from one slot to another by, for example, dragging a slot (or an object) to another slot (or object).

Properties of this command are shown in the following table.

| Property                                | Description                                                    |
|-----------------------------------------|----------------------------------------------------------------|
| «inputPin» source : Slot[0..1]          | The source slot whose value is to be copied to the target slot |
| «inputPin» srcObj : ObjectOfClass[0..1] | The source object whose slot is to be copied                   |
| srcType : Text[0..1]                    | The fully qualified name of the type of the source object      |
| srcPropName : Text[0..1]                | The fully qualified name of the property of the source slot    |
| «inputPin» target : Slot[0..1]          | The target slot that will get the value of the source slot     |
| «inputPin» tarObj : ObjectOfClass[0..1] | The target object whose slot will get the copied value         |
| tarType : Text[0..1]                    | The fully qualified name of the type of the target object      |
| tarPropName : Text[0..1]                | The fully qualified name of the property of the target slot    |

## Link Commands

Commands of this category are specializations of the commands from the previous category, and extend or redefine some features to work specifically on slots that are instances of association ends.

### **Class CmdDeleteLink Specializes CmdRemoveValueFromSlot**

CmdDeleteLink deletes one or more links by removing one or more occurrences of the given object in the collection designated by the given association end instance.

The following table lists the features this command adds or redefines. (The others are inherited.)

| Property                                                                                                     | Description                                  |
|--------------------------------------------------------------------------------------------------------------|----------------------------------------------|
| <code>&lt;inputPin&gt; target : AssocEndInstance[0..1]<br/>{redefines CmdRemoveValueFromSlot::target}</code> | The slot from which a value is to be removed |
| <code>&lt;inputPin&gt; source : ObjectOfClass[0..1]<br/>{redefines CmdRemoveValueFromSlot::source}</code>    | The value to be removed from the slot        |

### **Class CmdCreateLink Specializes CmdAddValueToSlot**

CmdCreateLink creates a new link between the source object and the target association end instance. If the association end instance is ordered, the object is inserted at the position given in addAt. The unlinkPrevious property offers another convenience: If this attribute is true, the link of the same association linking the source object will be deleted prior to creating a new one with the target slot.

For example, if an Employee must be reassigned to another Department by a simple drag-and-drop, this command can be configured for that drag-and-drop with unlinkPrevious set to true, if the multiplicity at the association end dept by the class Department is 0..1. When an Employee (which is the value of the input pin source of the command instance) is dropped onto a Department (or its slot members) as the target, a new link between the Employee and the Department will be established by issuing this command, but the previous link of that very Employee will be deleted first.

The following table lists the features this command adds or redefines. (The others are inherited.)

| Property                                                                                                | Description                                                                                                                                                            |
|---------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>&lt;inputPin&gt; source : ObjectOfClass[0..1]<br/>{redefines CmdAddValueToSlot::source}</code>    | The value to be added to the target slot.                                                                                                                              |
| <code>&lt;inputPin&gt; target : AssocEndInstance[0..1]<br/>{redefines CmdAddValueToSlot::source}</code> | The slot to which the value is to be added.                                                                                                                            |
| <code>unlinkPrevious : Boolean = false</code>                                                           | If this attribute is true, the link of the same association linking the object referred to by source will be deleted prior to creating a new one with the target slot. |

### **Class CmdReplaceLink Specializes CmdReplaceValueInSlot**

CmdReplaceLink has the same effect on the target association end instance as CmdReplaceValueInSlot on the target slot. The property unlinkPrevious has the same meaning as in CmdCreateLink.

## Part III: Concepts

---

The following table lists the features this command adds or redefines. (The others are inherited.)

| Property                                                                                                            | Description                                                                                                                                                           |
|---------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>&lt;&lt;inputPin&gt;&gt; source : ObjectOfClass[0..1]<br/>{redefines CmdReplaceValueInSlot::source}</code>    | The new value that replaces the previous one.                                                                                                                         |
| <code>&lt;&lt;inputPin&gt;&gt; target : AssocEndInstance[0..1]<br/>{redefines CmdReplaceValueInSlot::target}</code> | The slot whose value is to be changed.                                                                                                                                |
| <code>replaceVal : ObjectOfClass[0..1] {redefines<br/>CmdReplaceValueInSlot::replaceVal}</code>                     | The value to be replaced.                                                                                                                                             |
| <code>unlinkPrevious : Boolean = false</code>                                                                       | If this attribute is <code>true</code> , the link of the same association linking the source object will be deleted prior to creating a new one with the target slot. |

## Create and Link Object Commands

The following discussions examine the commands that create an object and link it to one or two other objects.

### Class CmdCreateObjectAndLinkToObject

`CmdCreateObjectAndLinkToObject` creates a new object of the class specified in `className`, and links the new object to the given association end instance. Having one dynamic parameter (the affected target association end instance specified in one of two available ways), this command can be issued for a target object or association end instance to create a new object and immediately link it to the target.

For example, creating in some way “dependent” entities (such as “sub-elements” of the target element, or “members” of the target element) can be easily achieved by configuring this command for the selected target. The command can be issued from the right-click context menu of the selected target, or from a button or another GUI control that is enabled when the target is selected.

Properties of this command are shown in the following table.

| Property                                                              | Description                                                                       |
|-----------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| <code>&lt;&lt;inputPin&gt;&gt; target : AssocEndInstance[0..1]</code> | The instance of an association end to which a newly created object will be linked |
| <code>&lt;&lt;inputPin&gt;&gt; tarObj : ObjectOfClass[0..1]</code>    | The target object whose slot will be changed                                      |
| <code>type : Text[0..1]</code>                                        | The fully qualified name of the type of the target object                         |
| <code>assocEndName : Text[0..1]</code>                                | The fully qualified name of the association end of the target slot                |

| Property                                 | Description                                                                                                     |
|------------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| className : Text                         | The fully qualified name of the class whose object will be created                                              |
| addAt : Integer = -1                     | The position of the newly created object in the collection of the target association end, in case it is ordered |
| «outputPin» output : ObjectOfClass[0..1] | The created object                                                                                              |

**Class** CmdCreateObjectAndLinkToTwo

CmdCreateObjectAndLinkToTwo creates a new object, and links it to two other association end instances. This command can be issued on a drag-and-drop of one element (the source object or association end) on another element (the target object or association end). This command is particularly useful for creating objects of classes that are conceptually association classes (that is, whose objects are always “dependent on” and thus, linked to two other objects).

For example, adding a Product to an Order can be achieved by configuring this command on a drag-and-drop of a Product to an Order, because it creates an Order Item that is linked to the Product and to the Order. Similarly, adding a Product to a Pricelist can be achieved with the same command because it creates a Pricelist Entry that is linked to the Product and the Pricelist, and so on. Many other cases of similar usage exist in practice.

The following table lists the properties for this command.

| Property                                   | Description                                                                        |
|--------------------------------------------|------------------------------------------------------------------------------------|
| «inputPin» source : AssocEndInstance[0..1] | The first association end instance to which a newly created object will be linked  |
| «inputPin» srcObj : ObjectOfClass[0..1]    | The first object whose slot will be changed                                        |
| srcType : Text[0..1]                       | The fully qualified name of the type of the first object                           |
| srcAssocEndName : Text[0..1]               | The fully qualified name of the association end of the first slot                  |
| «inputPin» target : AssocEndInstance[0..1] | The second association end instance to which a newly created object will be linked |
| «inputPin» tarObj : ObjectOfClass[0..1]    | The second object whose slot will be changed                                       |
| tarType : Text[0..1]                       | The fully qualified name of the type of the second object                          |
| tarAssocEndName : Text[0..1]               | The fully qualified name of the association end of the second slot                 |

*Continued*

## Part III: Concepts

(continued)

| Property                                 | Description                                                                                                                                 |
|------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| className : Text                         | The fully qualified name of the class whose object will be created                                                                          |
| addAtSrc : Integer = -1                  | The position of insertion of the newly created object to the collection of the association end of the first object (in case it is ordered)  |
| addAtTar : Integer = -1                  | The position of insertion of the newly created object to the collection of the association end of the second object (in case it is ordered) |
| «outputPin» output : ObjectOfClass[0..1] | The created object                                                                                                                          |

### Section Summary

- ❑ The OOIS UML model library contains a package with a set of predefined, built-in commands that enclose some fundamental UML structural actions:
  - ❑ CmdCreateObject — Creates a new object of the class provided to its input pin
  - ❑ CmdCreateObjectOfClass — Creates a new object of the class specified by its fully qualified name
  - ❑ CmdDestroyObject — Destroys an object
  - ❑ CmdClearSlot — Clears a slot
  - ❑ CmdRemoveValueFromSlot — Removes a value from a slot
  - ❑ CmdSetSlot — Sets the value of a slot
  - ❑ CmdAddValueToSlot — Adds a value to a slot
  - ❑ CmdReplaceValueInSlot — Replaces a value in a slot
  - ❑ CmdReorderValuesInSlot — Swaps the positions of two values in an ordered slot
  - ❑ CmdCopySlot — Copies the value of one slot to another
  - ❑ CmdDeleteLink — Deletes one or more links
  - ❑ CmdCreateLink — Creates a link
  - ❑ CmdReplaceLink — Replaces a link
  - ❑ CmdCreateObjectAndLinkToObject — Creates an object and links it to a slot
  - ❑ CmdCreateObjectAndLinkToTwo — Creates an object and links it to two slots

# Presentation

Although OOIS UML does not include concrete libraries or authoritative specifications of the elements that would support the application's presentation layer (that is, the GUI), it still gives a set of ideas, concepts, and suggestions that concrete implementations could follow in order to exploit the full power of the profile. A concrete implementation is free to interpret, implement, and extend these ideas, concepts, and suggestions in any way or to any extent. This section briefly describes the ideas, concepts, and suggestions in a way that provides an impression about how complex GUIs can be developed efficiently.

In this section, the concepts from the GUI presentation layer modeled with classes in this layer will be capitalized for convenience.

## The Presentation Layer Architecture

The set of concepts and ideas presented here does not tend to offer yet another classical GUI framework that deals with GUI controls, their properties, behavior, and mechanisms of interaction. In other words, this section does not describe the elements of a class library of a usual GUI framework, or how to program a GUI using it. Instead, these concepts and ideas assume that an existing classical GUI framework is used as the underlying basis and a hosting environment, and that a layer above it is built up, at a higher level of abstraction and of coarser granularity. The aim is to exploit the expressiveness of models in OOIS UML and boost the developers' efficiency in designing GUIs.

To illustrate this, let's look at several very simple and typical examples:

- A classical GUI framework may recognize a button or a hyperlink as the control by which the user issues a command to the application. The developer must write the program code of the method that handles a click on that control.

In the OOIS UML Presentation layer, such a control is wrapped up into a component that can be directly connected to a command prototype, so that the developer needs simply to connect a command prototype (which is an object) to that component (which is another object with full OOIS UML semantics). That connection is a standard link between two objects, and can be manipulated in all the usual ways. For example, the developer can establish the connection in a What-You-See-Is-What-You-Get (WYSIWYG) GUI design environment by a drag-and-drop or similar configuration action.

- Assume that the abovementioned button or hyperlink must perform the command on a selected object of a certain type. If an object of that type is selected in the given part of GUI, the control should be enabled; otherwise, it should be disabled. A classical GUI framework would require the developer to program this behavior by coding event handlers of other controls in which the selection is done, and to toggle the "enabled" property of the button or hyperlink.

On the other hand, the OOIS UML control relies on the same basic controls and mechanisms of the GUI framework, but also provides a concept of *input* and *output* pins of GUI widgets. As usual, these are input and output parameters of GUI widgets that can be bound together to ensure coupling and synchronization by means of propagation of values between them. In this case, another GUI widget (for example, a tree view) has its output pin on which it always supplies the currently selected item.

On the other hand, the button or hyperlink widget may have its input pin that accepts the object that it propagates to the input pin of the attached command when pressed. The widget

## Part III: Concepts

---

automatically disables itself if the value on its input pin is not such that the attached command can be applied to it, and vice versa. Consequently, the only thing the developer must do is to bind the output pin of other components to the input pin of the button or hyperlink component, and the implementation of the OOIS UML Presentation layer geared to the classical GUI framework would do the rest.

- A classical GUI framework may provide a widget that renders a picture, supplied as an array of bytes or a file. In order to render a photo of a person, modeled as an attribute of the corresponding domain class `Person`, the developer must write some glue code to bind the widget to the attribute value.

In the OOIS UML Presentation layer, the picture widget is wrapped up with features that enable its direct binding to the object space. It simply should be configured to render the attribute named `photo` of the object of class `Person` provided on its input pin, and that would be all. As in the previous example, this input pin can be connected to an output pin of, for example, a tree view, so that whenever the user selects an object of class `Person` in the tree view, the picture widget will automatically render that person's photo.

- A classical GUI framework usually requires some glue code to react to a drag-and-drop, and the developer must write that code for almost all particular types of the dragged and dropped objects and contexts.

In the OOIS UML Presentation layer, this is simply a matter of configuring a command for a certain pair of items, as already described in Chapter 6.

Figure 16-2 shows the elements of the OOIS UML Presentation layer. The first and lowest sub-layer is the GUI Style Configuration. It relies on the concepts of *GUI Item Setting* and *GUI Context* introduced in Chapter 6.

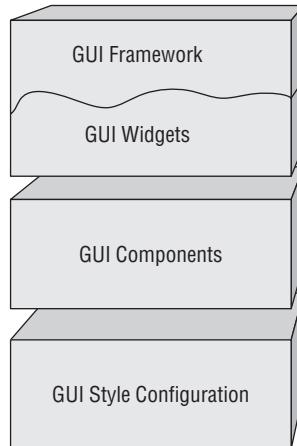
As described, a GUI Item Setting is an object that configures how items of a certain kind (for example, objects of classes, slots, and so on) appear and behave in the GUI. In particular, it defines the icons, textual labels, commands issued on drag-and-drop, and so on.

A GUI Context is a container of GUI Item Settings that defines a coherent style in one particular context of the application. The context can be one particular GUI component, a dialog, part of a Web page, a window, or a set of Web pages, depending on the design of the GUI. In that sense, a GUI Context is analogous to the notion of Cascading Style Sheets (CSS) in HTML Web design, although it has a slightly different level of abstraction and a completely different language of specification — it is an object structure instead of a textual specification. Actually, CSS in a Web environment can be part of a GUI Context (that is, its element). This layer is an ordinary structure of objects of classes with the OOIS UML semantics, while the concrete implementation of the entire Presentation layer in a certain hosting GUI framework must interpret the structure to achieve the specified appearance and behavior.

The rest of the Presentation layer is based on two additional fundamental concepts: *GUI Components* and *GUI Widgets*. A GUI Component is a static specification of a certain part of the GUI, a building block of the GUI construction. On the other hand, a GUI Widget is a dynamically instantiated element of the GUI, a control that appears in the GUI during the execution of the application.

On the one hand, GUI Components constitute the specification of the GUI, and are created and manipulated during the design or customization of the GUI. On the other hand, GUI Widgets are created and destroyed by the hosting GUI framework for each activation of the application, for each user session, or for each instance of the same type of window, Web page, and so on. GUI Components are conceptual

things; GUI Widgets are visible and tangible things on the computer screens (or more precisely, their object representations within the hosting GUI framework).



**Figure 16-2: The architecture of the Presentation layer.**  
The GUI Style Configuration and GUI Components sub-layers are structures of objects of OOIS UML classes that define the static structure of the GUI and its presentational and behavioral configuration. The GUI Widget sub-layer consists of dynamically managed instances of classes from the specific GUI framework that deal with actual GUI controls.

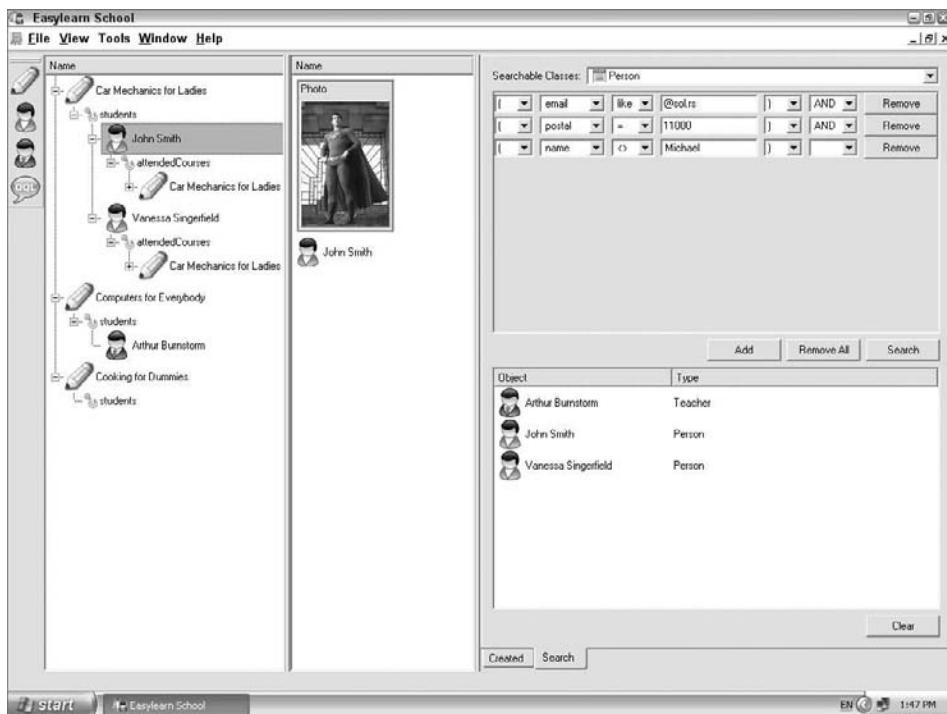
GUI Components exist even if the application is not running at all. They are part of its specification. GUI Widgets exist only within running applications. They are part of their execution. During the execution of the application, the implementation of the Presentation layer interprets the structure of GUI Components as interrelated objects, and dynamically creates and mutually binds GUI Widgets as objects within the hosting GUI framework. The framework is then responsible for providing the appropriate appearance and behavior of the GUI Widgets, according to the Style Configuration in the particular GUI Context. Finally, GUI Components are ordinary objects of classes with the full OOIS UML semantics. GUI Widgets are, on the other hand, objects of classes of the hosting framework or other classes that adapt the concepts of OOIS UML GUI configuration to the target framework. Normally, they do not have OOIS UML semantics, but the semantics of the target implementation language.

For example, let's consider a simple GUI whose one screenshot is shown in Figure 16-3a. Figure 16-3b shows how this GUI is built up from GUI Components.

The following numbers correspond to the sections shown in Figure 16-3b:

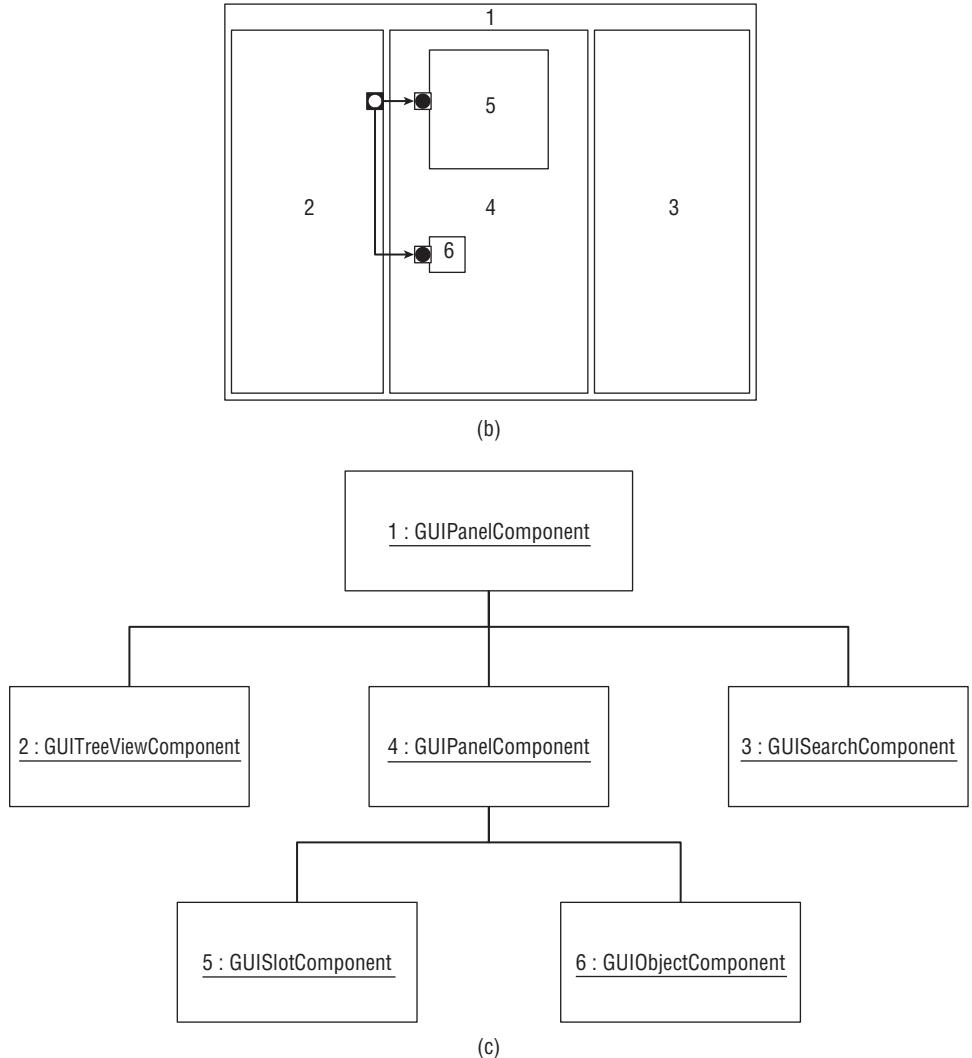
## Part III: Concepts

1. The topmost Component is a Panel Component that contains three other Components, 2, 3, and 4. A Panel is a simple Component container.
2. On the left-hand side, there is a standard Tree View Component. It is configured to render a tree rooted from a specified object. It has one output pin. At run-time, the GUI Widget that corresponds to this Component will provide the currently selected node to its output pin.
3. On the right-hand side is a standard Search Component. It provides the means for a generic search of the object space.
4. In the center is another Panel Component that contains two other subcomponents (5 and 6).
5. A Slot Value Component (5) specifies that the Widget created at run-time for it will render the value in a slot (as an instance of an attribute or association end) of an object provided to its input pin. As usual, the name of the slot's property is given in an attribute of the Slot Component. The kind of the Widget will depend on the type of the property. In this case, the property is of type Picture, and the Widget will be the one that renders a picture from the attribute value. The Component is configured to render the value of the attribute photo of the object provided to the input pin. Because the input pin is connected to the output pin of the Tree View Component, at run-time, the two corresponding Widgets will be coupled and synchronized — whenever the user selects an object of Person, the Widget 5 will render its photo. Except for connecting the two pins of Components, the developer has nothing else to do to ensure such behavior.



(a)

Figure 16-3: GUI Components and Widgets. (a) A screenshot of a simple GUI, consisting of a number of GUI Widgets.



**Figure 16-3:** (b) The GUI Components that constitute the GUI shown in (a). The boxes with circles in them represent input and output pins of components. (c) An object diagram of GUI Components shown in (b).

6. Below the picture there is an Object Component. At run-time, the Widget created from that Component renders the object provided to its input pin, according to the specifications in the GUI Context: the icon, the label, the commands in the right click pop-up menu, the reaction to drag-and-drop, and so on. Because its input pin is also connected to the output pin of the Tree View Component, at run-time, the two corresponding Widgets will be synchronized, too. Whenever the user selects a person in the tree view, Widget 6 will display the person's icon and name. Moreover, using this Widget, the user can apply the available commands on the selected person object (for example, delete it, open a specification dialog for it, drag and

## Part III: Concepts

---

drop it to another place, or drag and drop another object on it). Again, except for connecting the two pins of Components, the developer has nothing else to do to ensure such behavior.

As a result, this simple GUI, which can be constructed in a very efficient and easy way by simply configuring Components and binding their pins, provides very advanced appearance and behavior. The implementation of the Presentation layer for the hosting GUI framework has the task to interpret the provided Component configuration and instantiate it by creating the appropriate Widgets for them, and to couple the Widgets to provide the necessary behavior at run-time. A Component can be seen as a template from which many Widgets can be created at run-time. For each activation of the application, for each user session, and for each instance of a particular type of window or Web page, a new GUI Widget will be created from the same GUI Component that specifies it.

GUI Components are simple objects of classes with the full OOIS UML semantics provided in the Presentation layer model library. Figure 16-3c shows a diagram of the object structure that defines the GUI configuration shown in Figure 16-3b. GUI Widgets are, on the other hand, instances of the classes of the hosting GUI framework (or an OOIS UML adapter to that framework), and are tightly coupled with that framework and involved in its specific internal mechanisms. They do not have to obey the semantics of OOIS UML, and they usually do not. In other words, in order to implement the Presentation layer in a particular GUI framework, the GUI Widget layer must be adapted to the framework. On the other hand, the Component layer is a standard OOIS UML model package and is completely independent of the GUI framework.

Another thing that you should notice from the given example is that GUI Widgets are sometimes directly related to GUI Components that caused their instantiation at run-time. Those Components also define the binding of pins and other configurable parameters of appearance and behavior of the Widgets created from them. In the given example, such Widgets are the Widgets created for the Components 1–6 for each instantiation of the window type shown in Figure 16-3.

However, many Widgets in a running GUI are not directly connected to the Components, and are not their peer instantiations. For the given example, every node in the tree view within Widget 2 in Figure 16-3a that renders an object of a class is a separate Object Widget of the same kind as Widget 6. It renders an object attached to it with its icon, label, and so on. It can offer commands in the right-click pop-up menu, react to drag-and-drop, and so on, according to the specifications in the enclosing GUI Context.

However, this Widget is not created as a direct result of a GUI Component, as is Widget 6, but is created dynamically by its enclosing Tree View Widget, when the latter builds the tree starting from its configured root node by traversing the object structure according to the specifications in the enclosing GUI Context. This tree structure of Widgets is reconstructed or updated every time the Tree View Widget needs to be refreshed (that is, when it is notified on a certain important change from the object space, or on a change of value on its input pin). This similarly holds for Widgets created dynamically in the search result window of the Search Widget 3 in Figure 16-3a.

The GUI Style Configuration and GUI Component layers of an application consist of structures of objects of the corresponding OOIS UML classes. An implementation of the profile can provide any combination of the following ways to manage these object structures so that the developer can specify and configure the GUI of an application:

- By modeling creational object structures for these layers, possibly using a specific notation for this purpose, as described in Chapter 6.

- ❑ By maintaining these layers as usual object structures at run-time (creating and deleting objects and links and setting attribute values).
- ❑ By providing a specialized, WYSIWYG environment for designing and customizing the GUI. The environment can provide different mechanisms for this, such as specialized configuration dialogs and wizards, configuration by demonstration, or other WYSIWYG techniques. For example, to design a type of a window or a Web page of an application, the developer can simply point to the region of the screen (that is, of the container Component) and select and configure a Component wanted at that part of the screen, in a usual WYSIWYG manner. Note that because it ultimately affects an ordinary object structure in the background, this operation can be done at run-time, within the running application, without any need for recompilation as in classical WYSIWYG GUI design environments.

As a final remark, let's emphasize once more that all these observations and concepts hold both for desktop as well as for Web-based GUIs. Although the running example was for a usual desktop GUI, the idea is to use the same approach and concepts for building Web-based GUIs.

A Web-based GUI can also be built out of Components and Widgets as described here. A difference could be in the available library of Components and Widgets and their implementation. There is nothing that would prevent that approach because the only difference between a desktop and a Web-based implementation could be in the way and place the ultimate contents of the screen are rendered, and how the actions from input devices are handled on these two platforms, or in the behavioral and presentational capabilities of the GUI Widgets themselves. However, this approach leaves that level of detail to the responsibility of the concrete implementation for a specific GUI framework that can be dedicated for either desktop or Web.

For example, an implementation of this approach for a desktop GUI framework would create instances of the classes from the GUI framework that are then responsible to render the Widgets on the screen by system calls. On the other hand, a Web-based implementation can have similar classes that render the Widgets by producing an HTML output sent to the client's Web browser. In addition, the newest advances in Web-based GUI infrastructural technologies (including Ajax) tend to reduce that difference even more, which is an extra argument in favor of the presented approach.

### Section Summary

- ❑ The OOIS UML Presentation layer consists of three sub-layers:
  - ❑ The GUI Style Configuration layer provides the concepts of *GUI Item Setting* and *GUI Context*, and defines the structure of objects of these classes that specify and configure the appearance and behavior of GUI items in different parts of the application.
  - ❑ The GUI Component layer defines the configuration of the GUI in terms of GUI Components. A *GUI Component* is a static specification of a certain part of the GUI, a building block of the GUI construction.

*Continued*

GUI Components are objects of OOIS UML classes built in the implementation of the Presentation layer, independently of the hosting GUI framework.

- The GUI Widget layer is an adapter of the Presentation layer to the concrete hosting GUI framework. A *GUI Widget* is a dynamically instantiated element of the GUI, a control that appears in the GUI during the execution of the application. It is an instance of a class from the hosting GUI framework.

### GUI Style Configuration

Figure 16-4 shows the core parts of the conceptual model of the GUI Style Configuration layer. Figure 16-4a shows the concept of GUI Item Setting. It is an abstract generalization of different kinds of configuration settings for specific kinds of GUI items. A GUI Item Setting has a name and description for the convenience of identification by users.

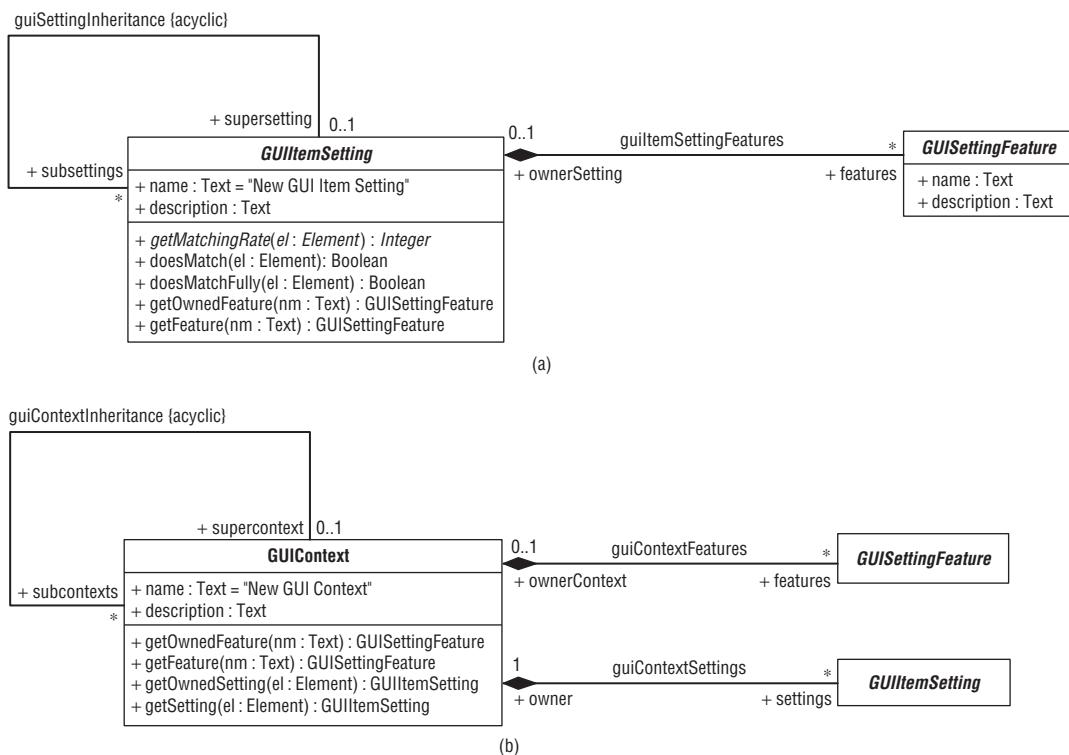


Figure 16-4: GUI Style Configuration fundamental concepts. (a) GUI Item Setting. (b) GUI Context.

A GUI Item Setting defines presentational and behavioral parameters for Elements of one kind, or for one particular Element, where Element is an abstract generalization of all model and run-time elements that

are supported by the implementation, and that can be represented in the GUI as items (classes, properties, objects, slots, and so on). The set of Elements for which a GUI Item Setting defines presentational and behavioral parameters (that is, the set of Elements whose appearance and behavior in the GUI can be affected by the Setting) will be called the *extent* of the Setting.

Each object of GUI Item Setting has its extent that is determined by the concrete (non-abstract) class specialized from `GUIItemSetting` of which it is a direct instance, as well as by the values of its properties. For example, an implementation of this layer can provide the following set of specializations of GUI Item Setting:

- ❑ **GUI Class Setting** — Configures appearance and behavior of items that represent any or one particular class. Its extent can cover all classes from the model, or one particular class with the fully qualified name given in a textual attribute of the class `GUIClassSetting`.
- ❑ **GUI Property Setting** — Configures appearance and behavior of items that represent any, some, or one particular property. Its extent can cover all properties from the model, or all properties of one particular class from the model, or one particular property, depending on the values of the properties of the class `GUIPropertySetting`.
- ❑ **GUI Slot Setting** — Configures appearance and behavior of items that represent any, some, or one particular slot. Its extent can cover all slots, or slots of all properties of one particular class from the model, or slots of one particular property, or one particular slot, depending on the values of the properties of the class `GUISlotSetting`.
- ❑ **GUI Object Setting** — Configures appearance and behavior of items that represent any, some, or one particular object. Its extent can cover all objects, or all objects of one particular class from the model, or one particular object, depending on the values of the properties of the class `GUIOBJECTSetting`.

As you can see, a GUI Item Setting can have an extent as broad as to include all classes, properties, objects, or slots, but also as narrow as to include just one particular class, property, object, or slot. Consequently, there can be two or more GUI Item Settings, even within the same GUI Context, with overlapping extents. In that case, two or more GUI Item Settings can specify appearance and behavior of the Element in question.

To resolve such a situation, the notion of *matching rate* is introduced. It is an integer that measures how a particular GUI Item Setting is relevant for the given Element. The integer measures the “conceptual distance” of the GUI Item Setting from the given Element. The rate 0 (zero) means that the GUI Item Setting “perfectly” matches the given Element and is most relevant to describe it. The bigger the positive integer, the worse it matches; the negative value  $-1$  indicates no matching (that is, an “infinite distance”). Therefore, non-negative values mean that the given Element is within the extent of the GUI Item Setting, while  $-1$  means that it is out of the extent.

The abstract operation `GUIItemSetting::getMatchingRate` returns the matching rate. Concrete subclasses provide the appropriate methods for this operation. In general, the rules are as follows:

- ❑ A GUI Item Setting whose extent includes only one particular Element returns 0 for that particular Element, and  $-1$  for all others.
- ❑ A GUI Item Setting whose extent includes all Elements of one kind (for example, all objects or all classes) returns a positive integer ( $>0$ ) for any Element of that kind and  $-1$  for all others.

## Part III: Concepts

---

The matching resolution algorithm will select the GUI Item Setting that best matches the given Element — that is, that has the smallest non-negative matching rate. Actually, it will return the GUI Item Setting with the smallest extent that includes the Element. That way, the GUI style for one kind of Element in one Context can be defined by a GUI Item Setting of a broader extent, and then redefined for a subset of Elements of that kind or for one particular Element by defining another GUI Item Setting with a narrower extent.

The operation `GUIItemSetting::doesMatch` returns `true` if the matching rate for the given Element is non-negative, while the operation `GUIItemSetting::doesMatchFully` returns `true` if the matching rate for the given Element is zero.

A GUI Item Setting can own an arbitrary number of *GUI Setting Features*. A GUI Setting Feature is an abstract generalization of different kinds of features of one GUI Item Setting or GUI Context. Each GUI Setting Feature defines one aspect of presentation or behavior of its owner's extent. For example, one kind of GUI Setting Feature can define the icon used for the configured Element(s). The icon is simply provided in the value of an attribute of type `Picture` of that subclass of the class `GUISettingFeature`. Similarly, another kind of GUI Setting Feature can be used to specify how textual labels are obtained. By combining attributes of enumeration and textual types, the Feature can specify that the label is a fixed value (defined in a textual attribute of the Feature), or taken from an attribute of the configured object, and so on.

A GUI Item Setting can have at most one *supersetting* and an arbitrary number of *subsettings*. Supersettings are those GUI Item Settings that are searched for a Feature for a certain Element, if the considered GUI Item Setting does not have that Feature. Consequently, a GUI Item Setting inherits the Features from its supersetting, and can redefine some. If it exists, a GUI Setting Feature redefines the Feature of the same kind and with the same identifier (name) in the supersetting. Otherwise, the Feature from the supersetting is inherited.

To support such behavior, the operation `GUIItemSetting::getOwnedFeature` returns the owned Feature with the given name. On the other hand, the operation `GUIItemSetting::getFeature` returns the first encountered Feature with the given name, owned (if such exists), or inherited, relying on a recursive call of the same operation of the supersetting.

GUI Context (refer to Figure 16-4b) is a container of GUI Item Settings. In addition, it can have its own Features that define presentational and behavioral parameters for the entire Context, and not for one particular extent. Similar to GUI Item Settings, a GUI Context can have its supercontext, and can inherit or redefine both GUI Setting Features and GUI Item Settings. The operations of GUI Context have the purpose of accessing only the owned Settings and Features of the Context, or the most effective ones (owned or inherited, recursively).

### Section Summary

- ❑ A GUI Item Setting defines presentational and behavioral parameters for Elements of one kind, or for one particular Element, where Element is an abstract generalization of all model and run-time elements.
- ❑ The set of Elements for which a GUI Item Setting defines presentational and behavioral parameters is called the *extent* of the Setting. A GUI Item Setting can

have an extent as broad as to include all classes, properties, objects, or slots, but also as narrow as to include just one particular class, property, object, or slot.

- ❑ The matching resolution algorithm will select the GUI Item Setting that best matches the given Element (that is, that has the smallest extent that includes the Element).
- ❑ A GUI Item Setting can own an arbitrary number of *GUI Setting Features*. Each GUI Setting Feature configures one aspect of presentation or behavior of its owner's extent.
- ❑ A GUI Item Setting can have at most one *supersetting* and an arbitrary number of *subsettings*. A GUI Item Setting inherits the Features from its supersetting, and can redefine some.
- ❑ A GUI Context is a container of GUI Item Settings. It can have its Features that define presentational and behavioral parameters for the entire Context, and not for a particular extent of Elements.
- ❑ A GUI Context can have its supercontext, and can inherit or redefine GUI Setting Features and GUI Item Settings.

## GUI Components and Widgets

Figure 16-5 shows an excerpt from the conceptual model for the GUI configuration using GUI Components. The diagram shows some of the features of Components by which they can fulfill their major responsibilities.

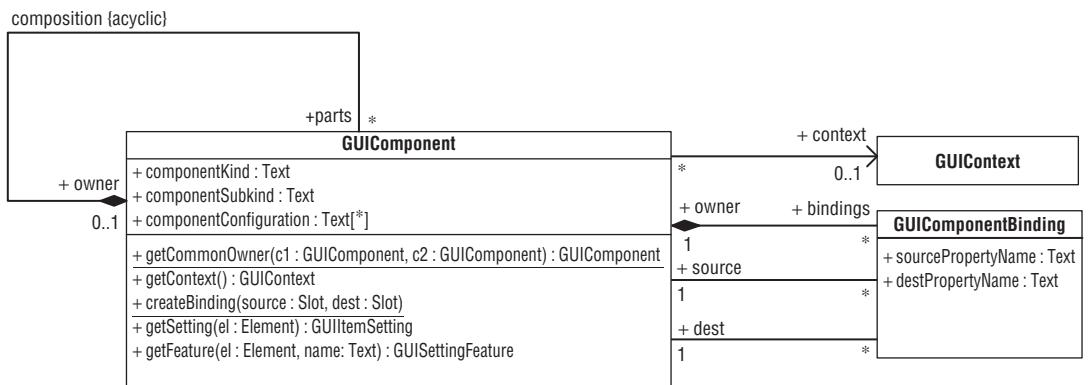


Figure 16-5: GUI Components and their composition and binding

### Composition

GUI Components can be nested — that is, one Component can contain other subcomponents, and so on. For example, a panel can contain subcomponents such as tables, pictures, other panels, and so on.

## Part III: Concepts

---

### Context

A Component must provide a GUI Context to the Widgets created from it. If a Component does not have a directly associated GUI Context, it will inherit one from its enclosing (owner) Component. The operation `GUIComponent::getContext` returns the GUI Context owned or inherited by the Component. Via that Context and its GUI Item Settings, Widgets created from the Component can get the proper styling.

### Configuration of Widgets

It is not necessary to design a separate subclass of `GUIComponent` for each and every kind of Widget. Different kinds of Widgets can be created from the same type of Component. Each object of Component can have different values of its properties, which then determine what kind of Widget will be created out from it in the GUI.

Actually, the Widgets that appear differently, but basically represent the same conceptual element and behave in a similar way, can usually be constructed from the same type of Component. For example, an object of a class can be represented by completely differently looking Widgets — one can render only the icon for the object, the other can render only a textual label for the object, and so on. Only one type of Component — for example, `GUIObjectComponent` — can be sufficient for creating all these kinds of Widgets, while the values of the properties of instances of that type of Component can determine which Widget will be created at each place. Similarly, an attribute value of type `Date` can be edited using completely different Widgets supporting different input formats or calendars.

### Pins and Bindings

Input and output pins of Components are modeled as simple attributes of the subclasses of `GUIComponent`, tagged with `<<inputPin>>` or `<<outputPin>>`. They can have any type and multiplicity, which are completely ignored. The existence of an attribute that is an input or output pin only means that all Widgets created from that Component will provide the adequate pins that will accept or provide values at run-time. The values of those attributes of the Component itself are ignored.

Input and output pins can be bound in the GUI configuration. Pin binding is represented with a simple object of the class `GUIComponentBinding` that carries the information about which pins of which Components are bound together. By convention, a GUI Component Binding belongs to the GUI Component that is the first common owner of the Components whose pins it binds. The operation `GUIComponent::createBinding` creates a GUI Component Binding between the two pins of GUI Components. The implementation can also provide an appropriate Command that wraps this operation, so that the user can simply drag one pin of a Component and drop it to another in order to create a binding.

### Widget Mechanisms and Responsibilities

The concrete design of the Widgets and their interaction with the hosting GUI framework is completely left to the implementation. However, the implementation should fulfill the following key mechanisms and responsibilities of Widgets:

- ❑ **Construction** — At run-time, a Widget should be created as specified in its Component. The construction of Widgets should follow the compositional structure of Components. In addition, some Widgets (for example, tables or tree views) should dynamically create their contained Widgets for the contents they render.

- ❑ **Connection to the underlying object space (if necessary)** — For example, a Widget that renders an Element (a class, property, object, slot, and so on) maintains a reference to that Element. A tree view maintains a reference to the root Element, and so on.
- ❑ **Proper appearance and behavior in the GUI** — This is as specified by the GUI styling configuration in the given Context accessible through the Component.
- ❑ **Notification and refreshing** — Whenever there is a significant change in the underlying object space, the interested GUI Widgets should be notified and refreshed (re-rendered). The implementation can incorporate various mechanisms for optimizing refreshing, by, for example, notifying only the Widgets that are interested in (and signed for) the given kind of change.
- ❑ **Pin binding and value propagation** — Widgets should have their proper input and output pins aligned to the pins of their Components. Widgets should provide a mechanism for binding these pins on their construction according to the bindings defined between their Components. In addition, Widgets are responsible for updating the values on their output pins and reacting to the changes on their input pins. It is left to the implementation to define the precise semantics of value propagation (especially in terms of timing and synchronization), as well as to handle the cases of circular bindings that may cause oscillatory changes in pin values and similar irregular cases. The developer should take care to properly configure the GUI and avoid all irregular or unsupported configurations.

### Section Summary

- ❑ GUI Components have the following features and responsibilities:
  - ❑ Enable compositional construction of the GUI.
  - ❑ Provide the GUI Context to their Widgets, and the GUI styling configuration via that Context.
  - ❑ Provide configuration parameters for their Widgets. Different kinds of Widgets can be created from the same type of Component, depending on its properties.
  - ❑ Define input and output pins and their binding.
- ❑ GUI Widgets have to support the following key mechanisms:
  - ❑ Construction as specified in their source Components (if any)
  - ❑ Connection to the underlying object space
  - ❑ Proper appearance and behavior in the GUI, as specified by the GUI styling configuration
  - ❑ Notification on changes in the underlying object space and refreshing
  - ❑ Pin binding and propagation of values through pins

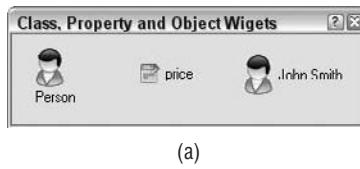
### GUI Component Library

To illustrate the potential of the presented concepts and mechanisms, as well as to give the directions and ideas of their use, implementation, and further development, this section provides a list of possible kinds of Components and their Widgets, and briefly describes them. The list is by no means complete or authoritative — an implementation is free to augment it or contribute in any other way.

#### Single Element Components

This category includes the Components whose Widgets are responsible for rendering single Elements. The Element can be a model element (for example, a class or a property) or a run-time element (for example, an object or a slot).

An *Element Component* is a Component whose Widget is responsible for rendering one single Element. For that reason, the Component has one input pin, because the input pin of its Widget refers to the Element it renders. The Element is rendered according to the GUI style configuration in the enclosing Context. The Component can provide additional configuration parameters for rendering the Element (such as whether the icon or the text label should be rendered at all, and so on). Figure 16-6a shows several examples of this kind of Widgets, rendering (from left to right) a class, a property, and an object.



**Figure 16-6: Examples of Widgets for GUI Element Components.**

**(a) Simple, generic Widgets for the Element Component rendering a class, a property, and an object.**

A *Slot Component* is a specialization of an Element Component. Each of its Widgets renders one instance of an attribute or of an association end of a certain object. The Widget can work in one of two modes, depending on a configuration parameter stored in the Component. Its input pin can either refer to the slot itself, or to the object owning the slot, while the name of the slot's property is stored in an attribute of the Component. Figure 16-6b shows examples of this kind of Widget.



(b)

**Figure 16-6: (b) Widgets for the Slot Component rendering an instance of an attribute and of an association end.**

*Slot Value Component* is a specialization of a Slot Component. However, its Widget does not render the slot itself (using the icon, label, commands, and so on, configured for that slot), but one value from that slot. The value can be the first (or the only) value from the slot, or one at the specified position if the slot is ordered. Figure 16-6c shows an example of this kind of Widget. A Widget of this Component can work in one of two modes, as configured in an attribute of the Slot Value Component. The commands available for the item (for example, in the right-click pop-up context menu or on drag-and-drop) can be taken from the GUI Style Configuration layer either for the slot itself, or for the rendered value.



(c)

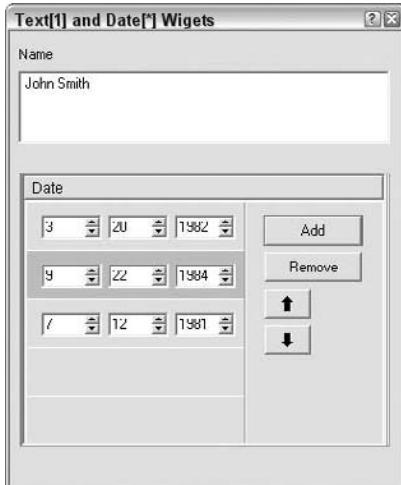
**Figure 16-6: (c) A Widget for the Slot Value Component rendering an association end instance.**

For example, let's consider an instance of the association end `customer` of an object of the class `Order` in an order processing system. Assume that the specification dialog for an object of `Order` has a Slot Value Component with the appropriately bound input pin. When the dialog is opened for a particular order, the Widget for the Slot Value Component will render the icon and label of the object linked to the order (that is, the customer of that order). However, that Widget can react to events according to the commands configured for the slot itself (and not for the linked customer).

For example, when another object of `Customer` is dragged and dropped on the slot, the `CmdReplaceLink` command can be executed upon the slot (and not upon the linked object), so that the dropped object replaces the previously linked one and becomes the customer of the considered order. This is very convenient and intuitive. The user has the brief information about the customer of the considered order, having its icon, name, and so on always displayed on the dialog for the order, while a new customer can be set by a simple drag-and-drop. What is most interesting is that such behavior can be ensured simply by proper configuration of Components and the GUI style layer, without any coding.

A *Slot Editor Component* is a specialization of a Slot Component. Its Widgets provide the means to edit the values of slots they work with. The concrete form of the Widget depends on the type and multiplicity of the slot's property.

For example, as shown in Figure 16-6d, if the property is an attribute of type `Text` and of multiplicity 1, the Widget is a simple text box. If it is, however, an attribute of type `Date` and of multiplicity \*, the Widget can be an array of cells, each of which accepts a date, while the cells can be arbitrarily added or removed from the array. Similarly, if the property is an ordered association end, the Widget can be an array of cells displaying icons and labels for the linked objects, and allowing removal, adding, or reordering of the elements in the array. In any case, the Widget relies on the built-in commands for modifying values in the slot when it performs editing. Many other forms are possible, too.



(d)

**Figure 16-6: (d) Widgets for the Slot Editor Component rendering instances of attributes of type `Text[1]` and `Date[*]`.**

### Section Summary

- ❑ An *Element Component* is a Component whose Widget is responsible for rendering one single Element.
- ❑ Specializations of an Element Component are a *Slot Component*, a *Slot Value Component*, and a *Slot Editor Component*.

## Multiple Element Components

The Widgets of a *Multiple Element Component* render collections of Elements. Figure 16-7 shows examples of such Widgets.

Through the values of its properties, a Multiple Element Component can specify various parameters of its Widgets:

- ❑ Whether the Widget renders the items in a table list, or tree view. If the selected view is tabular, each row of the table will render one Element in the collection. However, the columns of the table can be rendered in different ways. In one of them, the first column renders the current Element in the collection (using its style configuration — that is, icon and text), while the others render its subnodes as specified in the style configuration. In another, the first column renders the current Element in the collection, while the others render its name, description, and so on, as specified in the style configuration.

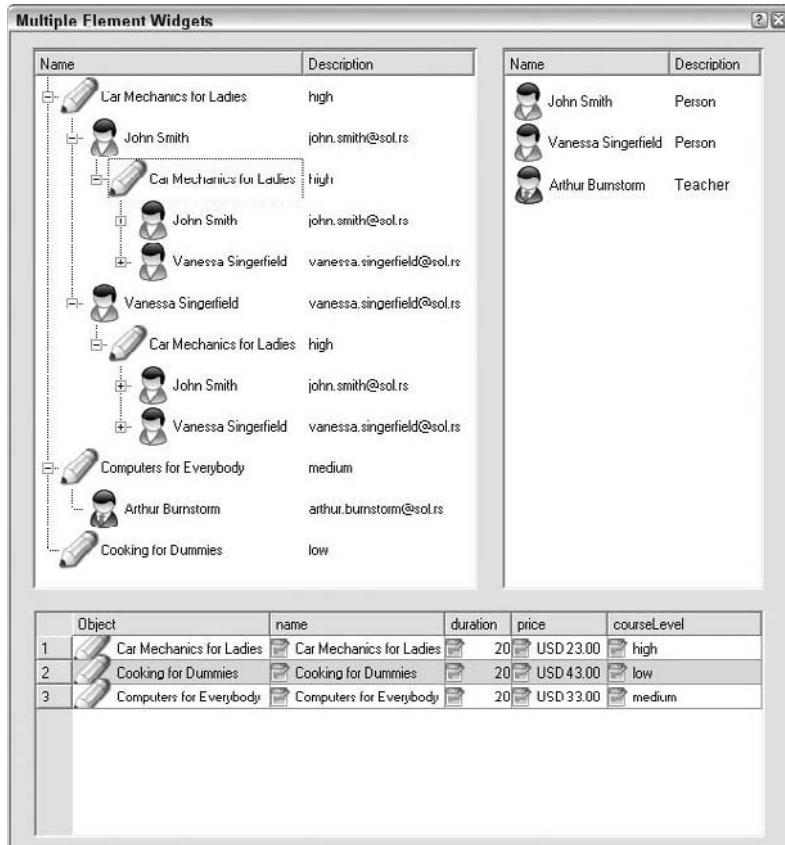


Figure 16-7: Examples of Widgets for Multiple Element Components

- Whether the Widget allows selection of items, and whether it allows selection of only one item or multiple items.
- Other elements of presentation and behavior.

The Widgets of this Component have an output pin on which they provide the selected Element(s).

The Multiple Element Component is just an abstract generalization of different kinds of Components that define how the collection of rendered Elements is obtained. Following are some of them:

- Subnodes Component** — The collection is determined as the collection of subnodes of the Element provided in the Widget's input pin, as specified in the style configuration.
- Query Result Component** — The collection is obtained from the result of a query specified in the Component.
- Collection Component** — The collection is directly provided in the input pin of the Widget.

### Section Summary

- ❑ The Widgets of a *Multiple Element Component* render collections of Elements in a table, list, or tree view, including the following:
  - ❑ **Subnodes Component** — The collection is determined as the collection of subnodes of the Element provided in the Widget's input pin, as specified in the style configuration.
  - ❑ **Query Result Component** — The collection is obtained from the result of a query specified in the Component.
  - ❑ **Collection Component** — The collection is directly provided in the input pin of the Widget.

### Search and Identification Components

In the GUIs of interactive business applications, searches for objects or tuples of objects using certain parameters entered by users are frequent. A *Search Component* supports easy implementation of such searches. It is a specialization of a Query Result Component because its Widget renders a collection of tuples returned as the result of a search initiated by the user. Figure 16-8 shows an example of such a Widget. The search is performed using the query specified by the Component. The query is supposed to have its input parameters whose actual values will be entered by the user.

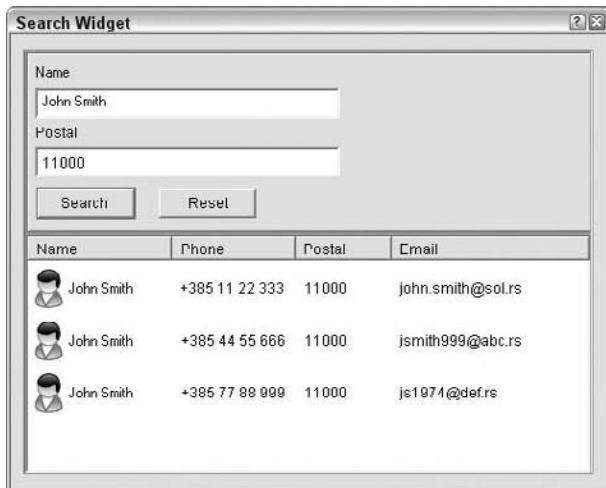


Figure 16-8: An example of a Widget for a Search Component

A Widget for this Component is a composite Widget that has the following:

- ❑ Input controls of the appropriate kind for the parameters of the query. The user enters the values for the search parameters through these input controls.

- ❑ The usual buttons (“Search” to perform the search, or “Reset” to clear the input controls).
- ❑ The table to display the result of the search (that is, the tuples returned by the query). If applicable (for example, in a Web-based GUI), the Widget can provide the usual pagination functionality, where only one page of the (potentially huge) result set is displayed, while the others can be accessed by navigation buttons (go to the first/last, previous/next page).

One or more of the resulting items can be selected by the user and are then provided to the output pin of the Widget. Note that all the developer must do to implement a search is to place an instance of this Component at the desired place and specify its properties (most notably, the query). The Component and the Widgets do the rest (for example, creation of the input controls and synchronization of the input controls with the result table).

An *Object Identification Component* provides a means to identify an object of the specified class by its identifier. The name of the class and the name of its identifier (as an attribute of the class) are specified in the textual attributes of the Component. A Widget for this Component is a simple input control of the kind appropriate for the type of the identifier. When the user enters the value of the identifier, the Widget identifies the object of the class and provides a reference to that object on its output pin. Optionally, the Widget can allow the user to enter partial values of identifiers. If the user enters a partial value, the Widget can display (for example, in a list that pops up) the subset of objects of that class that have that partial value of the identifier, and let the user select one of these objects.

### Section Summary

- ❑ A *Search Component* is a specialization of a Query Result Component because its Widget renders a collection of tuples returned as the result of a search initiated by the user. The search is performed using the parameterized query specified by the Component, where the actual parameters are entered by the user.
- ❑ An *Object Identification Component* provides a means to identify an object of the specified class by its identifier whose value is entered by the user.

## ***Creational Components***

Very often, the application requires that a new object be created and that some of its attributes be immediately set to the values provided by the user. This is typical when bulk input should be performed efficiently using the keyboard. For example, for a bulk input of products in an empty information system, it would be convenient to enter (in a series) the code of the product (as its identifier — that is, as its attribute value), its name, and short description, using only the keyboard but not the mouse (as its use would slow down the input).

For such a purpose, there is the *Create Object Component*. A Create Object Component is configured with a class for creating new objects and a list of attributes whose values will be set upon creation from the user’s input. All of these are specified through textual attributes of the Component. At run-time, a Widget of this Component is a composition of input controls, one for each of the configured attributes, for entering values of the attributes. When an associated button or a key (for example, Enter) is pressed, a new object

## Part III: Concepts

---

of the specified class will be created, and its attributes will then be set to the entered values. Figure 16-9a shows an example of such a Widget.



(a)

**Figure 16-9: An example of a Widget for a  
(a) Create Object Component**

This similarly holds true for two other Components, *Create Object And Link To Object Component* and *Create Object And Link To Two Objects Component*. Their Widgets do the same thing as those of the Create Object Component, except that they also link the newly created object to one or two other objects. The other objects are identified by the values entered by the user for the identifier specified in the Component.

For example, a bulk input of employees of departments in a company could be efficiently done by entering the code for a department (which is the value of the identifier of that object) and the name and some other attributes of the employee, when a new employee is created and immediately assigned (that is, linked) to the identified department, while his/her name attribute will be set to the given value. Figure 16-9b shows an example of such a Widget.



(b)

**Figure 16-9: and (b) Create Object And Link To  
Object Component.**

This similarly holds true for the Create Object And Link To Two Objects Component. An item in a purchase order can be created by entering the order number (as its identifier), the product code (as its identifier), and the ordered quantity. The new object of `OrderItem` will be created and linked to the identified order and product, and its quantity attribute will be set to the given value.

An optional mode of work of the Widgets for these Components is that (any of) the object(s) to which the newly created object will be linked is not identified by the entered identifier value, but is provided on the input pin of the Widget. For example, an order item is usually entered within the context of a selected order (for example, within its dialog or page). In that case, the order is provided on the input pin of the Widget, while the user identifies only the product of the order item by entering its code. However, the Widget does basically the same operation. The difference is only in the way the object to which the new object will be linked is provided.

The newly created object is supplied on the output pin of the Widgets for all these Components.

## Section Summary

- ❑ A *Create Object Component* is configured with a class for creating new objects, and a list of attributes whose values will be set upon creation of the objects from the user's input.
- ❑ Widgets of the *Create Object And Link To Object* and *Create Object And Link To Two Objects* Components do the same as those of Create Object Component, except that they also link the newly created object to one or two other objects, identified by the specified identifier values or provided on the input pin(s).

## Container Components

There is an entire family of Components that are simple containers of other Components. The *Container Component* is their abstract generalization. Concrete specializations include (but are not limited to) the following:

- ❑ **Tile Component** — A Component that contains a collection of other Components whose Widgets are tiled horizontally or vertically within the Widget of the container Tile Component, depending on the configuration of the Tile Component. Advanced implementations could support other built-in or user-defined layouts specified for the Tile Component.
- ❑ **Dialog Component** — A Component whose Widgets represent pop-up dialogs. A specialization of this Component, the *Element Dialog Component*, is a dialog opened for an Element provided on its input pin.
- ❑ **Dialog Tab Component** — A Component whose Widgets represent tabs that can be laid on a dialog or panel. A specialization of this Component is the *Element Dialog Tab Component*, whose Widgets are tabs opened for concrete Elements provided on their input pins. A further specialization of the Element Dialog Tab Component is the *Object Tab Component*, which represents a tab opened for an object of a class on its input pin. Its specializations include the following:
  - ❑ **Object General Tab Component** — A standard, generic dialog tab that renders the generic, technical information about the object, such as its internal system identifier (the so-called "technical ID"), its class, and so on.
  - ❑ **Object Attributes Tab Component** — A standard, generic dialog tab that displays Widgets for editing all attribute values of the object.
  - ❑ **Object Association Ends Tab Component** — A standard, generic dialog tab that displays Widgets for editing all association ends of the object.

These standard Components are used in the generic run-time environment of OOIS UML.

## Miscellaneous Components

A *Command Component* is a Component that has one input pin and an optional associated Command. Its Widgets can be invisible, or can be rendered as buttons, hyperlinks, or similar GUI controls that allow the user to issue a command to the system. The Widget of the Component accepts an Element on its input pin, and applies the specified Command on that Element, whenever the value on the input pin is changed, or when it is triggered by an explicit user request (for example, a button press or hyperlink).

## Part III: Concepts

---

click), depending on the value of a Boolean attribute of the Component. The Command can then perform a domain-specific scenario, or do some basic navigation within the GUI — open a dialog or a Web page. Preconfigured prototypes of built-in Commands may be responsible for such GUI navigation.

As a final example of flexibility of the proposed concepts, let's look at a sophisticated Hot Spot Component and its Widgets, as shown in Figure 16-10. Their purpose is to support the navigation through the object space by sensitive regions within pictorial presentations of objects.

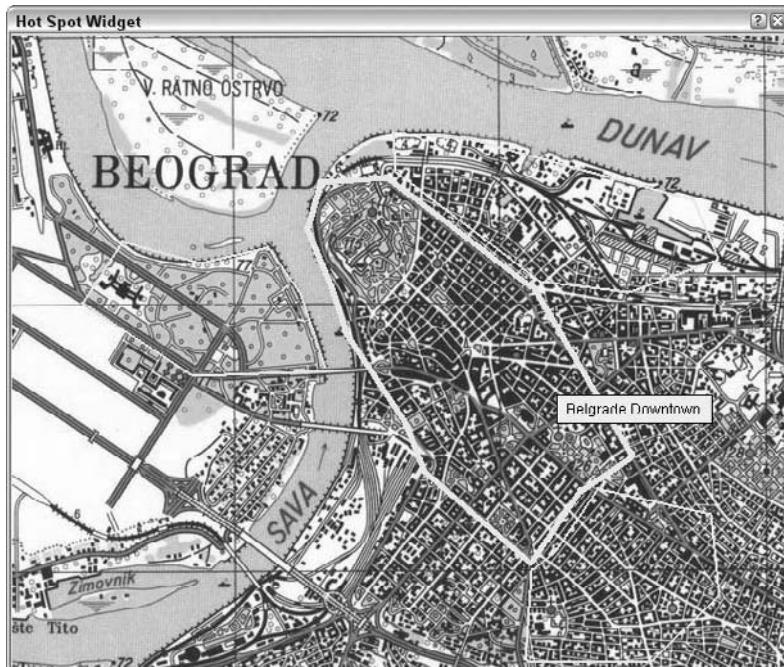


Figure 16-10: Example of a Hot Spot Widget

For example, consider a system that manages some information related to geographical regions (country, state, region, municipality, city, and so on.). The Hot Spot Component can provide a pictorial view to a certain geographical region (that is, show its map), while its sub-regions are rendered as hot spots sensitive to mouse clicks. Whenever the user clicks on a hot spot, the Widget can show the map of the selected region. The Widget can provide other similar behavior, such as showing a tool tip when the mouseover a hot spot, or issuing a command for the object represented by a hot spot, and so on.

Many other examples can illustrate the use of this Component. For example, a scheme of a composite device can be shown in a picture, when its subcomponents are represented by hot spots. A tabular map can represent a schedule of some activities or tasks allocated to certain resources, such as departments, people, or classrooms, and so on.

The appearance and behavior of this Component may look the same as that of the usual HTML picture tag with hot spots (also known as *maps*). Although that HTML tag can be used for its implementation, the level of abstraction of the Hot Spot Component is much higher. It is directly connected to the underlying

object space, so its use requires much less coding (actually, no coding at all) to provide sophisticated behavior.

To demonstrate how this Component and its Widgets work, let's briefly look at their main features. The variants of this Component and its functionality and configurability are virtually endless, and each particular framework can implement its own.

The Widget of this Component has one input pin accepting an object of a class. The object provided on the input pin is the currently shown object. The picture rendered for the currently shown object, referred to as the master picture, is taken from the value of an attribute of type `Picture` of that object. The name of the attribute is provided in a textual attribute of the Component.

The hot spots rendered in the master picture represent the objects linked to the currently shown object over an association end instance. The name of that association end is provided in another textual attribute of the Component. The definitions of the hot spots are taken from the attribute values of the linked objects, whereby the name of the attribute for those values is provided in another textual attribute of the Component. The way the hot spots are defined and the definitions are encoded in the attribute values is left to the implementation. The definition can be stored in an attribute of a specific abstract data type, or encoded in a textual attribute. The method for defining such a hot spot is also left to the implementation. The implementation can use the WYSIWYG approach to let the user define (using the mouse) the figures for the hot spots. The set of figures supported for defining hot spots is also up to the implementation. Anyhow, several hot spots referring to the same linked object can exist in the same master picture.

Finally, the behavior exhibited by the Widget is also left to the implementation. What is expected from the implementation is to provide the object represented by the selected hot spot to the output pin of the Widget. In that way, other Widgets whose input pins are bound to that pin can provide different behavior — apply a Command on the selected object, open a specification dialog or a Web page, and so on. If the output pin of the Widget is bound to the input pin of the same Widget, and the linked objects are of the same type as the current object, the Widget will simply render the selected linked object. In addition, the Widget can render the tool tip on mouseover held over a hot spot, a label across the hot spot, and other features, all obtained from the style configuration for the objects represented by the hot spots within the current GUI Context. All these features can be configurable through the attributes of the Component.

Many other sophisticated Components and Widgets can be provided by a concrete implementation using the described concepts.

### Section Summary

- ❑ A *Command Component* is a Component that accepts an Element on its input pin, and applies the specified Command on that Element, whenever the value on the input pin is changed, or whenever it is triggered by an explicit user request (for example, a button press or hyperlink click).
- ❑ Many other sophisticated Components and Widgets can be provided by a concrete implementation using the described concepts.

# Application Architecture

As a conclusion to this part of the book, let's look at a possible logical architecture of applications built using the OOIS UML approach. This particular architecture has proven in practice to be clear, simple, robust, flexible, stable, and scalable, providing a clear separation of concerns and good encapsulation. It follows the state-of-the art results presented in other software engineering literature, but is adapted to OOIS UML.

The architecture is schematically shown in Figure 16-11. It has three tiers with well-defined responsibilities and contents. Each tier is part of the OOIS UML model of the application, embodied in one or more UML packages. As a result, the tiers provide just a logical separation of packages, while the architecture actually suggests how to organize the elements of a UML model into cohesive and loosely coupled packages, and has no other semantic implications. The architecture does not deal with the deployment strategy of executable components, which is deemed a technical detail related to the selected implementation infrastructure that is out of the profile's scope.

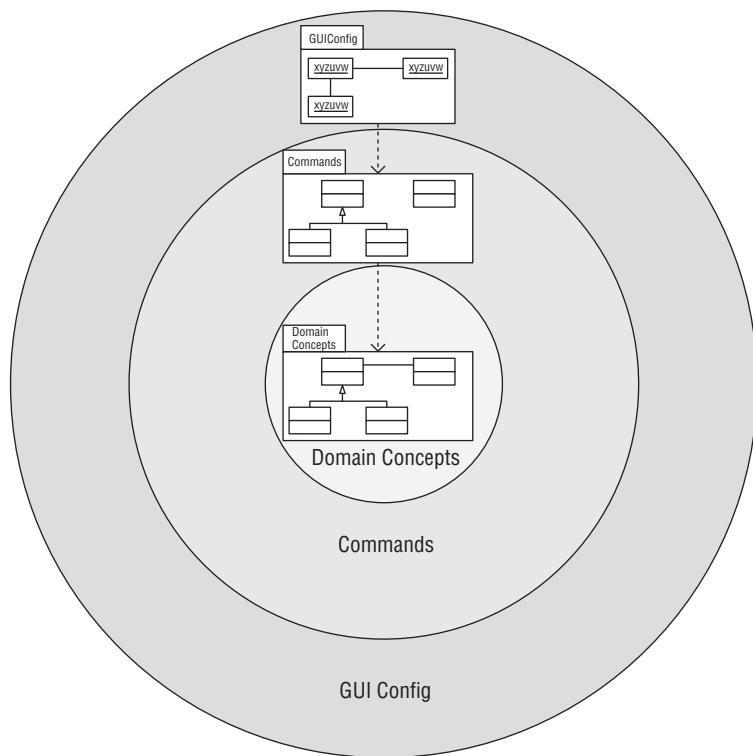


Figure 16-11: The suggested three-tier logical architecture of applications

The lower, core tier contains the structural and behavioral model of the problem domain. The structural model consists of classes, their properties, and their relationships that conceptualize the key abstractions from the problem domain. The operations of these classes should be designed to provide the services of the classes or their objects. Most often, these operations perform simpler structural manipulations (such as creations or modifications) upon the local structure in the close neighborhood of the host object.

They should be designed to represent elementary behavioral units, so that they can be reused in different scenarios conducted by the commands in the middle tier. Very often, such operations are polymorphic and sometimes recursive, when they recursively traverse some structure. In addition, the behavioral model in this tier consists of state machines and other possible behavioral features of classes. In some other OO design methods, the classes of this tier are referred to as the “entity classes.”

The middle tier consists of domain-specific commands (as classes, while the built-in commands are supposed to exist in a model library), along with the configuration of prototypes of built-in and domain-specific commands, provided by creational object structures or other means. The commands correspond to the application’s functionalities activated from the GUI or other systems, and their responsibility is to conduct the scenarios that implement those functionalities.

In general, the implementations of commands (that is, their `execute` methods) rely on the structure and behavior of the lower tier, but generally have a broader scope than operations of domain classes — they embody algorithms and interactions among objects of the domain classes, while the steps of these algorithms and interaction scenarios rely on the objects’ operations as elementary behavioral units possibly reused in many such algorithms and scenarios. In some other OO design methods, the classes of this tier are referred to as the “control classes,” while in the proposed architecture they are more specifically commands.

The upper tier provides the GUI configuration (that is, the GUI Style and Component layers). It consists of creational object structures and other parts of the model that specify creation of configuration objects that drive the GUI at run-time.

At run-time, each of these tiers has its counterpart in the object space. The instances of the classes in the domain tier constitute the domain object space and generally survive the execution of their operations. The instantiation of the middle tier consists of prototype and executed commands. If the executed commands survive completion of their own execution (which can be a configurable option of a command), they inherently represent the execution log. The instantiation of the GUI layer consists of GUI Style objects and Components that are predominantly static, and Widgets that are instantiated and destroyed dynamically during the interaction with users.

Although this architecture resembles the classical three-tier architecture, it has some significant differences. In the classical three-tier architecture, the lower layer consists of a database and the layer of entity objects in the target implementation language, responsible for object-to-relational mapping (also known as “data access objects,” or DAOs). Most often, the classes in that tier end up in providing only structure, without any (significant) behavior. Instead, the entire behavior is packed in the middle tier, usually referred to as the *business logic* tier. The developer is very often aware of (and has to take some care of) the object-to-relational mapping, and must provide certain configuration mapping and maintenance operations required by the framework. Some typical examples include the necessity of explicit modeling of technical identifiers not needed by the problem domain, but used as primary keys in the relational database, fields that provide type discriminators, or explicit calls of operations that load objects from the database to memory or vice versa.

Apart from unnecessarily increasing the developer’s responsibility and effort, such design often leads to an exhibition of the so-called *Anemic Domain Model anti-pattern*, first described by Martin Fowler. *Anti-patterns*, also called *pitfalls*, as opposed to design patterns, are commonly reinvented bad solutions to problems. They are studied as a category so they can be avoided in the future, and so that their instances may be recognized when investigating badly designed systems. The Anemic Domain Model anti-pattern represents the use of a domain model without any business logic, which is not in accordance with object orientation, because each object should embody both structure and behavior [Wiki].

## Part III: Concepts

---

On the other hand, OOIS UML and the proposed architecture do not suffer from these deficiencies. First, the domain layer does not reveal any unnecessary implementation details to the developer — the developer does not deal with any object-to-relational mapping or persistence infrastructure and maintenance. In addition, the architecture does not exhibit the Anemic Domain Model anti-pattern, because the domain model embodies both structure and behavior. The business logic is actually split among the domain model and the command tier, according to the described separation of concerns. Finally, the upper layer also deals with the GUI, although at a higher level of abstraction — not with its implementation (coding), at least not predominantly, but with its configuration (structure).

It should be emphasized that the described architecture is purely logical — it does not imply any physical architecture, whereby the physical software architecture assumes the physical components of the software artifacts (executables, dynamic linking libraries, and other deployable components) and their deployment on physical nodes (that is, hardware). The way parts of the logical model are packaged up into physical components and deployed on nodes, as well as on which processing nodes the concurrent processes run, is a concern of the physical architecture and the selected implementation infrastructure, which are out of the profile's scope. This is independent of the proposed logical architecture.

In the simplest case, the entire information system can consist of a single, common object space, while the behaviors are executed in concurrent threads of control as described in the OOIS UML concurrency model. The concurrent threads of control can be physically executed either by the (thick) client machines, or by the processes or threads deployed on one or more application servers, depending on the selected physical architecture. The concurrent threads of control access the common object space (that is, the structure), and synchronize by the implemented isolation mechanisms. In essence, this is known as the *shared variable* model of interprocess communication.

In some more demanding cases, the system may consist of separate subsystems that do *not* share a common object space, but each of the subsystems has its own, isolated object space and provides *services* to other systems. The subsystems communicate (that is, invoke services of one another) by a *message passing* mechanism — for example, by a *remote procedure call* (RPC). This approach is also known as the *Service-Oriented Architecture* (SOA). In that case, each of the subsystems can have its internal architecture as described earlier, and provide an interface to other subsystems. The upper layer of the logical architecture of such a subsystem can provide that interface.

One proven design in such cases is that the upper layer provides a very thin, business logic-free, stateless, structure-free set of utility classes having only static operations. These operations are aimed at remote invocation from other subsystems, and represent services of the subsystem offered to other subsystems and a kind of a “capsule” around the interior of the considered subsystem. The methods of these operations should have very limited responsibility — they should just promote every invocation into a command that supports that service. The middle layer should then provide the commands for the services and the rest of the architecture remains the same. In addition, the upper layer can adapt the interface to the infrastructural technology used for RPC (for example, Web services based on a standard protocol such as SOAP), do marshaling and unmarshaling of parameters (if necessary), and so on.

Although such service-oriented, distributed systems with communicating subsystems without a shared object space are very popular and compelling nowadays, system architects should be very careful when designing them, as they can have a number of pitfalls. I have seen quite a few poorly designed distributed systems that did not use a shared object space for no compelling reason. One of the pitfalls is the complexity of the design. The communication between the subsystems may become too complex and difficult to understand and maintain. In addition, the design and implementation of interfaces for such communication can become very time-consuming and inflexible because an operation and a command should

be developed for each particular service of a subsystem. For complex accesses to the object space (for example, for querying), this is usually much more difficult to implement than to access the object space directly by the available concepts (for example, by queries). Finally, such design could easily lead to poor scalability. If synchronous communication is used extensively without strong reasons, it may degrade the performance of the system because of unnecessary blockings and sequencing. Allocating too many responsibilities and too heavy workload to a single subsystem from which the others require services easily leads to bottlenecks.

On the other hand, the shared object space architecture usually ensures better scalability because the processes run concurrently while accessing the common object space. They usually access different parts of that space without causing conflicts and congestion. In addition, DBMSs and servers that implement the shared object space are generally built to ensure good scalability by sufficient internal concurrency.

In short, the shared object space model should be used whenever possible, while the distributed model should be used very carefully and only with strong reasons. Some of the reasons for building distributed systems include the following:

- ❑ The shared object space cannot be physically implemented, or is very costly and unreasonable to implement because of the inherent distribution of the system.
- ❑ The subsystems should be deployed on separate hardware nodes with specialized responsibilities and specialized hardware or other resources.
- ❑ The entire system consists of heterogeneous subsystems, while some or all of these subsystems are developed in different technologies or standards, or by loosely coupled development teams or vendors, or some of the subsystems are legacy or closed for changes, and so on.
- ❑ There are requirements for strong encapsulation or security for some or all subsystems.

Anyway, if subsystems with too complicated and unclear interfaces or communication emerge in a distributed design, this is a good signal for considering introduction of a shared object space.

### Section Summary

- ❑ The proposed logical architecture of the system has three tiers with well-defined responsibilities and clear separation of concerns. Each tier is part of the OOIS UML model of the application, embodied in one or more packages.
- ❑ The lower, core tier contains the structural and behavioral model of the problem domain.
- ❑ The middle tier consists of domain-specific commands, along with the configuration of prototypes of built-in and domain-specific commands (provided by creational object structures or other means).
- ❑ The upper tier provides the GUI configuration (that is, the GUI Style and Component layers).

*Continued*

## Part III: Concepts

---

- ❑ The physical architecture of the system can be the following:
  - ❑ With a shared object space accessed by concurrent threads of control
  - ❑ Distributed, with isolated subsystems without a common object space, communicating over a message passing mechanism
- ❑ The shared object space physical architecture is simpler to implement and usually ensures better scalability. The distributed architecture should be chosen only with strong reasons and care.

# Part IV

# Method

**Chapter 17:** About the Method

**Chapter 18:** Conceptual Modeling

**Chapter 19:** Modeling Functional Requirements



# 17

## About the Method

To wrap up the profile into a method, this chapter discusses the activities conducted during development of the system and their characteristics. Rather than prescribing strict rules, it provides development guidelines and describes best practices. In particular, this chapter focuses on the activities of requirements engineering.

### Activities and Artifacts

A modeling language is simply a vehicle for modeling. It only specifies concepts that can be used in modeling, as well as their semantics and rules for forming correct models. However, the language itself does not give any direction as to *how* to specify requirements, create models, or develop a system. In other words, the language does not define development activities, their artifacts, and guidelines. They, in conjunction with the used modeling language, form the *method*.

### Development Activities and Artifacts

In iterative software engineering processes, all development *activities* (such as those concerning business modeling, requirements engineering, analysis, design, implementation, testing, and deployment) are applied iteratively, and may (and usually do) occur in parallel, but at different levels of intensity during different stages of the project.<sup>1</sup> Thus, the requirements specification, for example, is not something that is finalized at the end of the first project stages, but rather it keeps getting updated throughout development. The implementation activities are much more intensive during the middle project stages than in the very early stages. The deployment activities are much more intensive in the later stages of the project, although they are (less intensively) conducted in the earlier stages too, and so on.

---

<sup>1</sup>The Rational Unified Process (RUP) calls these top-level project stages *phases*, such as Inception, Elaboration, Construction, and Transition.

## Part IV: Method

---

The following table summarizes the main groups of activities that are central to this discussion and their major artifacts.

| Activities               | Artifacts                                                     |
|--------------------------|---------------------------------------------------------------|
| Requirements engineering | Requirements specification document and UML analysis model    |
| Design                   | UML design model                                              |
| Implementation           | Implementation forms (code, database, components, and others) |
| Testing                  | Testing plans, models, stubs, programs, and reports           |

Requirements engineering encompasses activities aimed at capturing, analyzing, and specifying requirements. Requirements capturing includes interviewing stakeholders and collecting their written or oral requirements. Requirements analysis and specification include intellectual activities aimed at understanding, structuring, formalizing, and formulating the collected requirements in a complete, consistent, and unambiguous form. The result of these activities should consist of two artifacts — a textual requirements specification and a UML analysis model. A more detailed description of these activities and artifacts will be given in the remainder of this chapter.

Design activities are oriented toward extending the UML analysis model with design-related model elements, completing it with details, and formalizing its informal parts, so that a UML design model is obtained. The UML design model is a complete, unambiguous, and formal specification of the software under construction from which the running system can be achieved with all its features and functionalities planned for a certain iteration. The details about the relationship between the UML analysis and design models are given in the next section.

Implementation activities are focused on the level of detail that is necessary to get a fully functional and complete implementation from the design model. They may include coding of methods in the detail-level implementation language. With a mature methodology and strong tool support, the rest of the task can be done completely or almost completely automatically (as described earlier in this book). Implementation may also include other activities such as developing components that are outside the scope of the model, components that cannot be modeled (or are impractical to model) in UML, or interfaces and components that interoperate with outer systems and legacy systems, import or export data in external formats, and so on.

Testing is particularly important for software development and deserves separate discussion and training. Its purpose is to discover errors in software. Many good books on software testing are available for more information about testing. Testing activities are outside the scope of this book, and they are simply mentioned as an important and unavoidable task in every development.

In my experience in applying the OOIS UML method for many years in different projects of various sizes and complexities, the method described here brings tremendous or less significant improvements in different activities of development. First, it does not bring such dramatic benefits during requirements engineering because this task must cope with the fundamental, inherent complexity of the problem domain. In other words, the effort required for capturing, understanding, structuring, and formalizing requirements and conceptualizing the system seems to be, to a great extent, independent of the applied method. However, the OOIS UML method still brings some considerable benefits, even in these activities, because the

specification of requirements can be significantly easier and less demanding than in traditional approaches. This is because the specification uses a modeling language of a higher level of abstraction (UML), which is more expressive than some other older languages, thus requiring fewer “words” for expressing the same thoughts. In addition, some techniques described later in this book can reduce the volume of the specifications because many parts of the system can be implicitly assumed or defined in a generic way.

Design, and especially implementation, notably gain from the OOIS UML approach because of the highly abstract and expressive concepts described earlier in this book, as well as because of the general advantages of development based on executable models. Actually, the main improvements enabled by the OOIS UML approach are in these activities because it eliminates much of the accidental complexity introduced by the traditional approaches.

Finally, testing also gains from the approach, although maybe not so much as design and implementation. The benefit still exists because of many aspects of the approach. Most of design decisions are localized (for example, in GUI configuration) and, thus, reduce the number of tests. A major part of the system’s functionality is obtained generically, and, thus, does not need testing as long as the run-time environment is considered to be error-free. The volume of design decisions and statements (and potential errors) made by humans is much reduced compared to the traditional approaches, too.

Note that the previous parts of the book described in detail the concepts that can be used in analysis and design, and how an implementation can be obtained from a UML design model. Furthermore, as noted previously, testing is out of the scope of this book. Consequently, the main remaining issue is how to obtain a UML model from the initial requirements. This is why the remaining part of this book focuses on the requirements engineering activities, as the most challenging, sophisticated, and fuzzy task.

### Section Summary

- ❑ Requirements engineering encompasses activities aimed at capturing, analyzing, and specifying requirements.
- ❑ During design, the UML analysis model is extended with design-related model elements, completed with details, and formalized, so that the UML design model is obtained.
- ❑ The task of implementation is to transform the UML design model into a fully functional implementation.
- ❑ The aim of testing is to discover errors in software.

## ***UML Analysis and Design Models***

The UML analysis model is a product of requirements analysis and represents a semi-formal specification of the requirements and the problem domain. In other words, the UML analysis model reflects the modeler’s understanding of the problem domain and the requirements.

Design activities take the analysis model as its starting point and add details to it. The UML design model is a product of design activities, when there is a better understanding of how to meet the

## Part IV: Method

---

requirements and solve the problem. It reflects the modeler's solution to the requirements. In its final stage, it is completely formal and executable, and represents the specification of the software system itself. In other words, all the parts from the analysis model that had informal meaning (and were interpreted by humans in the way that satisfy the requirements) are transformed into fully formal and executable elements of the design model, so that the system specified by these elements performs as required.

More precisely, the analysis model may use all concepts from standard UML that have informal meaning or imprecise semantics and, thus, are not executable (it typically does use many such concepts). In addition, many parts of the analysis model can be unspecified or incomplete. However, this does not prevent the entire model from being executable in only those parts that have executable semantics. In fact, executable models are one of the best ways of validating requirements early enough to prevent mistakes and misunderstandings, because they allow more direct communication with users. All that is required is a tool that allows execution of incompletely specified models (usually with some human-assisted guidance). The capability to execute incomplete models should be one of the major benefits of model-driven development relative to code-oriented methods because programming languages are far too unforgiving to allow this capability.

Of course, such a tool will simply ignore all informal elements from the model, while those formal (but incompletely specified) elements may be interpreted in some default way, or the tool may require some human assistance to interpret them. This means, among other things, the following:

- ❑ Attributes do not need to have names and types, or their types do not need to be classifiers that exist in the model. In such cases, the tool can provide some automatically generated names and default types and ways of manipulation, such as a simple text without any validation.
- ❑ The modifiers of properties (such as multiplicity, ordering, uniqueness, and so on) do not need to be specified. The tool can simply assume their defaults.
- ❑ Constraints do not have to have formal semantics. They do not need to be specified in a formal language, but can be given in a spoken language instead. The tool will simply ignore them, while humans may interpret them as constraints that will be ensured by the system when it becomes fully implemented. At this stage, they may be treated as annotations that will be interpreted formally in the design model.
- ❑ Operations do not need to have fully specified parameters or methods. If the parameters are not fully specified, their types and values are set to defaults. If the methods are not explicitly specified, they are assumed to be empty (that is, do nothing, as stubs). During the execution of such operations, the modeler can be asked to provide their output parameters, or even demonstrate their effects on the object space in the execution environment. A more sophisticated tool can record the modeler's settings of output parameters (or the demonstration of a stub operation) and then repeat its behavior in later calls, and so on.
- ❑ State machines (in particular, events, guards, and actions of their transitions) do not have to be fully specified. The tool may inject triggers to demonstrate their behavior, or require human guidance in cases of unspecified guards or triggers for transitions going out from the same state.

The task of design is to transform all informal or incomplete elements from the analysis model that have to be fully implemented in the selected iteration, into fully specified, well formed, and formal elements in the design model. In addition, design is focused on the solution, so it adds other elements to the model that are necessary for the implementation. For example, the design model may contain classes and relationships that are not part of the domain's conceptual model.

To summarize, then, these two models focus on different concerns and have different extents and levels of details:

- ❑ The analysis model may have many informal parts. That is, it may contain model elements that have no executable semantics or other effects on the running system, such as use cases, high-level interactions, or informal constraints. The design model has fewer such parts, or such parts still have no effect on the generated system, but are simply ignored by the tool and serve for documentation purposes.
- ❑ The analysis model consists mainly of the definition of the domain's conceptual (structural) model and use cases. Actually, it consists of the domain layer of the architecture. The design model adds design and implementation-related structure and behavior.
- ❑ In most cases, the structural domain model has very few operations of high-level and state machines for the domain-specific behavior. The operations often do not have specified methods. The state machines often do not have fully specified triggers, guards, or actions for all transitions. The design model may introduce many detail-level operations, even for the domain classes. The methods of non-abstract operations in the design phase can (but still need not) be fully defined. This similarly holds true for the state machines.
- ❑ The analysis model typically does not deal with commands, queries, GUI configuration, and other design-level details. They are added in the design model.

In fact, the terms “analysis model” and “design model” refer to the same model at different stages of evolution, reflecting different levels of understanding, focusing on different concerns, and having different amounts of formal elements and levels of detail. They are built using the same language, UML, whereby some elements that are covered by OOIS UML have formal interpretation as defined in that profile, while the others have the same general (often imprecise and non-executable) meaning as in standard UML. Of course, the boundary between the two models is not sharp and the overall approach of using the same language and semantics of formal elements in both guarantees a smooth transition from one to the other.

### Section Summary

- ❑ The analysis model is a product of requirements analysis, and represents a semi-formal specification of requirements and the problem domain. It may have many informal parts and incompletely specified elements.
- ❑ The design model is a product of the design phase and an evolution of the analysis model, when there is a better understanding of how to meet the requirements and solve the problem. It has much less or no informal or incomplete parts, or such parts are simply ignored.
- ❑ Although the analysis and design models reflect different levels of understanding, focus on different concerns, and have different amounts of formal elements and levels of detail, they represent the same model at different stages of evolution, in the same language, both being executable.
- ❑ Executable models are one of the best ways of validating requirements early enough to prevent mistakes and misunderstandings. All that is required is a tool that allows execution of incompletely specified models (usually with some human-assisted guidance).

## Requirements Engineering

The requirements engineering activities are the most challenging activities because they have a very sophisticated task: to start from often informal, unstructured, unclear, ambiguous, incomplete users' requirements, and transform them into a well-structured, unambiguous, semi-formal, and complete requirements specification and analysis model. These activities heavily depend on the skills, knowledge, and experience of analysts, and, unfortunately, to a significant extent on analysts talent for abstract thinking, inventiveness, and pragmatism. Fortunately, however, software engineering has matured enough to provide a useful know-how that can help in these activities, and can reduce their dependence on human talents.

### Activities and Artifacts of Requirements Engineering

Requirements engineering includes the following activities:

- ❑ **Requirements capturing** — Collecting requirements from written specifications or interviews
- ❑ **Requirements analysis** — The sophisticated mental process focused on conceptualizing, structuring, and formalizing the collected requirements
- ❑ **Requirements specification** — Building coherent, well-structured, and unambiguous artifacts — the requirements specification document and the UML analysis model

In most simple cases, the initial requirements are provided by the initiators of the development, which are typically users or customers. (Note that there is a difference between *customers* and *end users* of a system in many cases.) However, there are often other stakeholders in the system who have requirements, including experts from the domain (who have knowledge about the business and problem domain under consideration), implementers (who may have feasibility and practicality requirements), testers (who have testability requirements), documenters (with documentability requirements), government regulators (with regulatory requirements), usability experts (with usability requirements), system operators/maintainers (with operational requirements), and others. Their requirements are often in conflict, which is one of the reasons why it is so important and difficult to define system requirements.

The activity of requirements capturing may start from a written specification of requirements provided by the interested stakeholders. However, in many cases, such specifications do not exist at all. Even when they exist, they are most often unclear, incomplete, and full of ambiguities or contradictions. Therefore, interviews are conducted between the parties involved. These interviews may also become brainstorming sessions, where all parties work and think together about the best way to design a useful system efficiently. It is much better for the parties to establish such a brainstorming atmosphere during interviews, where all stakeholders work together as partners, instead of keeping a negotiating attitude of a supplier toward a customer (one of the principles of agile development reads, "Customer collaboration over contract negotiation").

Because there can be many parties that pose requirements, these parties will be referred to simply and abstractly as the *users*, wittingly sacrificing precision for brevity. The supplier's party is represented by system architects, system analysts, or modelers. This party will be abstractly referred to as the *system analysts*.

In my experience, system analysts can work with several kinds of users.

The first are the people who do not have high skills for formal thinking and expressing themselves precisely. It is rather difficult to capture requirements in such circumstances. System analysts must have significant skills, knowledge, experience, and pragmatism to cope with the situations of obscure (and often contradictory or volatile) requirements.

Unfortunately, such circumstances often tend to create a sense of superiority in analysts, who feel that the major problem in requirements capturing is the user's lack of understanding of the implementation technology, rather than their own lack of understanding of the problem domain. In my experience, if anything, the latter is by far the bigger issue — which is why it so often happens that the system that is delivered is not the one that solves the user's problem.

The first and primary responsibility of an analyst is to learn about the problem domain as much as possible, before any detailed discussions about specific requirements are initiated. In addition, the first thing that must be established and shown by the analyst is respect for the users and their problems — even though they may not be technically literate. One simple and practical piece of advice is that the analyst should always start with listening carefully to the users explanations without interruption, and hold off on offering opinions until at least some initial understanding of the problem domain and a big picture of the envisioned system is formed.

The second kind of users are those who do not have software engineering skills and background, do not know UML or other software specification formalisms, but do have a talent for formal thinking and expressing themselves precisely. Usually, those are the people with some technical background or significant experience in using computer applications (which have taught them to think precisely, formally, and in an abstract way). It is much easier to capture requirements from these people because they will more likely express their thoughts in a structured and consistent way. They might just need the analyst's help in doing that. They may also try to learn the basics of UML over time, or at least to understand informally the semantics of the UML analysis diagrams sketched by the analyst. If this is not the case, the analyst can try to encourage such users to express or sketch their thoughts in any precise or formal notation, even in an ad-hoc one that can achieve a common unambiguous understanding (for example, a block diagram, a flowchart, a dataflow diagram, and so on). The task of the analyst is then to transform the requirements stated in that (semi-) formal notation into UML.

Finally, the third kind of users comes from the software engineering world. They may be with or without knowledge of UML or another software design language or notation. They will often offer their own requirements specification as an input to the process. Of course, it may be easiest to capture requirements from that kind of user, but they seem to be least frequent in practice. They play the role of those who pose requirements only if they themselves cannot develop the software because, for example, they do not have enough resources or they want to outsource the development. However, a severe problem with this category may occur because they often specify solutions rather than requirements. This can be quite problematic because they may not have the necessary understanding of the specific technology being brought to bear on the problem. Also, they may actually provide the wrong requirements because they are more technology-oriented than solution-oriented.

The requirements engineering activities are not carried out sequentially, and they do not have strict boundaries. Instead, they are interleaved and performed iteratively. Requirements are captured during interviews or brainstorming sessions with the users. Then, the system analyst analyzes the requirements and makes a preliminary textual requirements specification and UML analysis model, and presents the former to the users. The users review the specification and accept it or require a revision.

## Part IV: Method

---

If a revision is required, the activities are repeated in a new iteration. In my experience, this procedure is usually completed after two (or at most three) iterations.

A significant support to the specification of requirements can often be a quick and incomplete (but executable) analysis model that may be regarded as the very early prototype of the system. As discussed later in more detail, it may help with gaining a common understanding and reducing risks of misunderstandings.

Following are the artifacts of this phase:

- An informal, but precise, complete, consistent, and unambiguous textual requirements specification
- The UML analysis model of the system

Both of these artifacts are needed for several reasons. The UML analysis model is needed because it is (semi-) formal and precise, so it is the starting point of software development and the input for the next (design) activities (the software product will emerge from that artifact). Because of the expressiveness of UML, it is much easier for analysts to express their thoughts in UML diagrams using a modeling tool than in a written language (text). In addition, it is much more convenient to maintain (modify and extend) a UML model than a written text. Finally, as already mentioned, it may produce a quick and early demonstrational prototype of the system in some cases. This is why a UML analysis model with sketchy diagrams is much more appropriate for the early phases of requirements analysis and specification.

On the other hand, the textual requirements specification is needed for several reasons. First, many users cannot read UML. This is why the central part of the requirements specification document uses descriptions in natural language illustrated with UML diagrams. Actually, the natural language descriptions may simply interpret (“read”) the UML diagrams. In that way, the users are assisted in understanding both the diagrams and the descriptions, and are indirectly introduced to the basic meaning of UML. Second, the requirements specification document can contain some requirements that cannot be expressed in UML, as described later in this chapter. Finally, the textual requirements specification has another role — to set up the context to its readers, most notably to the developers who will develop the software from the UML model, and to introduce them quickly to the problem domain.

Of course, these two artifacts should be consistent, which introduces the problem of keeping them synchronized when the requirements are changed. Although some help can be provided by tools (for example, to update the UML diagrams incorporated in the requirements specification document automatically whenever the UML analysis model is changed), there will always be parts (for example, glue text) written in a natural language that cannot be modified automatically. Fortunately, in practice, such parts most often do not carry significant meaning, and are simply helpers to readers, so their consistency is less relevant. In my experience, although caution is necessary, this is not a big issue in practice.

### Section Summary

- Requirements engineering includes the following activities:
  - Requirements capturing* — Collecting requirements from written specifications or interviews

- Requirements analysis* — The sophisticated mental process focused on conceptualizing, structuring, and formalizing the collected requirements
- Requirements specification* — Building complete, well-structured, and unambiguous artifacts
- Following are the artifacts of this phase:
  - An informal, but precise, complete, consistent, and unambiguous textual requirements specification
  - The UML analysis model of the system

### **Requirements Specification Document**

The purpose of the requirements specification document is to present the user's requirements in a structured, complete, and unambiguous form so that:

- The users can verify that the system to be implemented according to the specification is what they need and require.
- All participants in the development process (that is, modelers, developers, testers, documenters, and others) have a clear understanding of *what* has to be implemented.

In that sense, the requirements specification document is a kind of an agreement between the customers and the suppliers about the subject matter — the system under construction. If appropriate, that document can be the basis for the legal contract between the two parties, and can be referred to from it. For that reason, this document is particularly important.

It should be emphasized that the purpose of the document is to describe *what* must be done, not *how* exactly it must be implemented. In other words, the document should focus on the *requirements* and the *problem statement*, not on the *implementation* and *solution* to the problem. Of course, it is not always easy to keep a sharp boundary between the two, but the specification should refrain from influencing the implementation prematurely. If the requirements specification prejudices an implementation-level detail, then it makes an unnecessary restriction to the implementation. Often, during development, a design or implementation decision is changed to a better or more efficient one for any reason whatsoever — development cost and time, performance, simplicity, usability, and so on. The absence of such premature implementation decisions in the requirements specification leaves much more freedom for later development.

Of course, exceptions to this rule are possible. It is sometimes an explicit requirement of the users to pursue a specific implementation decision, technology, algorithm, platform, and so on. Of course, in such cases, these requirements should be introduced into the requirements specification document. However, the document should leave as much freedom as possible for the future development. For example, the specification can identify available alternatives and discuss their consequences, so that the decision can be deferred.

A collateral purpose of the document is to introduce the participants of the development process (modelers, developers, testers, documenters, and so on) to the context of the problem domain. This role is also

## Part IV: Method

---

important because those participants who have not taken part in requirements capturing often do not have any idea about the problem domain.

On the other hand, software engineers can build a useful and proper system that meets users' needs only if they truly understand the problem domain and the purpose of the system. No requirements specification can replace the understanding of the main purpose of the software system by its developers. However detailed and authoritative the requirements specification is, there will always be points during development when the developers must make some decisions on their own that may affect the usability and usefulness of the application. Only if they have some basic understanding of the problem domain can they make good decisions. However, in more complex projects, if the developers rely only on the requirements document for that purpose, chances are there will be big problems. As a rule, developers should have a much greater exposure to the problem domain and even to users than that, especially in more complex projects. The lack of understanding is one of the major sources of bad software.

One important consequence of this is the style of writing. The document should be written in such a way that a developer who reads it (and who is completely unfamiliar with the problem domain) can gain proper understanding of that domain. This is one potential pitfall of writing the requirements specification. Namely, when system analysts start the process of requirements capturing and interviews, they are usually completely unfamiliar with the problem domain and its business processes. Over time, during interviews, the analysts improve their knowledge about the domain and business processes, like collecting a puzzle in their minds.

During that process, the analysts often build some mental pictures about the domain or make decisions and parts of model that go astray in the end, or get improved or corrected by the further analysis. In the end, once the analysts have a good understanding about the problem domain and the user's requirements, they are ready to complete the UML analysis model and write the specification.

At that moment, many people make a mistake in the positioning their writing. They write from the viewpoint of their current (good) understanding of the problem domain, and not for those who have poor understanding (or no understanding at all). In other words, they forget that they might have felt lost at the beginning of the process — and the developers, who are supposed to build the system from those specifications, may feel exactly the same way at the very beginning. The writers tend to skip the introductory parts as if they write for the domain experts only, and go directly to the formal requirements specifications that then become unclear to the unprepared readers.

For that reason, it is important to follow the style of writing that allows those who are not familiar with the problem domain to quickly and efficiently gain the proper understanding. This does not mean that analysts should write all details from requirements engineering and minutes from interviews, or discuss all discarded design decisions they had in mind, or any misleading paths that were not taken. Instead, the point is to summarize conclusions, to describe briefly the problem domain or the user's motivation for building a new system to the extent it is necessary and sufficient for the target readership to understand them. Of course, if some discarded alternatives are important to emphasize because the developers might also think of them as logical and might get misled by them, it is useful to describe them, as well as the reasons why they have been discarded.

The very structure and contents of the requirements specification document heavily depends on the project, problem domain, company, and other factors. In fact, every company or development team might have its own preferred structure and document template. This is why this discussion merely provides some guidelines for structuring the document, and does not strictly define the structure. The contents mentioned here are useful in the requirements specification document in almost every project.

However, for every particular project, the document can have more or less contents and sections than described here. Some of the elements can be reduced to a few sentences, or enhanced to whole chapters or even separate documents, depending on the concrete needs and complexity of that element of the requirements.

The requirements specification document can have three main parts in general:

- Introductory part** — Describes the document, the problem domain, and the system (its scope and boundaries) in an informal way
- Central part** — Provides the precise structural and behavioral (functional) requirements
- Closing part** — Contains other non-technical requirements, risk assessment, and supporting material, if necessary

The suggested roles and contents of these parts will be described in the discussions that follow.

### Section Summary

- The purpose of the requirements specification document is to present the users' requirements in a structured, complete, and unambiguous form so that:
  - The users can verify that the system to be implemented according to the specification is what they need and require.
  - All participants in the development process (that is, modelers, developers, testers, documenters, and others) have a clear understanding of *what* has to be implemented.
- The purpose of the document is to describe *what* must be done, not *how* exactly it must be done. Exceptions are possible with compelling reasons.
- The document should be written in such a way that a developer who will read it and who might be completely unfamiliar with the problem domain can understand it.
- The requirements specification document can have three main parts:
  - Introductory** — Describes the document, the problem domain, and the system (its scope and boundaries) in an informal way
  - Central** — Provides the precise structural and behavioral (functional) requirements
  - Closing** — Contains other non-technical requirements, risk assessment, and supporting material, if necessary

### ***The Introductory Part***

The purpose of this part is to introduce the requirements specification document itself, as well as the system under consideration. Its task is to set up the context and introduce the readers to the problem

## Part IV: Method

---

domain, and to provide an overview of the system's scope, role, and boundaries. Its contents are mostly informal, and do not influence the system development directly. In other words, the development of the system is not driven by this part, but by the central part.

The suggested possible chapters in this part are examined in the following discussions. Every one of these chapters can be as short as a couple of sentences or paragraphs, or as long as an entire separate document, if necessary.

### Introduction

This part introduces the document itself. It should always begin with a stereotypical sentence such as, "This document represents the requirements specification for the system X." Immediately after such an introductory sentence, the system under consideration should be introduced by one or more informal, descriptive sentences, such as, "The system X is a ... " In that way, the reader will immediately get a rough idea of what the document and the system are all about.

In the rest of this part, the scope of the document should be described in more detail, if necessary. That is, the reader should understand what the document contains, and what it does not contain that might be expected.

The introduction should also list the prerequisites for reading and understanding the document. For example, some other documents should be read prior to reading this document because it relies on the contents of those other documents referenced from it.

In addition, the internal conventions, terminology, and special notation/symbols used throughout the document should be defined. One of the useful conventions that will be described in Chapter 18 can be introduced here: The key abstractions (concepts) from the problem domain are capitalized and italicized at the first mention.

Finally, following the conventional style of writing technical documents, the introduction can be concluded with a description of the structure and contents of the remainder of the document. Each chapter of the document can be announced in one or more sentences.

### Domain Description

This part sets up the context for the system. It describes the problem domain in an informal way. To do that, it describes business processes from the problem domain. A *business process* is a set of related activities that are performed among participants in a business to achieve a specific goal of a certain importance for the business. (The notion of a business process and its relationship with the system's functionality and their specification will be examined in Chapter 19.) Within the descriptions of the business processes, actors (roles of participants in the processes) and key abstractions from the problem domain are introduced on-the-fly.

Note that this part does not have to be formal, because it will not drive the development of the system directly. It simply sets up the context for the system. The formal description of the concepts from the problem domain and the system's functionalities will be given in the central part of the document.

Finally, this part can comment on the motivation of the stakeholders for building the new system under consideration. It is useful to emphasize the most critical parts of the business processes and the most

compelling reasons for introducing the new system, in order to draw the developer's attention to those parts and the system's features that will support them. If appropriate, the problems in exploitation of the legacy system should be also addressed.

If necessary, for more complex business systems, this part can be extracted into a separate document.

## System's Purpose

This part should describe the system's main purpose and goals. To do that, it is usually easiest to list and briefly describe the system's main, most important features and functionalities — those the users are most interested in and those that motivate its construction. Again, the description of the features and functionalities does not have to be precise and complete at this point. This will come in the central part of the specification. Instead, the purpose here is again to set up the scene for the precise requirements given later in the central part.

## System's Scope, Boundaries, and Environment

This part defines the system's scope and boundaries — that is, defines what the system will do and cover, and what it will not (but might be expected). It is particularly important to emphasize what the system will not cover, particularly the aspects that might be implicitly expected by the users, to prevent serious misunderstandings between the parties.

In addition, this part can define the system's environment — the way it interacts with users and other systems, the requirements related to the hardware and software platforms, and so on. Depending on the needs, the description of the interfaces with the outer or legacy systems should be described in more or less detail, or even extracted into separate documents.

### Section Summary

- ❑ The purpose of this part is to introduce the requirements specification document itself and the system under consideration.
- ❑ Suggested subparts include the following:
  - ❑ **Introduction to the document itself** — What the document is about, its scope and boundaries, the references, internal conventions, and structure
  - ❑ **Domain description** — Description of the problem domain and business processes, along with the motivation for the new system
  - ❑ **System's purpose** — Brief and informal description of the system's purpose and its main features and functionalities
  - ❑ **System's scope, boundaries, and environment** — What the system will do and cover, and what it will not, how it interacts with its environment, and so on

### The Central Part

This is the most important part of the requirements specification document because it directly drives the development of the system. Actually, this part is a combination of UML diagrams taken from the UML analysis model and textual descriptions of the elements from the model and of other technical requirements related to the elements from the model. These descriptions can also be extracted from the model (that is, from the documentation of model elements entered through the modeling tool).

This part addresses the following two basic aspects of the system that focus on the structural and behavioral part of the UML analysis model:

- Conceptual model of the problem domain (structure)** — This part describes the key abstractions from the problem domain and their properties and relationships, and, thus, defines the *vocabulary* of the domain.
- Functional requirements (behavior)** — This part specifies the functionalities of the system in terms of its *use cases*. Use cases are explained in detail in Chapter 19.

The central part, especially the specification of functionality (use cases), can be (and should be, where appropriate) accompanied by other technical requirements closely related to structure and functionality. They are central and important because they may have direct impact on the construction of the system. These include the requirements related to the following:

- Performance** — These are parameters such as throughput or response time and others.
- Capacity** — These are parameters such as memory or database capacity and volume of data.
- Reliability, availability, and fault tolerance** — Some requirements of this kind can be posed to the system in general, and some to particular use cases.
- User interface** — Some requirements of this kind can be posed to the system in general, and some to particular use cases. For example, a screenshot or sketch of an input mask, or a dialog, can be given for a specific use case, and so on.
- Data migration** — What kind and volume of legacy data must be migrated to the new system and how, what is the relationship between the legacy data structure and the new conceptual model, and so on?
- System migration** — How will the business environment be moved from the legacy to the new system and how does this relate to the specified use cases?
- Future extensions** — It is a good idea to indicate what future extensions of the system are envisioned because the designers and implementers should be aware of the extensions — not because these requirements must be implemented in the current version of the system, but because the designers must design the system so that the announced new features can later be incorporated seamlessly.

Other technical parameters, parameters, and requirements can also be specified here, if and where necessary and appropriate.

Because this is the most important part of the document, the discussion about how it should be written, along with the guidelines for how to get to the UML analysis model, are given in more detail in Chapter 18 (for conceptual modeling) and Chapter 19 (for functional modeling).

### Section Summary

- ❑ The central part is the most important part of the requirements specification document because it directly drives the construction of the software. It is a combination of diagrams from the UML analysis model and textual descriptions.
- ❑ Following are its two main subparts:
  - ❑ **Conceptual model of the problem domain (structure)** — Defines the vocabulary of the domain (key abstractions and their relationships) and other technical requirements closely related to them.
  - ❑ **Functional requirements (behavior)** — Specifies the use cases and other technical requirements closely related to them.

### The Closing Part

The closing part of the requirements specification document presents other technical and non-technical requirements related to the system under construction, which cannot be modeled in UML. It can contain other supplementary and supporting material needed for the specification.

It may also address the main factors of risk identified at the moment of project definition. For some projects, it is important to indicate those risks because the steering of the project and planning of development iteration will be driven by those risks, so that risks can be eliminated as early as possible.

### Section Summary

- ❑ The closing part of the requirements specification document presents other technical and non-technical requirements related to the system under construction.
- ❑ The specification can also address the main factors of risk identified at the moment of project definition.



# 18

## Conceptual Modeling

This chapter addresses conceptual modeling, one of the most critical activities in the process of building information systems. This chapter also gives some tips on how to identify concepts in the problem domain, and map them to elements of good models.

### The Process of Conceptual Modeling

The goal of conceptual modeling is to develop the *conceptual model* of the problem domain — the structural part of the UML analysis model, which defines key abstractions (that is, concepts), as well as their properties and relationships. It defines the *vocabulary* of the problem domain because it specifies the meaning of the concepts and terms from the problem domain.

The process of conceptual modeling is very challenging, difficult, and mentally intensive. Although it is so unpredictable, often very haphazard, and cannot be precisely prescribed, this section provides some guidelines.

As already suggested, at first, the system analysts should let the users speak and should listen to them carefully. It is a good idea to let the users state at least a brief (but complete) overview of their requirements before the parties go on with details, questions and answers, and brainstorming. Although the initial presentations of requirements are sometimes poorly structured, it happens that answers to the questions come implicitly in later stages of the presentations.

While carefully listening to the users' presentation of requirements and later answers to the questions and other discussions, the system analysts should try to identify the concepts and their relationships from the problem domain. Users typically present key abstractions from the problem domain in two ways:

- ❑ **Explicitly**, by trying to introduce and define terms and concepts from the domain
- ❑ And, more often, **implicitly**, by describing *use cases* (to be defined more precisely in Chapter 19 — that is, the desired functionalities of the system and scenarios of its behavior in use)

## Part IV: Method

---

However, direct extraction of concepts only from the users' presentation is usually insufficient. System analysts should generalize things and discover formal, abstract concepts that cover the users' requirements in a more general way. In other words, system analysts should constantly abstract things out from concrete examples and statements given in the users' talks.

During this process, the well-known design patterns can be of great help, especially those that can be applied during analysis for conceptual models (also known as *analysis patterns*) — mostly the structural patterns. Design patterns are elegant solutions to repeating problems of different kinds. Knowledge of design patterns is a powerful tool for every system analyst. System analysts recognize occurrences of the problems addressed by design patterns in the users' requirements, and try to apply design patterns, assessing the consequences that are also well-documented in design pattern catalogues.

Of course, a system analyst's experience is also of crucial significance. The solutions and models that a system analyst applied in the past are also a kind of a library of patterns individually established by the system analyst.

When the system analysts get an idea about the concepts and their relationships, they are ready to design a draft of (a part of) the UML conceptual model. They do this using a modeling tool, a piece of paper, or a whiteboard. More experienced analysts can develop pieces of the model in their mind and perform smaller manipulations of it (that is, improvements) even before they use a modeling tool. The model is extended and refined afterward, until it reaches a stable version that satisfies the users' requirements.

The developed (part of the) UML model is, at this stage, only a candidate model. Before being adopted, it should be carefully examined — the analysts should check whether the candidate model fully satisfies the users' requirements. To do that, the system analysts should take the use cases (the functionalities) described in the users' requirements and perform mental tests comprised of the scenarios of manipulating and querying the structure implied from the candidate model, just like the future system would do, to see whether the structure can satisfy the use cases and provide the information desired by the users.

However, testing against only the use cases that the users have explicitly mentioned in their requirements is usually not sufficient. System analysts should be creative enough to discover other use cases that can be expected in this or future versions of the systems. Users very often forget to mention some use cases that they expect, or state them quite late during the project. Creativity of the system analysts in this phase is, thus, crucial. The candidate conceptual model should also be robust enough to satisfy all such expected use cases. In addition, the users should be warned about all semantic implications of the envisioned model, especially those that are not obvious.

In addition, as already described, system analysts are not constrained to the concepts explicitly defined by the users. Instead, analysts should creatively discover generalized concepts that may simplify the implementation of the system, because they can cover several apparently different special cases in a common, unique way. Such invented concepts may often be biased by concrete ideas of implementation, or be rather formal and general, so that all their implications cannot be easily understood without prior knowledge of the semantics of the modeling language. An approach that is not careful enough in presenting the ideas may cause misunderstandings with the users who are not familiar with the implementation technology or modeling language. However, the inventions still must be discussed with and approved by the users.

In all such cases, the system analysts should describe the proposed concepts and the envisioned candidate model using domain-specific use cases and terminology, with concrete examples from the problem

domain that clearly (and unambiguously) demonstrate the manifestation and consequences of the proposed concepts and the envisioned conceptual model.

Users' requirements are very often not clear enough and sometimes ambiguous. To address an encountered ambiguity, the system analysts should first identify all possible and acceptable alternative interpretations of the users' statements. To select the right alternative, the system analysts should carefully select (discover or invent) the scenarios of behaviors of the system that qualify one and disqualify the other alternatives. The analysts then describe the scenarios to the users, ask them to identify the desired behavior of the system, and, finally, select the right alternative. If there is still more than one possible solution, the system analysts should select the simplest or the most flexible one. This is often a trade-off.

The process of conceptual modeling is also iterative and incremental. System analysts iteratively improve their understanding of the problem domain and build the conceptual model. The conceptual model is designed by extending, modifying, and refining the initial understanding and idea. The analysts invent a candidate model, then check it against the users' requirements (most notably, use cases), and improve the model, and so on, iteratively. Ultimately, when the system analysts achieve a good understanding of the problem domain and obtain a stable conceptual model, they are ready to describe it in the textual requirements specification document.

### Section Summary

- ❑ The goal of conceptual modeling is to develop the *conceptual model* of the problem domain — the structural part of the UML analysis model, which defines key abstractions (that is, concepts) from the problem domain and their properties and relationships.
- ❑ The process of conceptual modeling is difficult, mentally intensive, and unpredictable. It cannot be prescribed, but following are some guidelines:
  - ❑ Listen carefully to the users.
  - ❑ Identify abstractions, generalize things, and invent formal concepts.
  - ❑ Apply known analysis/design patterns when appropriate.
  - ❑ Check your abstractions and candidate model against the required use cases.
  - ❑ Invent other expected use cases to check your candidate model.
  - ❑ Propose generalized concepts and discuss them. Explain the concepts to the users by using domain-specific use cases.
  - ❑ If faced with ambiguities, discover alternatives. To select the right alternative, invent scenarios of behavior that qualify one and disqualify the others. Describe the scenarios to the users and select an alternative. If several alternatives are acceptable, select the simplest for implementation, or the most flexible one (usually a trade-off).

## Identifying Concepts and Relationships

The following discussions describe guidelines for how to identify different kinds of elements of a conceptual model. They will also give some illustrative examples of the process just described.

### ***Identifying and Specifying Classes and Attributes***

Key abstractions are most often modeled with classes. Therefore, to identify what will be a class in the conceptual model, the system analysts should first identify key abstractions from the problem domain. These are notions from the problem domain that are significant enough to be incorporated in the system under construction, regardless of the way it is implemented. To be modeled with a class, instances of such a concept should have the semantics of objects of classes — that is, they should have inherent identity, regardless of the values of their properties; they should be clearly distinguishable from other instances of the same or other classes; and they should have clearly defined lifetimes (although sometimes dependent on the lifetimes of other objects).

A good starting point for identifying key abstractions is the set of nouns that the users repeatedly use in their requirements, and those that the users consider significant for the problem domain. Of course, having noticed just a frequent noun in the users' requirements is still a weak rationale for introducing a class, because some nouns may not need to have manifestations in the conceptual model at all, or some may be modeled with other kinds of elements (such as data types or relationships). However, it is always a good idea to carefully address such emphasized nouns and notions in the users' requirements, and decide whether they should be modeled with classes, other elements, or not incorporated in the model at all.

As already discussed, the notions explicitly mentioned in the users' requirements are not the sole origin of classes in the conceptual model. System analysts should generalize things and invent other formal concepts that may be modeled with classes.

To distinguish the key abstractions to be modeled with classes from those to be modeled with data types, system analysts should be familiar with the semantic difference between the two, described in Chapter 8 in the section, "Discriminating Characteristics of Classes and Data Types." The clue is in the decision of where to set up the logical boundaries of the system. The abstractions whose instances should have their representatives with identities and be manipulated inside the system should be modeled with classes. The abstractions whose instances should remain outside the boundaries of the system and just be referenced from the objects in the system (over their attributes) should be modeled with data types.

It is generally simpler to model an abstraction with a data type and refer to an instance of that abstraction that resides outside the system over an attribute of another class. This is because the system is not responsible for the lifetime of those instances, which means less functionality and a simpler GUI. For that reason, it is sometimes useful to simplify the system by using a data type for an abstraction and attributes for the references to its instances (instead of a class and associations with it) in the initial version of the system. Modelers can then extend the boundaries of the system in later versions by promoting the data type into a class and the attribute into an association end. Actually, using attributes and data types instead of associations and classes is an efficient technique for reducing some unnecessary complexity of the system by narrowing its scope or responsibility to manage entities.

Once a key abstraction is identified and modeled, it can be defined and explained in the textual requirements specification in a combination of one or more of the following ways:

- By an explicit definition in one or more sentences such as, "X is . . ." In such a definition, many terms (usually nouns) can appear. Some of them can represent other key abstractions from the

problem domain (which carry semantics important for the system), while the others may be ordinary words taken from the spoken language (which do not have semantic significance for the system). The latter should be taken with their intuitive, everyday meaning, with the purpose of helping to understand the meaning of the concept being defined. It is extremely important for a reader of the specification to clearly distinguish between these two kinds of terms, because the former are important for the system, while the latter are not. This can be accomplished by simple typographic conventions. One intuitive and useful convention is to capitalize key abstractions in all places, and to italicize them at the first mention.

- ❑ In addition to an explicit definition (or when such is not possible or adequate), a key abstraction can be described in terms of its *roles*. Role is the function or position of an entity in a specific context. Such a context can be modeled in the UML analysis model with collaborations, while the role of the abstraction can be described in more detail using interaction diagrams, for example.
- ❑ Similarly, a key abstraction can be described by its *responsibilities*. Responsibility is a duty of an abstraction toward other abstractions.
- ❑ A key abstraction can be described by explaining its properties and/or relationships with other abstractions.

Finally, it is often useful to complete an explanation of a key abstraction by giving concrete examples from the problem domain. The examples can be instances of the abstraction (that is, concrete objects) easily recognizable by the users. That way, the users can more easily understand the abstraction and its purpose by mapping it to the concrete examples from the problem domain with which they are familiar. Examples help the users to understand how the key abstractions will classify concrete things from the problem domain.

Let's illustrate some of these guidelines with a simple example. Let's conceptualize the requirements for a personal organizer system. The purpose of the system is to enable its users to schedule their personal obligations and note them in the calendar. The discussions here will be much simplified and sometimes contrived in order to emphasize the described principles and guidelines. They will, however, reflect the real situations that occur during requirements capturing and analysis.

Without going into too much detail, let's assume that the users describe their requirements in terms of different kinds of things they want to write down in the organizer, such as working hours (for example, 8 hours a day, 5 days a week, except holidays), business meetings, business trips, vacations, reminders for taking medicine or making phone calls, and many others. Obviously, these examples, although representing important terms in the users' domain, are too specific to deserve separate classes in the model. Instead, the system analysts apply abstraction in order to discover their commonality while neglecting their specific characteristics. That way, the system analysts invent the proper abstractions that can cover all these items as just simple instances.

In other words, the system analysts introduce a classification of all items mentioned by the users according to their commonalities important for the system. Ultimately, the system analysts may come up with two abstractions that they can define explicitly in the requirements specification document as follows:

- ❑ *Obligation* is an element of the user's schedule that represents an activity that takes some time. Examples of Obligations include working hours, business meetings, trips, and so on.
- ❑ *Reminder* is an element of the user's schedule that represents a point in time at which the user should be reminded about some activity. Examples of Reminders include taking medicines, making phone calls, and so on.

## Part IV: Method

---

Notice that the discovered key abstractions represent sets (classifiers) of items from the problem domain according to their commonalities. Obligations take some time (that is, they are intervals on the timeline), while Reminders do not take time (that is, they are points on the timeline). Also notice that the abstractions are explained by explicit definitions. In those definitions, key abstractions are capitalized and italicized at the first mention in order to distinguish them from other words (most notably, nouns) that should not be taken formally, do not have semantic significance in the system, and do not represent key abstractions, but should be taken with their normal spoken-language meaning. Such words are, for example, “activity,” “time,” and so on. Finally, the definitions are followed by examples that show how the abstractions cover and classify the items given by the users. In that way, the users are aided in understanding the newly introduced abstractions, and can map them into concrete examples they are familiar with.

Identifying attributes of the discovered abstractions is usually straightforward. For this example, you can expect that the users have immediately required some properties, while the others could be proposed by the system analysts. For Obligation, these could be the following:

- Name** — A short title of the Obligation, for the purpose of quick referencing and presentation in the GUI
- Description** — A longer, arbitrary textual description of the Obligation
- Preparation** — Arbitrary text that reminds the user what must be done as a preparation for the Obligation
- Starting and ending time** — The date and time when the Obligation starts and ends
- Priority** — This can be “low,” “medium,” or “high”
- Color** — Used for the graphical representation of the Obligation in the calendar

Let’s assume that the users have another requirement. For obvious reasons, some Obligations can overlap in time, while others cannot. For example, scheduling a business meeting during (and as a part of) a business trip is obviously a desired feature, while the system should prevent its users from scheduling two business trips to different destinations at the same time.

To address this requirement, the analysts must apply the abstract, formal way of thinking and invent a solution. Of course, there are many ways to meet this requirement, whose flexibility and expressiveness usually increases the complexity of the model. Probably the simplest one is to introduce a Boolean attribute `canOverlap` in the class `Obligation` with the following meaning: If an Obligation has the value `false` of this attribute, it cannot overlap with other Obligations; otherwise, it can.

Obviously, such a solution cannot handle some more complicated situations. For example, it cannot handle a situation in which a business trip can overlap with a business meeting scheduled on that trip, but should not overlap with another, unrelated trip possibly also having this attribute set to `true`. To handle such cases, a more complex model should be invented. For example, there can be an association whose links explicitly indicate the Obligations that can overlap, or the notion of the location of an Obligation could be introduced, and so on.

The question is, which of these alternatives should be adopted? The first is simpler, while the others are more expressive. This is a typical trade-off. The answer can be obtained by asking the users whether they need a more complex solution at all. To get a reliable answer to this question, it is a good idea to invent scenarios that demonstrate the behavior of the system implied from a solution, and qualify one,

but disqualify the other solutions. You are encouraged to invent such sample scenarios for the current example. It might easily happen that the simpler solution is quite sufficient, and a more complex one might lead to a confusing behavior. In that case, the simpler solution should be taken. Let's assume that this is the case here.

This simple example also illustrates one interesting and important general classification of properties (attributes and association ends) in conceptual modeling. A property can be of one of two kinds:

- ❑ **Insignificant**, meaning that the system only stores and displays the value of the property (according to its type) — that is, provides the generic reading and writing functions, most often through the GUI, but does not base any computation, decision, or derivation of other information in the system on that value other than searches and reporting. In other words, such properties do not have semantic significance for the system; they are only significant for the users. The users simply enter, read, and search for the values (pose queries). For the given example, such attributes of Obligation are name, description, preparation, color, and priority.
- ❑ **Significant**, meaning that the value of the property has semantic significance for the system, because the system performs computation, makes conclusions, or derives other information from the value. For the given example, such attributes of Obligation are starting and ending time, and canOverlap.

It is useful to emphasize the category of a property to the developers because they can address these two kinds of properties with completely different attention. However, it is also useful to emphasize this distinction to the users, so that they can understand what to expect from the system. A property can be annotated as significant or insignificant in the model. These annotations do not affect the semantics of the model, but simply serve as hints to its readers. However, sophisticated tools can take these annotations as assertions, and check whether they are met in the rest of the model. For example, if a property annotated as insignificant is referred to from some computation in the executable model, the tool can issue a warning to the modeler.

As a result of this small initial analysis, the simple candidate model shown in Figure 18-1 is obtained. The operations `Obligation::canOverlap` return `true` if the two given Obligations (this and the one referenced by the parameter, or the two referenced by the parameters, respectively) can overlap (that is, if both have `canOverlap` set to `true`).

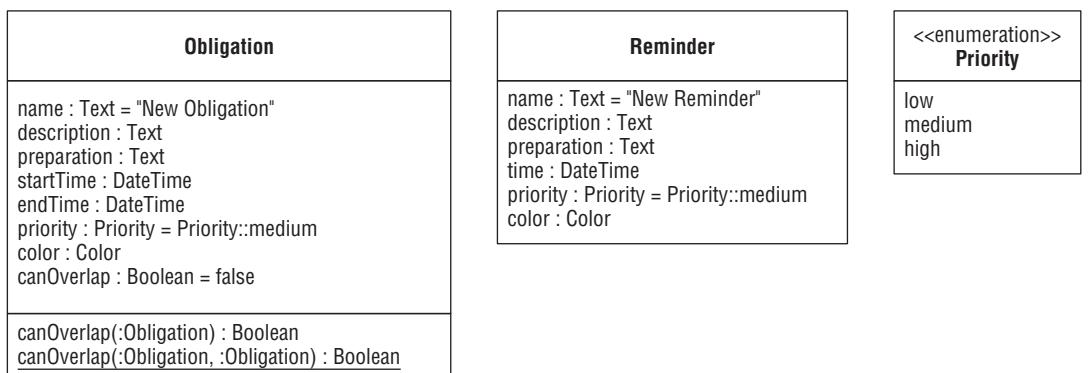


Figure 18-1: The initial candidate conceptual model for the personal organizer system

### Section Summary

- ❑ System analysts should identify key abstractions from the problem domain, but also generalize things and invent other formal concepts that will be modeled with classes.
- ❑ A key abstraction can be specified by an explicit definition, and/or by its role, responsibilities, attributes, and relationships with other abstractions.
- ❑ To distinguish between classes referenced by association ends and data types referenced by attributes, system analysts should set up the boundaries of the system and think about the identity of instances of the key abstractions.
- ❑ A property can be annotated as follows:
  - ❑ *Insignificant*, meaning that the system only stores and displays the value of the property (according to its type) — that is, provides the generic reading and writing functions, but does not base any computation, decision, or derivation of other information in the system on that value other than searches and reporting.
  - ❑ *Significant*, meaning that the value of the property has semantic significance for the system because the system performs computation, makes conclusions, or derives other information from the value.

## ***Identifying Generalization/Specialization Relationships***

Generalization/specialization relationships can be identified in two ways: by discovering specializations of an already discovered abstraction, or by discovering commonalities of already existing abstractions and factoring them out in a generalizing abstraction.

To identify specializations, pay attention to the users' qualifications, such as, "There are the following kinds of X ..." or "... is kind of X." However, such statements are often misleading and do not represent specializations. Following are possible alternatives:

- ❑ What are qualified by the users as "kinds" of a certain abstraction can be completely unrelated in the system. This is seldom the case because if the notions are related in the users opinion, they should be very likely related somehow in the system, too.
- ❑ What are qualified by the users as "kinds" can be simple objects of the same class, distinguished by attribute values, or can be different structures of objects connected with links.

A compelling reason for introducing specializing classifiers is redefined (polymorphic) behavior.

To identify generalizations, you should seek out commonalities of existing classifiers. Some motivations for generalizations of existing abstractions are as follows:

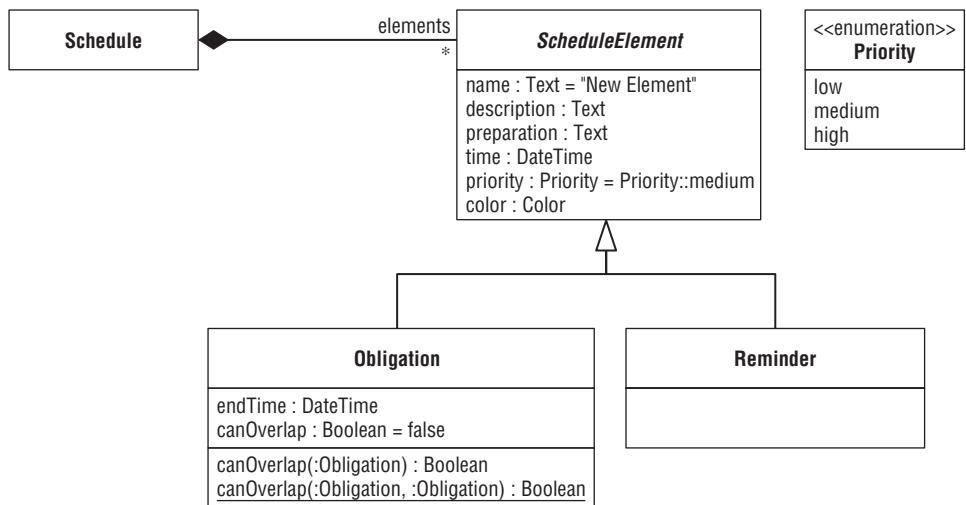
- ❑ **The abstractions have common properties.** This is a weak motivation, because completely unrelated classifiers often have some common generic attributes (such as name, title, description, and so on), as well as associations with other classes.

- The abstractions have common semantics (that is, meaning in the problem domain), and clients (that is, other classifiers) want to use their instances without seeing the difference between them. This is the most compelling reason for introducing a generalization.

In fact, premature generalizations are often as bad as no generalization. As a rule, a generalization should be introduced only after the analysts are sufficiently familiar with the domain.

Once a generalizing abstraction is introduced, the common features of the specializing classifiers should be carefully moved to the generalizing classifier to eliminate redundancy.

In the ongoing example of the personal organizer system, Obligations and Reminders have a common client — the *Schedule*. The Schedule is the common container of Obligations and Reminders. Apart from containing both Obligations and Reminders, the Schedule can have other interests in using them in a unique way. For example, it can ask them to draw themselves in the graphical representation of the Schedule, relying on a polymorphic operation for drawing. An Obligation can be rendered as a rectangular region in the calendar, spanning over a time interval, while a Reminder can be shown as a spot. For that reason, the generalization *Schedule Element* is introduced, as shown in Figure 18-2. Once it has been introduced, many of the common features of the two specializations can be moved to that class (actually, all of the attributes of Reminder).



**Figure 18-2: The conceptual model for the personal organizer system with the introduced abstract generalization Schedule Element**

Note that moving (outwardly accessible) features of specializing classifiers up in the generalization hierarchy cannot affect any of the existing clients of these classifiers because they will still have those features inherited. Therefore, such refactoring is harmless, and can be done at any time. On the other hand, the opposite transformation is very critical because instances of the generalizing classifier will not have some features that have been moved down the hierarchy.

During conceptual modeling, generalization/specialization hierarchies often reflect gradual classifications of instances in the problem domain. Really, a specific class represents a subset of the set determined

## Part IV: Method

---

by the general class. However, such sub-classifications can often be accomplished according to different orthogonal criteria relevant for the problem domain. For example, persons can be classified into male and female persons according to the gender, but also into adults and minors according to the legal status related with age.

Standard UML supports forming different, orthogonal groups of specializations through the notion of a *generalization set*. Each generalization set defines a particular set of generalization/specialization relationships having a common general classifier that describes one way in which the general classifier may be subclassified.

For example, as shown in Figure 18-3a, vehicles can be divided into subclasses according to manufacturers and according to kinds. Each of these generalization sets represents one such orthogonal categorization. The first generalization set, named `manufacturer`, includes the relationships between the common generalizing class `Vehicle` and the subclasses representing manufacturers (`Mercedes`, `Peugeot`, `Fiat`, and so on). The second generalization set, named `kind`, includes the relationships `Vehicle-Car` and `Vehicle-Truck`. There may be another different partitioning of the same class `Vehicle` into different subclasses defined by another generalization set.

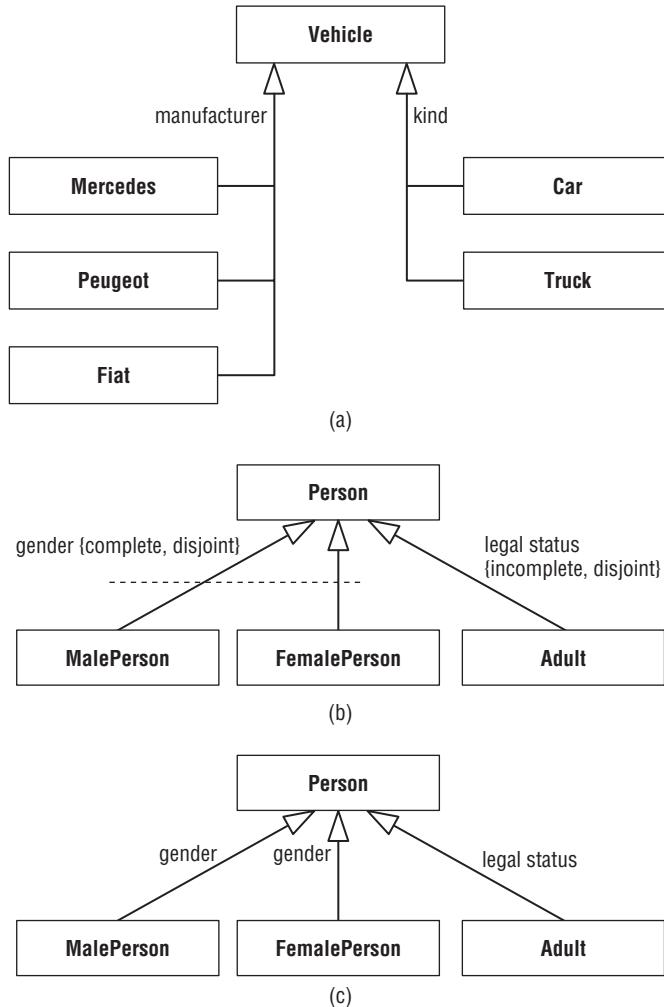
A generalization set is a packageable element and may have a name. All generalization/specialization relationships in a generalization set must have the common general classifier. A generalization set can be annotated as follows:

- ❑ **Complete or incomplete** — When a generalization set is *complete* (or *covering*), the specific classifiers of the generalization set cover the entire set of instances of the common general classifier, meaning that the union of the specific classifiers (as sets of instances) is the general classifier. In other words, every instance of the general classifier is also an instance of at least one of the specific classifiers. When it is *incomplete* (or *non-covering*), there are one or more instances of the general classifier that are not instances of at least one of its specific classifiers in the generalization set. For the example in Figure 15-3b, the generalization set `gender` is covering because every instance of `Person` would be an instance of `Male Person` or `Female Person`. In contrast, the generalization set `legal status` with only one specific classifier `Adult` is not covering because there are instances of `Person` that are not adults, but rather something else (minors).
- ❑ **Disjoint or overlapping** — When a generalization set is *disjoint*, the specific classifiers of the generalization set have no common instances (that is, their intersection is empty). When it is *overlapping*, the specific classifiers of the particular generalization set have one or common instances (that is, their intersection is not empty).

A complete and disjoint generalization set is also called a *partitioning* of the common general classifier.

A generalization set can appear in diagrams in several different ways, as shown in Figure 18-3. First, as shown in Figure 18-3a, the lines for generalization/specialization relationships from the same generalization set could share a common arrowhead, different from other generalization sets. This sharing indicates the generalization set. The name of the generalization set can be shown near the common line segment, and is optional.

Second, as shown in Figure 18-3b, when the relationships from the same generalization set do not have a shared arrowhead, a dashed line crossing the relationships indicates that they belong to the same generalization set. The name of the generalization set can be placed close to the dashed line, and is also optional.



**Figure 18-3: Examples of generalization sets and their alternative notations. (a) Vehicles classified according to manufacturers and kinds, shown using the shared arrowhead notation. (b) Persons classified according to gender and legal status, shown using the separate target notation. (c) Alternative notation for (b) with labeled generalization lines.**

Finally, as shown in Figure 18-3c, every generalization/specialization line can be labeled with the name of the generalization set it belongs to. The same name on different lines indicates that the generalization/specializations belong to the same generalization set. The annotation of a generalization set is shown within curly braces near the name of the set, as shown in Figure 18-3b, and can be one of the following: {complete, disjoint}, {incomplete, disjoint} (default), {complete, overlapping}, or {incomplete, overlapping}.

Note that existence of several generalization sets as orthogonal dimensions of specialization for the same general classifier implies the need for the support for classification of instances to several classifiers at

## Part IV: Method

---

the same time (or at least for multiple inheritance). Namely, for the example in Figure 18-3b, if a specific Male Person is an adult, he is then an instance of both `MalePerson` and `Adult` at the same time. This means that the language should support associating multiple classifiers with the same instance (which standard UML does), or the modeler should introduce another class `MaleAdult` derived from both `MalePerson` and `Adult`.

OOIS UML does not support generalization sets. More precisely, generalization sets do not have any formal semantics in OOIS UML, and are simply ignored in the design models. Consequently, a generalization set is a highly abstract concept used in early phases of conceptual modeling. In later phases of analysis and design, the parts of models containing generalization sets should be transformed to some other forms having executable semantics according to the purpose of the analysis model.

### Section Summary

- To identify specializations, attention should be paid to the user's qualifications, such as "There are the following kinds of X" or "... is a kind of X." However, such expressions are often misleading and do not indicate specializations, but simple objects of the same class or structures of objects.
- To identify generalizations, common features and clients of existing classifiers should be discovered.
- A *generalization set* defines a particular set of generalization/specialization relationships having a common general classifier that describes one way in which the general classifier may be subclassified.

## Identifying Associations

Following are some hints on how to identify associations:

- Discover structural connections between objects. Classify such connections into associations according to their meaning, purpose, and way of manipulation.
- Discover complex structures of objects. They indicate associations.
- Apply design patterns. They often include associations.
- Discover more implicit relationships that can be modeled with associations.

Let's illustrate the process of identifying associations with a new requirement for the ongoing example of the personal organizer system. The users now want a reminder for making a phone call to book the tickets for a business trip to be canceled if the trip is canceled.

This simple requirement illustrates one common occurrence. Although the users have adopted the introduced abstractions (such as `Obligation` and `Reminder` in this example), they get back to particular examples (usually instances of these abstractions), such as, "a reminder for making a phone call to book the tickets," "business trip," and "cancellation," as soon as they come up with a new requirement that they are not sure how to map to the existing abstractions. It is a challenge for system analysts to handle such cases.

To cover this particular requirement, the system analysts propose the following concept. A Schedule Element X can *depend on* another Schedule Element Y, meaning that when Y is removed from the Schedule, X will be also automatically removed.

The introduced relationship *depends on* should be obviously modeled with an association between the classes X and Y, because the objects of X and Y should be structurally connected by a link that stores the information of “being dependent on.” If there is a link between an X and a Y, then the X is dependent on the Y; otherwise, it is not. Obviously, the X and the Y are elements of the Schedule, and are, thus, (direct or indirect) instances of *ScheduleElement*, but are they *Obligations* or *Reminders*, or can they be both? How can the classes X and Y be determined? In other words, how do you determine where to attach the ends of the association *dependsOn*?

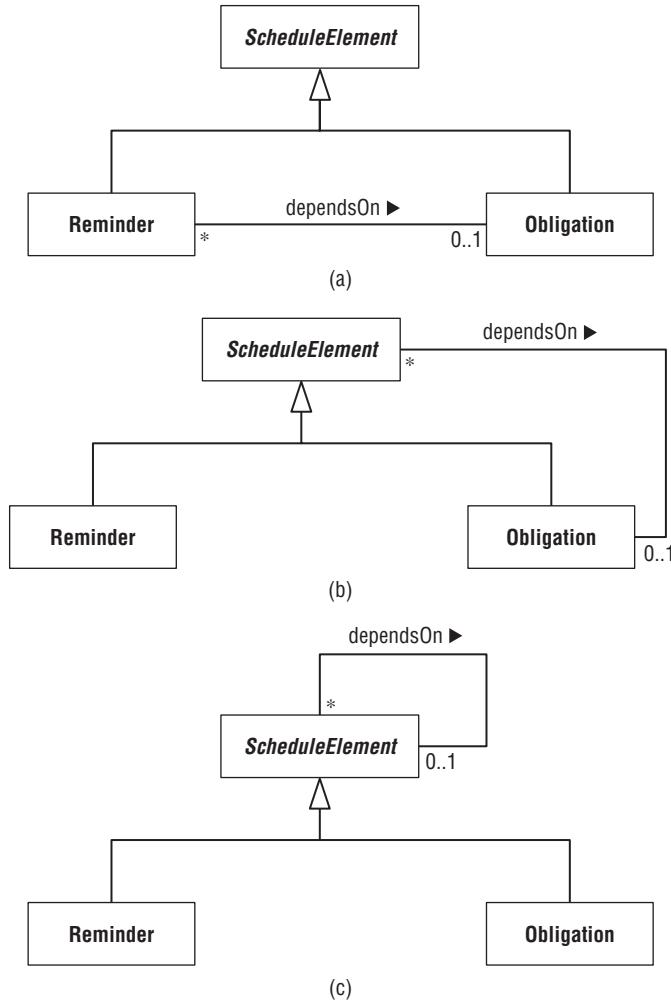
As already explained, there are usually no ideal solutions to such problems. Instead, system analysts should identify possible alternatives and invent the proper use cases that qualify one and disqualify the other alternatives. The use cases should then be discussed with the users, in terms of concrete examples from the problem domain (usually, instances of the participating abstractions), in order to select the right alternative.

In this example, the analysts may start from the most specific alternative, the one directly implied from the requirement given by the users. Obviously, the requirement to “remove a reminder for making a booking phone call,” which is a *Reminder*, “when a business trip is removed,” which is an *Obligation*, indicates the necessity that the association at least allow linking *Reminders* (as dependents) and *Obligations* (on which they depend), as shown in Figure 18-4a. It also immediately disqualifies the alternatives where the association is of *Reminder* with itself or *Obligation* with itself.

The question now is whether this alternative is sufficient (that is, whether it covers all necessary cases), or a more general model is needed. Actually, it is necessary to find out whether an *Obligation* can depend on another *Obligation*. To get the answer to that question, instead of drawing UML diagrams and asking the question directly, the system analysts can invent a concrete example, such as, “Is it possible that a business meeting (which is obviously an *Obligation*) has to be canceled (removed from the Schedule) when a business trip on which it is planned is canceled?” The system analysts can easily conclude, even without asking the users, that the answer to this question is positive. Consequently, the model shown in Figure 18-4a is insufficient, and a more general one is needed.

The next step is to move the end of the association *dependsOn* from *Reminder* to *Schedule Element*, as shown in Figure 18-4b. From the previous analysis, it is necessary that both *Reminders* and *Obligations* can depend on *Obligations*, but is this model sufficient? The analysts can find that out if they know whether an even more general model shown in Figure 18-4c is needed. Namely, if there were a need that an *Obligation* depend on a *Reminder*, then the model in Figure 18-4b would not be adequate, and the model shown in Figure 18-4c would be necessary.

To find that out, the analysts can ask the users whether they would need a business trip or meeting to be canceled when a reminder for a phone call (or similar) is canceled. Let’s assume that the users answer negatively to this question, as they can be confused by it. In addition, the system analysts can conclude that such a model could lead to mistakes during the exploitation of the system because it does not prevent incorrect swapping of roles of the dependent element and its master. In other words, a user can incorrectly set up that a business trip depends on a reminder to make a booking phone call, instead of the reverse. Note that the model in Figure 18-4b would prevent such errors. Actually, superfluous generalization removes information of concrete types and relaxes type constraints, so it can lead to mistakes.



**Figure 18-4: Alternatives for setting up the association**  
**dependsOn**

As a conclusion to this brief analysis, the correct placement of the association `dependsOn` for the users' requirements is shown in Figure 18-4b.

Identifying multiplicities of association ends is usually straightforward. Analysts should pose the questions targeted at discovering the lower and upper multiplicity bounds. For the given example, the questions could be the following:

- ❑ Can a Schedule Element be dependent on (removed along with) more than one Obligation at a time? A probable answer is, "No."
- ❑ Can it be independent? The obvious answer is, "Yes."

- ❑ Can an Obligation have an arbitrary number of dependent Elements, including none? The obvious answer is, “Yes.”

As the result of this line of questioning, we get the model in Figure 18-4b completed with multiplicities.

It should also be noted that the association dependsOn is not a composition (with the whole part at the Obligation end). Note that a Schedule Element is already part of another composite — the Schedule, and, by definition, one object cannot be part of more than one composite at the same time.

An option would be to use propagated deletion from Obligation to the dependent Schedule Element, but note that the invented concept of dependency means propagated removal from the Schedule, not necessarily propagated deletion from the system. Removal from the Schedule might be implemented, for example, by linking the removed Schedule Elements to a special, reserved object that serves as a “recycle bin,” so that they can be restored or used in other Schedules, and so on.

### Section Summary

- ❑ Following are hints on how to identify associations:
  - ❑ Discover structural connections between objects. Classify such connections into associations according to their meaning, purpose, and way of manipulation.
  - ❑ Discover complex structures of objects. They indicate associations.
  - ❑ Design patterns often include associations.
  - ❑ Discover more implicit relationships that can be modeled with associations.

## Modeling Type-Instance Relationships

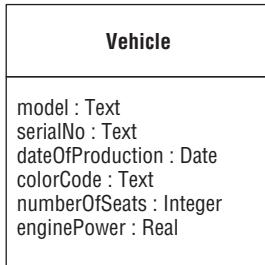
Very often, the problem domain includes concepts that represent *types* or *classifiers* of other concepts from the domain. Let's consider an example of an information system for a car dealership that sells vehicles. Each vehicle has its own identity and properties (such as serial number, year of production, and so on), and it can be sold to a customer. Additionally, each vehicle has its model (for example, Mercedes S300, Peugeot 307, Fiat Punto, and so on).

In this system, Vehicle is obviously a key abstraction because it has instances with clear identity and properties. The company wants to trace each instance of Vehicle (that is, to make a booking, and then sell it to a particular customer). But how can we model the type of a Vehicle?

The remainder of this section discusses several available alternatives to handle such cases. The alternatives are presented gradually, starting from the simplest to the most complex and flexible. Each of the alternatives has its positive and negative consequences. In a concrete circumstance, the modeler should opt for the simplest alternative that is acceptable, despite its possible drawbacks.

### Modeling with an Attribute

The simplest solution is to use an attribute that specifies the “type” of an object, as shown in Figure 18-5. The attribute can be of a simple textual type, or, more adequately, of an enumeration or another abstract data type.



**Figure 18-5: Modeling type of an instance with an attribute**

An advantage of this solution is that it results in a very simple implementation because it does not have to deal with complex object structures. Managing attribute values is much simpler than managing more complex object structures. This also results in simple use cases and a simple GUI.

A disadvantage of this solution is that the objects of the same “type” are loosely coupled or completely decoupled. In fact, the “instances” of the same “type” are recognized by having the same values of certain attributes. This can be referred to as an *indirect* or *implicit* classification, or *value-based* classification. The “types” (models of vehicles, in this case) are not separate abstractions, and cannot have their own instances with identities, attributes, associations, and behavior. For example, how to easily set the number of seats or the engine power rate, modeled as attributes of `Vehicle`, of all vehicles of the same type to the same values, and to keep them synchronized later (if the user modifies one, how to update all the others)? To accomplish this, a more complex “find and replace” operation is necessary. If such features are needed and frequently initiated, this approach is not the right way to go.

#### Section Summary

- ❑ The simplest solution is to use an attribute that specifies the “type” of an object. The attribute can be of a simple textual type, of an enumeration type, or of another abstract data type.
- ❑ An advantage of this is that it can result in a simple implementation and GUI.
- ❑ A disadvantage is a loose coupling of objects of the same “type” (that is, implicit classification over attribute values).

## Applying the Prototype Design Pattern

The next approach is a direct application of the *Prototype* design pattern [Gamma, 1995]. The idea is to have a prototype object or an entire object structure for each “type” of objects (vehicle model, in this case), with preconfigured attribute values, which can be cloned to create other instances or structures identical to the prototype.

The idea of this approach is illustrated in Figure 18-6. There is still only one class that models Vehicles, the same one as in the previous approach. It is just extended with the `clone` operation. The clue is in the object space, where two distinct categories of objects of the same class are recognizable.

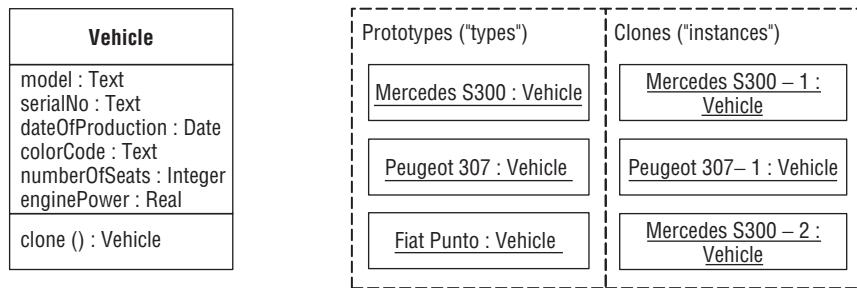


Figure 18-6: Using prototypes to conceptualize types of instances

The first category comprises the prototype objects, one for each “type” (vehicle model, in this case). The prototype objects (or entire structures of objects) are preconfigured with the values of their attributes according to their “types.” For example, they can have preconfigured engine power and number of seats. The “instances” are created as clones of prototypes by the operation `clone`. Being clones of their prototypes, they get the same values of the attributes that reflect their belonging to the corresponding “type.” However, the clones of a prototype live their lives independently from their creation on, including having independent values of attributes.

A well-designed GUI can provide a complete illusion that users can separately work with “types” and their “instances” as different abstractions. For example, a separate part of the application (possibly accessible to privileged users only) can allow manipulation with the prototype objects — their creation, deletion, browsing, and modification. Another part (accessible to a wider group of users) can allow only selecting a prototype object and creating its clones, as well as manipulation with clone objects. These objects can even be labeled and depicted differently by using GUI configuration and context features.

Let’s summarize the advantages of this approach:

- ❑ It is still a very simple approach, almost as simple as the previous one, with no additional classes.
- ❑ Users can define prototypes and set their attribute values arbitrarily, at run-time.
- ❑ Complex structures of objects can be created as prototypes (if necessary).
- ❑ If needed, derived classes can be defined for the base class. Clients of prototypes are independent of their actual type.
- ❑ The GUI can successfully fake the type/instance dichotomy as described earlier.

## Part IV: Method

A disadvantage of this solution is that since its creation, a clone lives independently of its prototype. For example, if the engine power rate of a car model is changed in the prototype, the existing clones are not affected and cannot be kept synchronized without specially designed synchronization behavior. Actually, this is only conditionally a disadvantage, because such behavior can even be a desired feature. For example, if the producer modifies a vehicle model (for example, modifies the engine power rate of the model), the system may still have to keep the original value of that attribute for all existing vehicles that have already been produced in the old configuration. Again, this is a matter of the specific needs from the problem domain. In addition, unless subclasses are introduced, the “types” (which are actually simple prototype objects) cannot have different attributes and specific behavior.

### Section Summary

- ❑ When applying the *prototype design pattern*, the idea is to have a prototype object or an entire object structure for each “type” of object, with pre-configured attribute values, which can be cloned to create other instances or structures identical to it.
- ❑ This solution is simple and allows creating “types” at run-time, introducing of subclasses with extended or redefined structure and behavior, and making illusions in the GUI of separate manipulation with “types” and “instances.”
- ❑ A limitation is that, since their creation, clones live independently from their original prototypes.

### Modeling with Specialization Hierarchy

In this approach, “types” are directly modeled with subclasses in a class hierarchy, as shown in Figure 18-7. This is a straightforward (and often naive) use of the generalization/specialization relationship to model the “kind of” relationship between key abstractions in the problem domain, often shown in beginner’s textbooks on OO programming as a starting example of inheritance.

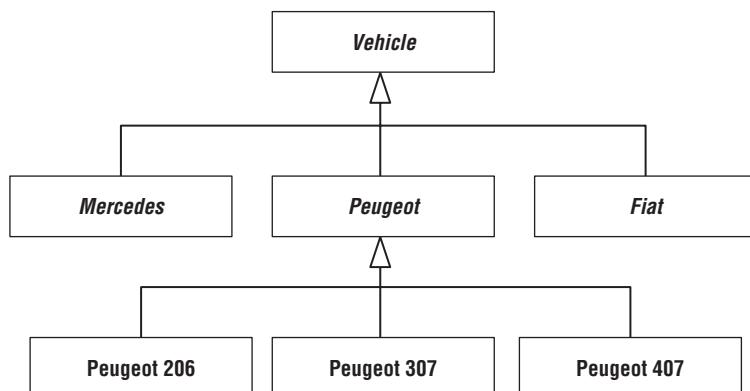


Figure 18-7: Using class hierarchies to conceptualize different types of instances

Unlike the previous approaches, this is one with *explicit* or *direct classification* by the “vertical” type/instance dichotomy between the model and the object space, where “types” reside in the conceptual model, and “instances” reside in the object space.

Following are the advantages of this solution:

- ❑ The hierarchy often directly fits in the domain’s classification and is easy to recognize.
- ❑ By virtue of being classes, the “types” can have their own properties, relationships, and specific behavior.

Following are the disadvantages of this solution:

- ❑ Even for modestly complex domains, the hierarchies quickly become clumsy and difficult to manage.
- ❑ New types cannot be added by users at run-time, unless the framework supports modeling at run-time, which is rare and not easy to implement.

### Section Summary

- ❑ In this approach, “types” are directly modeled with subclasses in a class hierarchy.
- ❑ The hierarchy often directly fits in the domain’s classification. The “types” can have their own properties, relationships, and specific behavior.
- ❑ The disadvantages are that the hierarchies quickly become clumsy and difficult to manage, and that new types cannot be added by users at run-time.

### Modeling with Type/Instance Association

This approach assumes modeling “types” and “instances” explicitly with two classes, related with an `instanceOf` association, as shown in Figure 18-8. Now, objects of `VehicleType` represent the “types” of vehicles (models), whereas objects of `VehicleInstance` represent concrete pieces that are sold to the customers. These two classes are associated so that every `Vehicle Instance` has a link to its type (model). Note how the attributes (and other features) are now distributed among these two classes, according to their logical belonging. The attributes whose values are shared among different instances of the same type belong to the `VehicleType` class, while those that are individual characteristics of every piece are in `VehicleInstance`.



Figure 18-8: Modeling “types” and “instances” explicitly with two associated classes

## Part IV: Method

---

Of course, when the data about one particular piece is displayed in the application, some of the data (such as the number of seats and engine power rate) are taken from the Vehicle Type associated with that object of Vehicle Instance. Such an associated “type” object always exists, because it is guaranteed by the multiplicity 1 at the type end of the association. By configuring the command CmdCreateObjectAndLinkToObject properly in the GUI, the user can easily create an “instance” of a certain “type” of vehicle.

This approach uses *explicit* or *direct classification* by the “horizontal” type/instance dichotomy modeled with an association between two classes, where both “types” and “instances” reside in the object space.

Following are the advantages of this solution:

- ❑ Users can define “types” at run-time and set their attribute values arbitrarily.
- ❑ “Instances” are kept synchronized with their “types.” For example, the engine power rate of a vehicle is an attribute of its type, so it is modified for all instances of a type at once. Of course, this is an advantage only if such behavior is really needed, as discussed earlier.

Following are the disadvantages of this solution:

- ❑ It is somewhat more difficult to implement — more use cases are needed to manage the structure.
- ❑ “Instances” of different “types” do not have specific features.

Note that subclasses of `VehicleType` can be introduced to extend or redefine features for “types,” but they do not directly extend or redefine features of their “instances.” Alternatively, the `VehicleInstance` class can provide services that delegate their responsibility to polymorphic operations of `VehicleType`.

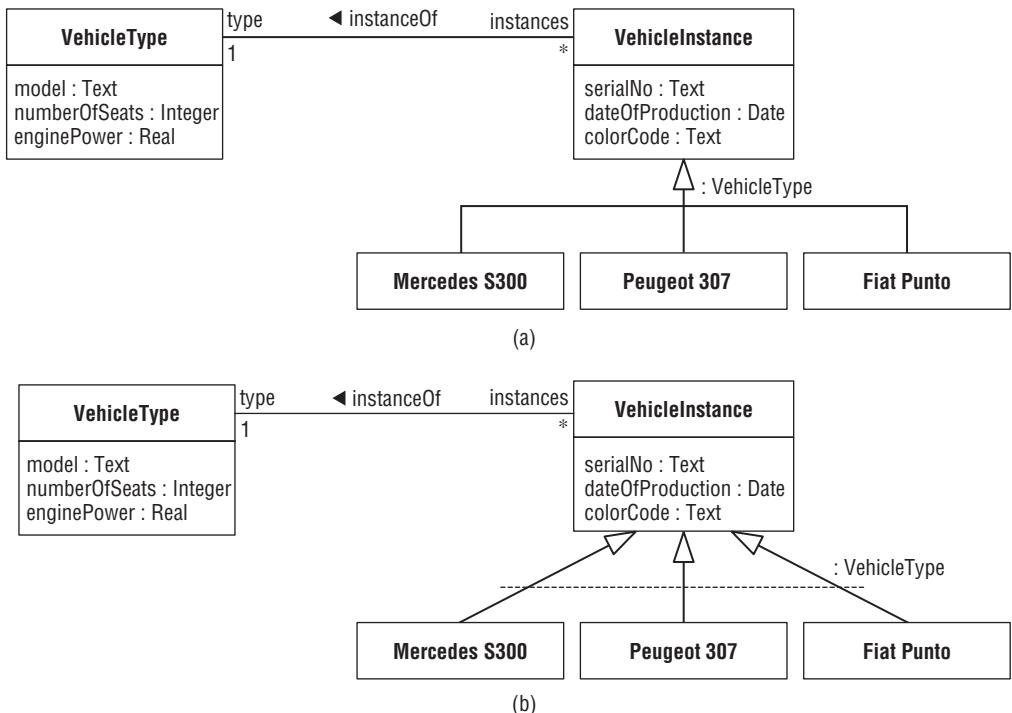
### Section Summary

- ❑ This approach assumes modeling “types” and “instances” explicitly with two classes, related with an `instanceOf` association.
- ❑ The advantages of this approach include the fact that the user can define “types” at run-time and set their attribute values arbitrarily, and that the “instances” are kept synchronized with their “types.”
- ❑ The disadvantages are that it is somewhat more difficult to implement, and “instances” of different “types” do not have specific features.

## Power Types

It is interesting to note that the last two approaches are opposites, to some extent, because their advantages and disadvantages are reversed. In the approach with specialization class hierarchies, subtypes can have their specific features, whereas in the approach with the associated type-instance classes, subtypes are represented with objects with their property values and behavior. But they are also orthogonal, and can be combined. Their combination is supported by the notion of *power type* in UML.

Figure 18-9 illustrates the idea. The class `VehicleType` is a power type for the generalization set gathering the generalization/specialization relationships having `VehicleInstance` as the common general class. This means that for every object of `VehicleType`, there is exactly one subclass in the generalization set, and vice versa. In fact, every object of `VehicleType` is a specific subclass of `VehicleInstance` as well.



**Figure 18-9: Modeling “types” and “instances” with a power type. Alternative notations (a) with a shared generalization arrowhead and (b) with a dashed line.**

In general, a power type is a classifier whose instances are at the same time specializations of another classifier. In a UML model, a power type is a classifier related to a generalization set. Instances of that classifier are at the same time the special classifiers in the related generalization set. Actually, a power type crosses the boundary between the type (model) and the instance (object) spaces — its instances reside in both.

In diagrams, the power type classifier for a generalization set is indicated by placing its name (preceded by a colon) next to the generalization set. Figure 18-9 shows this for both notational variants for generalization sets — the shared arrowhead and the dashed line.

At first glance, it could seem that having both the `VehicleType` class and the subclasses of `VehicleInstance` in Figure 18-9 is redundant and can cause potential model maintenance issues. For example, if a new car type must be introduced, an object of `VehicleType` must be created in the object space, but the corresponding subclass of `VehicleInstance` must also be added to the model to maintain its integrity. This similarly holds true if one must be deleted — both the subclass of `VehicleInstance` and the instance of `VehicleType` would have to be deleted from the two spaces. However, this is only an

## Part IV: Method

---

apparent redundancy and integrity issue because the subclasses of `VehicleInstance` and the objects of `VehicleType` are logically the same entities.

In short, the concept of power type is extremely expressive and flexible, and combines the advantages of the two previous approaches to modeling type-instance relationships. Types can be defined dynamically at run-time. They can have their own property values and behavior, while instances of specific subtypes can have extended or redefined features.

However, all these considerations are purely conceptual, and do not have to have practical implications. Implementing the concept of power types may be a big issue. One approach is that the modeling framework and run-time environment support dynamic creation of objects (as instances of power type classes) and the subclasses at the same (run)time (that is, to support objects being classes at the same time). Unfortunately, this may be extremely difficult and inefficient to implement, and very few (if any) UML modeling and execution frameworks support it completely. If they do not, redundant structures must be introduced, and the integrity maintenance issue mentioned previously should be solved.

For all these reasons — similar to generalization sets, which are closely related — OOIS UML does not support power types. More precisely, power types do not have any special formal semantics in OOIS UML, and are simply ignored in design models. Consequently, power type is a highly abstract concept used in early phases of conceptual modeling to describe real-world situations. In the later stages of analysis and design, the parts of models containing power types can be transformed to some other forms having executable semantics according to the intention of the analysis model, or the synchronization between the instances of the power type class and the subclasses in the model should be kept by other explicit means.

### Section Summary

- ❑ A *power type* is a classifier whose instances are at the same time subclasses of another classifier.
- ❑ A power type is a purely conceptual notion used in early stages of analysis.

# 19

## Modeling Functional Requirements

Modeling functional requirements is another important aspect of requirements engineering. This chapter introduces the notion of a use case as the basic building block for modeling functional requirements. It also presents tips on how to deal with the sheer number of use cases in complex information systems.

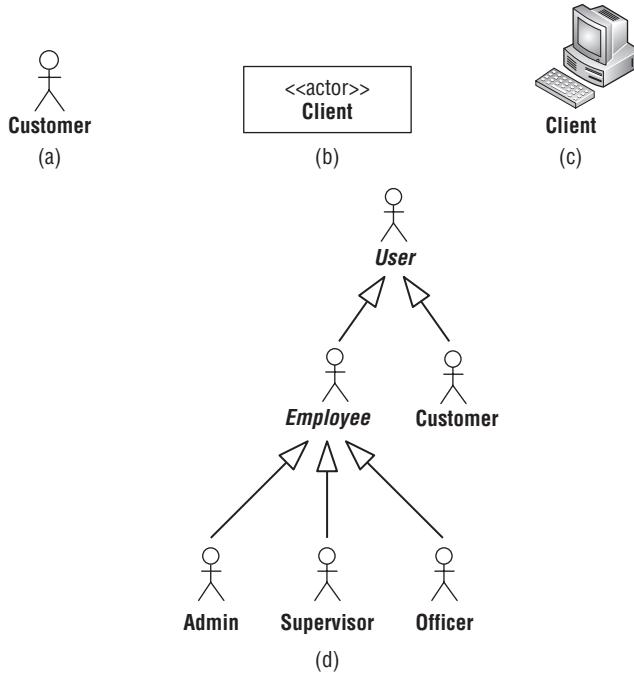
### Actors and Use Cases

Modeling with *use cases* is a technique for specifying functional requirements of a system. A use case describes a unit of the required functionality of the system. Use cases describe what the system is supposed to do. Each use case describes one or more scenarios that convey how the system under consideration should interact with users or other systems (called *actors*), to achieve a specific business goal or function. The system under consideration to which the use cases apply is referred to as the *subject*.

#### Actors

An *actor* models a role played by external entities (such as human users or other systems) that interact with the subject by interchanging signals and data. In any case, the entity represented by an actor is *external* to the subject. Although actors exist in the model, they represent entities that are not part of the modeled system. They are used in the analysis model to define the system's environment. For example, the actor *Customer* in Figure 19-1a models the set of all users of the system under consideration who play the role of customer when they interact with the system.

An actor represents a role, not necessarily a specific physical entity (for example, a concrete person or a certain system). Namely, a single physical entity (for example, the same particular user) may play several roles represented by different actors. The term "actor" thus refers to one of those roles, and not to that particular entity. For example, John Smith could be playing the role of a *Customer*



**Figure 19-1: Actors.** (a) The default “stick man” notation. (b) Alternative notation with the class symbol. (c) Alternative notation with an icon. (d) An example of a hierarchy of actors.

when using an Automatic Teller Machine (ATM) to withdraw cash, or playing the role of a *Bank Teller* when using the system to restock the cash drawer. The implied actors are Customer and Bank Teller, not John Smith. This similarly holds true for actors that represent non-human entities (that is, other systems). The actor **Client** in Figure 19-1c can represent the entire set of computers of a certain type that interact with the subject in a given way.

Obviously, many physical entities can play the role represented by a single actor. Consequently, an actor designates a set of entities that play a certain role. This is why actors are classifiers in UML. Actors live in the model, at design time. Instances of actors live in the subject’s environment, and interact with the subject at run-time. Of course, an actor can also designate a single-element set with one particular person or system.

In UML models, an actor must have a name. Actor names should follow the naming guidelines for classes.

The symbol for an actor is the “stick man” icon. The actor’s name is placed near the icon, usually below or above it, as shown in Figure 19-1a. If the actor is abstract, its name is shown in italics. Alternatively, an actor may also be represented as any other classifier with a rectangular symbol having the keyword **<<actor>>** (see Figure 19-1b). Finally, an actor can be depicted with any other icon that conveys its specific kind. For example, a special icon may be used for non-human actors, as shown in Figure 19-1c.

Actors are classifiers and, thus, can be related by generalization/specialization relationships (see Figure 19-1d). As always, their meaning is that an instance of the specializing actor can substitute an instance of the generalizing actor — whatever holds for the generalizing actor, also holds for the specializing actor. Whenever an instance of the generalizing actor is expected, an instance of the specializing actor can occur. That is, all interactions with the subject defined for the generalizing actor also hold for the specializing actor. An instance of the specializing actor can be involved in all of them. Hierarchies of actors represent subclassifications of roles, as shown in Figure 19-1d. Generalization of roles may reduce redundancies because all use cases associated with a generalizing role are implicitly associated with a specializing role.

### Section Summary

- ❑ An *actor* models a role played by an entity that interacts with the subject, but is *external* to the subject.
- ❑ An actor is a classifier and represents a particular facet of a set of concrete entities (instances) when interacting with the subject at run-time.

## Use Cases

A *use case* is the specification of a set of actions performed by the subject, possibly including variants, which yields an observable result of value to one or more actors or other stakeholders of the subject. A use case is actually a description of one or more scenarios of interaction of an actor (or actors) and the subject, that conveys how the subject is used in a particular situation and how it should behave in order to achieve a certain business goal or function. It may be regarded as an outwardly visible and testable behavior of the subject with a clear goal, or a description of a collaboration between the subject and one or more actors.

Let's consider a classical example of an Automatic Teller Machine (ATM) as the subject. The use cases (shown in Figure 19-2) could be the following:

- ❑ **Withdraw Cash** — A Client (as an actor) can take an amount of money out of her account in a Bank (as another actor of the ATM).
- ❑ **Transfer Money** — A Client can transfer an amount of money from her account to another account.
- ❑ **Check Balance** — A Client can find out how much money there is in her account.
- ❑ **Disable ATM** — An Administrator (as another actor) can disable the machine. The machine becomes inoperable.
- ❑ **Enable ATM** — An Administrator can enable a disabled ATM. The machine becomes operable.
- ❑ **Read Log** — An Administrator can read the log file of the ATM in order to investigate its operation.

## Part IV: Method

---

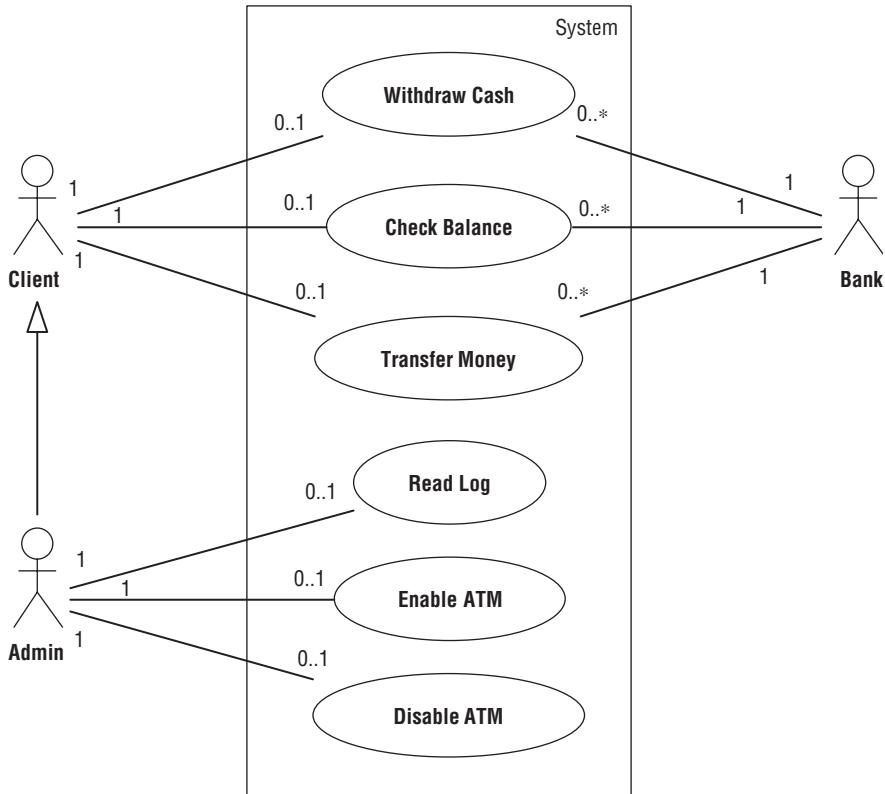


Figure 19-2: Use cases of an ATM

Use cases are specifications of functional *requirements*, not of their implementation. They describe *what* the subject is supposed to do, not *how* it will do it internally. From the use case perspective, the subject is treated as a black box, meaning that only its outwardly visible behavior is perceived and important for a use case. This helps to avoid the trap of making premature assumptions about how a unit of functionality will be implemented in the requirements engineering phase. A use case will be implemented by scenarios of interactions between internal elements of the system (its physical or logical components) in later stages of system development.

Use cases define the scenarios of interactions between the involved actors and the subject. For example, the behavior of the use case Withdraw Cash can be described as the following scenario of interaction:

1. The Client inserts her card.
2. The system checks the validity of the card using the data stored on the card, and by querying the Bank for card validity.
3. The system asks the Client to enter the personal identification number (PIN).
4. The Client enters the PIN.
5. The system verifies the PIN.
6. The system asks the Client to enter the desired amount to withdraw.

7. The Client can select one of the predefined amounts or enter a custom amount. The Client specifies the amount.
8. The system checks the available amount of cash in the machine.
9. The system performs the atomic transaction with the Bank in which it checks the balance of the Client's account and takes out the specified amount.
10. The system returns the card to the Client.
11. The system hands the cash over to the Client.

A use case can include possible variations of its basic behavior, including exceptional or error cases, alternative or auxiliary cases, and so on. The criterion for grouping scenarios into use cases (that is, which interaction scenarios constitute the same use case and which belong to different ones) is the *goal* of the use case. Each use case is targeted at a certain goal (result, outcome) that is defined in an affirmative way.

For example, the use case Withdraw Cash leads to the (positive) outcome that the desired amount of money is withdrawn. Usually, only one scenario of a use case (or a small number of them) leads to that goal. However, there may be many alternative, optional, or exceptional scenarios that do not achieve that goal. Because they represent “side paths” to the main (positive) scenario (that ends in reaching the goal), they belong to the same use case. For example, exceptional scenarios of the use case Withdraw Cash can include the following situations:

- The inserted card is invalid (for example, expired).
- The entered PIN is invalid. The Client can be allowed to enter the PIN three times at most, after which the card is taken away from the Client by the ATM.
- The specified amount of money is bigger than the current amount of cash available in the machine.
- The transaction fails (for example, because the specified amount is bigger than the current credit allowed by the Bank).

A use case should define a complete unit of functionality in the sense that, after its execution, the subject will normally be in a state in which no further inputs or actions are expected that would continue the use case, and the same or another use case can be initiated again.

In UML, a use case must have a name. Use case names should follow the capitalization guidelines for classes. A common style is that use case names have the verb-noun form, such as “Withdraw Cash” instead of “Cash Withdrawal.”

A use case can be attached to one or more classifiers that represent its subject(s). A use case does not need to be attached to any subject, however. This is the case, for example, when the subject is implied because the entire model is for one system that is the implicit subject of all modeled use cases.

A use case is a classifier and represents a description (that is, a specification) of sequences of actions interchanged between the actors and the system. A use case is instantiated (in an abstract way) at exploitation time, when a concrete interaction is initiated by an instance of an actor. In other words, at run-time, an instance of an actor is involved in an occurrence of a use case. The actor instance can enact (create) an occurrence of a use case, or simply take part in it. Anyway, instances of actors and occurrences of use cases are conceptually connected at run-time. That is why their classifiers (that is, actors and use cases) are associated in the model as shown in Figure 19-2.

## Part IV: Method

---

Actors and use cases can be associated with binary associations only. The multiplicities at the ends of associations between use cases and actors should be interpreted according to the basic semantics of multiplicities. Most often, the multiplicity at the use case end is 0..1. For the ATM example (see Figure 19-2), one particular Client can take part in at most one occurrence of the Withdraw Cash use case at one instance of ATM at a time, but she can also be disconnected from it (that is, inactive, without interaction with an ATM). This is why the multiplicity at the use case end is 0..1.

When the association has a multi-valued multiplicity at the use case end, it means that the given actor can be involved in multiple occurrences of that use case. The precise meaning of this multiple involvement is not defined in UML and depends on the specific situation. It usually means that the actor may initiate or take part in multiple occurrences of the same use cases simultaneously. For the example in Figure 19-2, a Bank can take part in many occurrences of the Withdraw Cash use case simultaneously.

The multiplicity at the actor end is most often exactly 1, as shown in Figure 19-2. A multi-valued multiplicity at the actor end would mean that more than one instance of the same actor can be involved in the same occurrence of the use case. The precise meaning of this multiple involvement is again not defined in UML, and depends on the specific situation. For example, in opening a double-key safe deposit box, a particular occurrence of the use case requires simultaneous participation of several separate instances of the same actor.

A use case is owned by a package or by a classifier. If it is owned by a classifier, that is usually the subject of the use case.

The usual symbol for a use case is a solid-outline ellipse. The name of the use case is placed inside or below the ellipse, as shown in Figure 19-2. Alternatively, a use case can be rendered as any other classifier with a rectangle having a small ellipse icon in its top-right corner, and with optional compartments for its features. This alternative is more suitable when the use case has extension points (to be explained later in this chapter).

When the subject is shown in the same diagram with use cases (as in Figure 19-2), the use cases' symbols are placed inside the subject's rectangular symbol, indicating that the use cases apply to that subject. Note that this does not necessarily mean that the subject (as a model element) owns the use cases, but merely that the use cases apply to that subject. For example, the use cases in Figure 19-2 can be owned by different packages (for example, `ClientUseCases`, `AdminUseCases`) and not by the subject itself.

Diagrams that show use cases, actors, and their relationships are known as *use case diagrams*. The diagram in Figure 19-2 is a use case diagram. Note that such diagrams do not convey any details about the behavior specified by use cases (that is, about the dynamics of the interaction between the actors and the subject). Instead, they represent a rather static view to the functional requirements of the system. The detailed behavior defined by the use case is notated according to the chosen description technique, in a separate diagram or in text, as explained later.

### Section Summary

- ❑ A *use case* is the specification of a complete unit of functionality, and describes a set of actions performed by the subject, possibly including variants, which yields an observable result of value to one or more actors or other stakeholders of the subject.

- Use cases are specifications of functional *requirements*, not of their implementation. They describe *what* the system does, not *how* it does it internally.
- A use case can include possible variations of its basic behavior, including exceptional cases, error cases, alternative cases, auxiliary cases, and so on. The criterion for grouping scenarios of interaction into use cases is the *goal* of the use case.
- Use cases are classifiers owned by packages or other classifiers (usually, but not necessarily, by the subject).
- Use cases and the involved actors are related with associations.

## ***Relationships Between Use Cases***

Each use case focuses on describing a unit of functionality with a certain goal. For complex software systems, this means that many (perhaps dozens or even hundreds) use cases are needed to cover the entire scope of the system's functionality. The complexity of use cases and the effort for their specification and later implementation can be tackled by identifying relationships between use cases and sorting them out. This can be done by identifying, factoring out, and reusing commonalities of use cases, as well as by distinguishing between basic use cases from their optional extensions.

UML recognizes three kinds of relationships between use cases for that purpose: include, extend, and generalization/specialization.

### ***Include Relationship***

An *include* relationship is a directed relationship between two use cases that indicates that the behavior of one use case contains the behavior specified in the other use case. In Figure 19-3a, the include relationship is between the use cases *A* and *B*: *A* is included in *B* (that is, *B* includes *A*). This means that the behavior of *A* is inserted into the behavior of *B* in some way.

For example, let the behavior of *A* be specified as the sequence of steps (actions between the actor and the subject):

Step-1, Step-2, Step-3

Let the behavior of *B* be specified as the sequence of steps:

Step-4, Step-5, (include *A*), Step-6, Step-7

As you can see, the specification of the behavior of *B* explicitly states the point of inclusion of the behavior of *A*. In this case, an occurrence (execution) of *B* would imply the following sequence of steps:

Step-4, Step-5, Step-1, Step-2, Step-3, Step-6, Step-7

In short, execution of the included use case is analogous to a subroutine call: When the execution of the including use case (*B*, in this case) reaches the specified point of inclusion, the entire behavior of the

## Part IV: Method

---

included use case (*A*, in this case) is executed, and the execution of the including use case is resumed once the behavior of the included use case completes. The including use case may also use the result of the execution of the included use case. The relationship itself does not identify the point and way of inclusion of the included behavior into the behavior of the including use case. Instead, this is left to the specification of the behavior of the including use case itself.

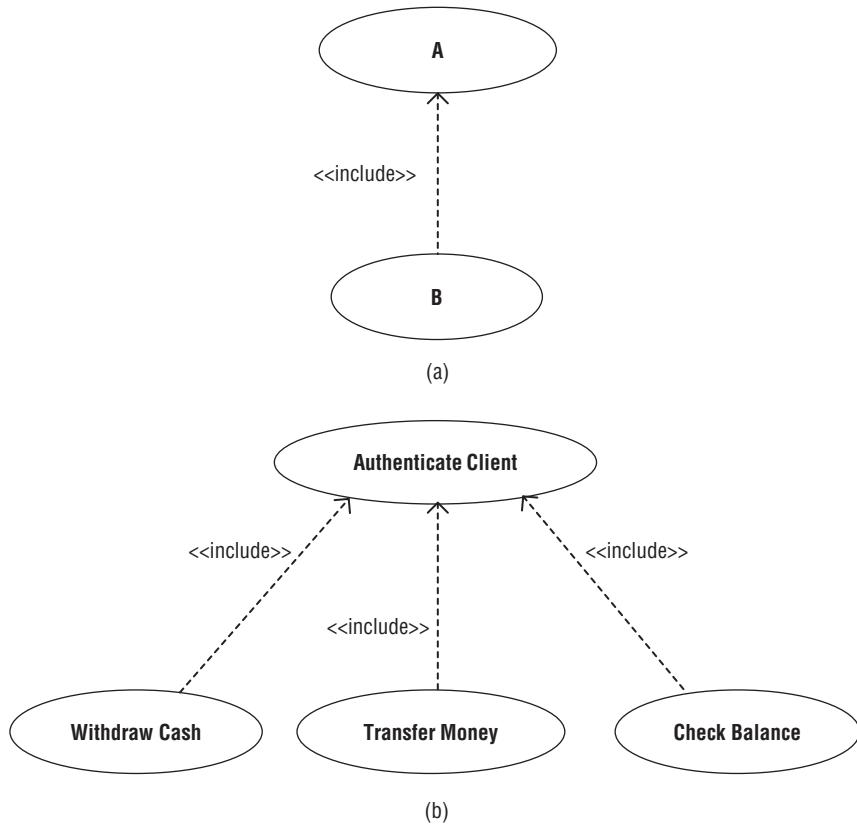


Figure 19-3: Include relationship between use cases. (a) The notation. (b) An example from the ATM system.

For that reason, the use of the include relationship is most often a result of a classical procedural decomposition. When several use cases have a common fragment of behavior, such a common fragment should be extracted to a separate use case that is then included by all other use cases having this fragment in common. The point of discovering common parts of use cases in early analysis activities is to remove redundancies in specifications and later implementation of use cases, reducing the amount of work that must be done.

For the ATM example, all use cases associated with the Client actor require authentication of the person in the same way, by checking the validity of the inserted card and entering the PIN. In other words, all these use cases, if kept unrelated, would have the same subsequence of actions of inserting the card, checking the card validity, and entering the PIN, with possible exceptional variants. To remove this

redundancy, the common behavior has been extracted into the use case Authenticate Client and that use case is included into the others, as shown in Figure 19-3b.

When common behavior fragments of several use cases are extracted in an included use case, the behavior that remains in any including use case is normally not self-contained. Instead, it depends on the included behavior to be complete and meaningful. In effect, the included behavior is not optional, but is always and unconditionally inserted in the including behavior. This is reflected in the direction of the relationship, indicating that the including use case depends on the included one. Similarly, the common behavior extracted in the included use case is usually not self-contained to be executed as an independent, meaningful, and complete use case. In other words, the included use case is most often not intended for direct activation (that is, standalone execution).

You might notice that there is a potential conflict between the need to factor common fragments of use case behavior and the fact that a use case should be something that produces a result of value to actors. Reusable use cases do not always give useful results. Instead, they are often “use-case fragments,”<sup>1</sup> as opposed to “regular use cases.” However, the requirement that every use case have value in every use case is wittingly sacrificed for the pragmatism of identifying and extracting commonalities for the sake of avoiding redundancies and reducing the work to be done.

For the example in Figure 19-3b, the use case Authenticate User is such a use-case fragment that is not intended for direct activation by the Client actor. It is simply not accessible as a standalone sequence of actions.

An include relationship is a named model element, so it can have a name in the namespace of its owning element, which is the including use case. Graphically, an include relationship is shown by a dashed arrow with an open arrowhead pointing to the included use case, labeled with the keyword «include», as shown in Figure 19-3.

### Section Summary

- ❑ An *include* relationship is a directed relationship between two use cases that indicates that the behavior of one use case contains the behavior specified in the other use case.
- ❑ When the execution of the including use case reaches the specified point of inclusion, the entire behavior of the included use case is executed, and the execution of the including use case is resumed once the behavior of the included use case completes.
- ❑ An include relationship is a result of a classical procedural decomposition. It is intended to be used for extracting common fragments of the behavior of several use cases.

<sup>1</sup>“Use-case fragment” is not an official UML term, although it does describe a real case.

## Part IV: Method

### Extend Relationship

An *extend* relationship is a directed relationship between two use cases indicating that the behavior of one use case augments the behavior specified in the other use case in some circumstances. It also specifies how and in which circumstances this is done. The behavior fragments defined by the extending use case are inserted at one or more locations in the specification of the behavior of the extended use case. These locations are called the *extension points*.

In Figure 19-4a, the extend relationship is between the use cases A and B: A is extended by B (that is, B extends A). This means that the behavior of A is optionally extended by the behavior of B at the extension point ep2 under the specified condition cond.

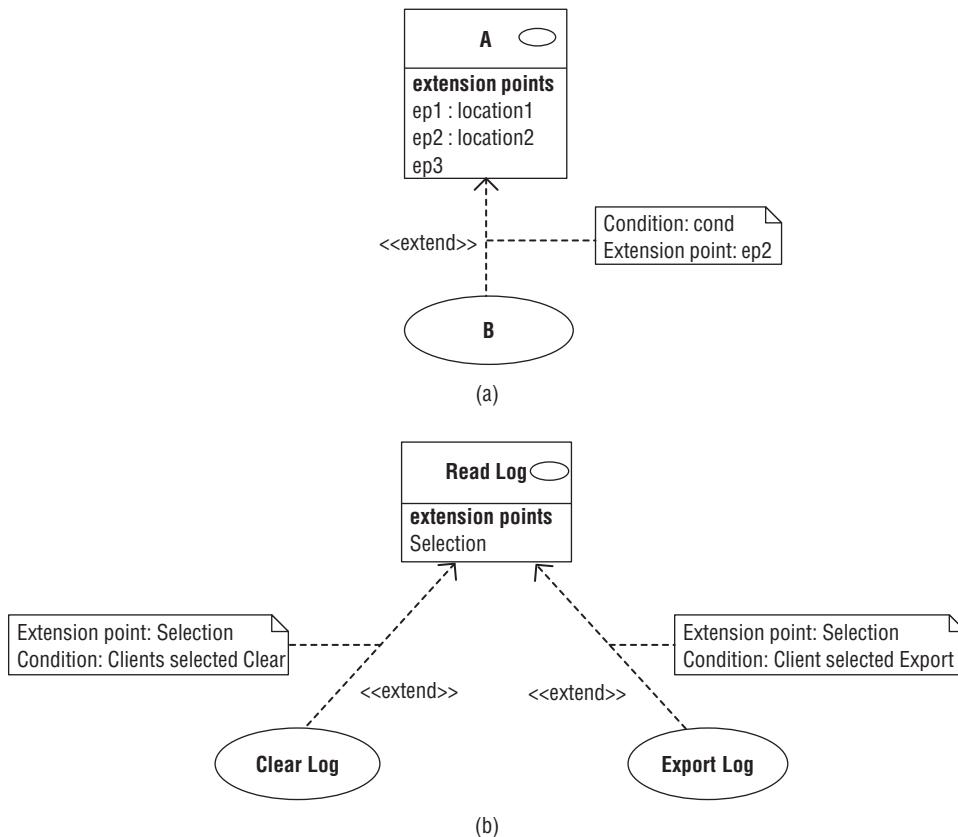


Figure 19-4: Extend relationship between use cases. (a) The notation. (b) An example from the ATM system.

For example, let the behavior of A be specified as the sequence of steps (actions between the actor and the subject):

Step-1, Step-2, (extension point ep2), Step-3

## Chapter 19: Modeling Functional Requirements

---

Let the behavior of *B* be specified as the following sequence of steps:

Step-4, Step-5, Step-6, Step-7

In this case, an occurrence (execution) of *A* implies the following sequence of steps:

Step-1, Step-2, Step-3

On the other hand, an occurrence (execution) of *B* happens when the execution of the behavior of *A* reaches the specified extension point *ep2* and the condition *cond* is fulfilled. It consists of the following sequence of steps:

Step-1, Step-2, Step-4, Step-5, Step-6, Step-7, Step-3

As you can see, extending behavior means diverting the base scenario specified in the extended use case at one or more specified extension points, by including one or more fragments of behavior specified in the extending use case. However, as opposed to the include relationship (which always means unconditional insertion of behavior), an extension is usually optional and takes place only if the specified condition is fulfilled. In addition, the behavior of the extended use case may be self-contained and meaningful independent of the extending use case (unless it extends another use case). This means that it can be activated as a self-contained and independent unit of functionality. On the other hand, the extending use case defines one or more behavioral fragments that augment the behavior of the extended use case under a certain condition. It deals with auxiliary matters that are woven in the behavior of the extended use case.

An extend relationship is used to specify an optional variant (extension) of a concrete use case, which leads to the accomplishment of a task or goal different from that of the extended use case (and, thus, does not represent a simple variant within the same extended use case). For example, in the ATM system, when the Administrator reads the log of the system, she can also be allowed to clear the log, or to save (export) it to an external device to bring it with her for later analysis. This function can be accessible at a certain point during the Read Log use case where the Administrator can optionally select to clear or export the log. Note that these two features of the system lead to certain goals of importance for the actor, significantly different from the task of simply reading the log. This may imply specific interactions between the actor and the system. This is why two use cases, Clear Log and Export Log, extend the use case Read Log at a specified extension point named Selection, as shown in Figure 19-4b.

In general, an extension point is a member of a use case that identifies a location in the use case's behavior where the behavior fragments of extending use cases can be inserted. It must have a name that is unique within the namespace of its owning use case. The location of an extension point can be specified in any appropriate (formal or informal) way. In diagrams, extension points are listed as text strings within the use case oval or rectangular symbol, as shown in Figure 19-4a. The location specification is optionally given as a string following the colon.

An extending use case typically contains the specification of one or more behavioral fragments to be inserted in the extended use case. An extend relationship defines the following:

- An ordered list of extension points belonging to the extended use case, specifying where the respective behavioral fragments of the extending use case are to be inserted** — The first fragment in the extending use case will be inserted at the place referred to by the first extension

## Part IV: Method

---

point in the list, the second fragment at the second point, and so on. Once a given fragment is completed, the execution resumes the behavior of the extended use case following the extension point.

- ❑ **An optional condition specified as a constraint in any formal or informal way** — The entire extension defined by this relationship will take place only if this condition is satisfied when the execution of the base behavior of the extended use case reaches the location referred to by the first extension point. If no condition is specified, the extension is unconditional.

An extend relationship is a named model element, so it can have a name in the namespace of its owning element (which is the extending use case). In diagrams, an extend relationship is shown by a dashed arrow with an open arrowhead pointing to the extended use case and labeled with the keyword «extend». The condition of the relationship, as well as the references to the extension points of the extended use case, may be shown in a note symbol attached to the relationship arrow, as shown in Figure 19-4.

### Section Summary

- ❑ An *extend* relationship is a directed relationship between two use cases indicating that the behavior of one use case augments the behavior specified in the other use case in some circumstances.
- ❑ The behavioral fragments defined by the extended use case are inserted at one or more *extension points*, which refer to locations in the specification of the behavior of the extended use case, only if the condition specified for the extend relationship is met when the execution of the behavior of the extended use case reaches the first referenced extension point.
- ❑ An extend relationship is used to specify an optional variant (extension) of a concrete use case, which accomplishes a task or reaches a goal different from that of the extended use case (and, thus, does not represent a simple variant within the same extended use case).

## Generalization/Specialization Between Use Cases

The generalization/specialization relationship between use cases has the same general substitution meaning as for all other classifiers: each instance of the specific use case is also an indirect instance of the general use case. Thus, the specific use case inherits the features, behaviors, and relationships of the more general use case. In other words, the general use case gathers common things of specific use cases.

For example, in the ATM system, during the execution of use cases Withdraw Cash, Transfer Money, and Check Balance, the system can provide optional online help to the Client. In other words, all these use cases can be extended by browsing the online help when the Client selects that option at a certain extension point. To avoid redundancy in specifying extensions for all these use cases separately, an abstract general use case can be introduced, as shown in Figure 19-5a. The general use case specifies the extension point and the extension, which the remaining use cases inherit.

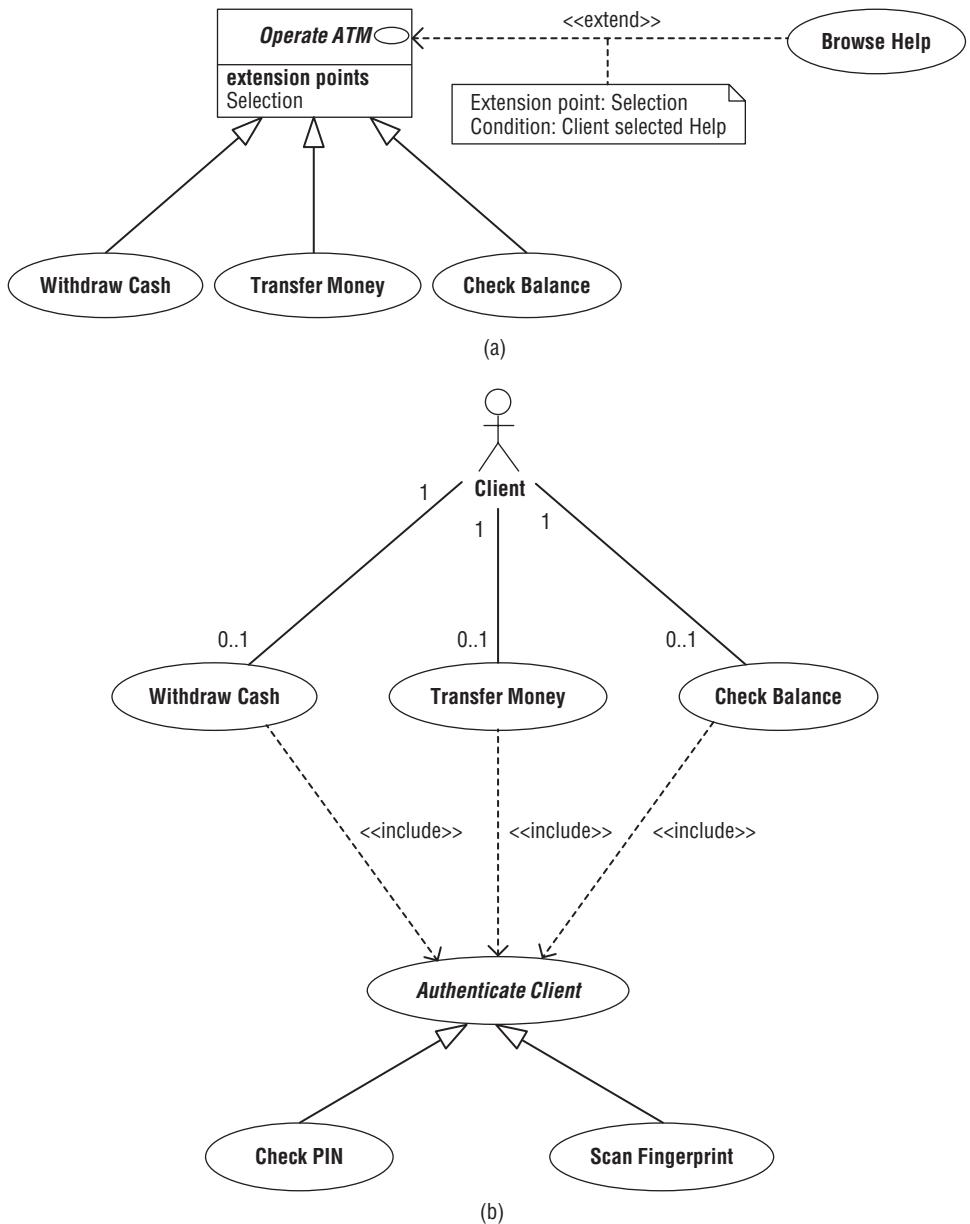


Figure 19-5: Generalization/specialization relationship between use cases

Similarly, the actor Client is associated with the use cases Withdraw Cash, Transfer Money, and Check Balance (see Figure 19-5b). However, all these use cases require user authentication on entry. Therefore, these use cases include the use case Authenticate Client as their common subpart. On the other hand, the authentication can be accomplished in several alternative ways (for example, by checking the PIN or even fingerprint scan). Consequently, two use cases, Check PIN and Scan Fingerprint, are introduced as

## Part IV: Method

---

specializations of Authenticate Client. According to the semantics of this relationship, an instance of Scan Fingerprint or Check PIN is also an instance of Authenticate Client. The former can occur whenever an occurrence of the latter is expected, in particular, as a fragment of behaviors of Withdraw Cash, Transfer Money, or Check Balance.

A specializing use case inherits the features and behaviors of its generalizing use case, but can also redefine any of them. For example, a specializing use case can redefine an extension point by defining (if it is not defined in the generalizing use case), or redefining the location of the extension point of the generalizing use case. Similarly, the specializing use case can define (if it is not defined in the generalizing use case) or redefine the behavior of the generalizing use case or any of its fragments.

### Section Summary

- Each occurrence of the specific use case is also an indirect occurrence of the general use case.
- The specific use case inherits the features, behaviors, and relationships of the more general use case, but can redefine them.

## **Specifying Use Cases**

Use cases are specified within the UML analysis model and textual requirements document. The extent and the level of detail required for the specification of use cases depends on the nature and complexity of the system under construction, as well as on the degree of formality taken in the project.

Specifications of use cases should not become overkill. They should be sized to what makes practical sense and serves the primary purpose of use cases — to specify functional requirements in an effective way so that the system's functionality can be designed, implemented, tested, and documented from that specification. A specification of a use case should not serve just itself or represent merely an artifact of someone's perfectionism. For example, if a use case is so simple that it is easier and takes less time to implement it than to specify it, its detailed specification is very questionable. As a general rule (which also holds for all other kinds of artifacts of software development), use case specifications should bring a quality to the project by carrying information that is not redundant (that is, it is not carried by or easily derivable from other more convenient artifacts).

In some projects, it may be quite sufficient just to enumerate only the most important use cases. The specifications of those use cases can be as brief as one or a few simple descriptive sentences. In some other projects, however, the same may be adequate in earlier stages of requirements capturing, but later it may be necessary to specify all the planned use cases with more or less detail.

For more detailed specifications, a use case specification template can be used. There is no standard template for documenting use cases. Instead, there are a number of proposed schemes, and modelers

and teams are encouraged to use the template that best fits the needs and nature of their projects. The available templates can also be customized for every particular development team, project, project phase, or even use case. It is more important and useful to use a consistent approach throughout a single project and team than to use a specific available template. One quite complete template for specifying use cases includes the following sections [Cockburn, 2000]:

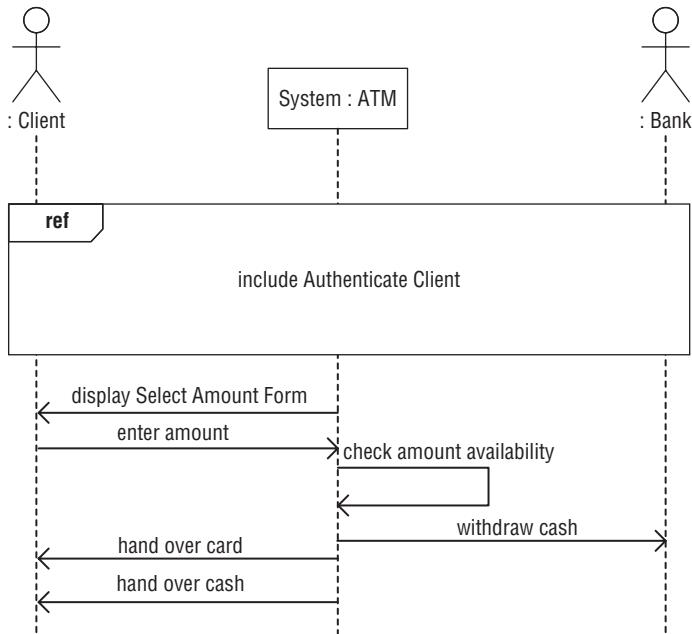
- Use case name.
- Iteration of refinement of the specification or its version number, author, and date.
- A summary, consisting of one or a few sentences describing the essence, the principal actor (the initiator of the use case), and the goal of the use case.
- Preconditions that must hold when the actor initiates the use case.
- Triggers that are the starting conditions that cause a use case to be initiated — the way the initiating actor initiates the use case.
- Basic behavior, which describes the primary scenario of interaction.
- Alternative behaviors, which describe the variations of scenarios of interaction (secondary paths, exceptions).
- Postconditions that must hold when the use case completes.
- Business rules that determine how an organization conducts its business with regard to the use case.
- Other unstructured comments (notes).

Templates may also have additional sections stating, for example, assumptions, exceptions, recommendations, performance requirements, and other technical requirements. For example, if the outlook of the user interface (for instance, a form or Web page) is part of the user's requirements for a specific use case, that user interface could also be specified along with the use case. Otherwise, this aspect should be left to the user interface design. There may also be other domain-specific sections.

The central part of a use case specification is certainly the specification of its behavior, including the basic and alternative scenarios. Those behaviors can be described in any formal or informal way, using UML modeling concepts, simple narrative text, tables, charts in other diagrammatic languages, or other idiosyncratic ways. These descriptions can also be combined.

In most cases, narrative text describing the scenario of interactions (as the one given for the ATM example) is sufficient. If UML is used to describe the behavior, interaction diagrams are usually most convenient to illustrate the scenario of interaction between the actor(s) and the subject (although other behavioral modeling elements can also be used).

For the ATM example, an interaction diagram for the basic course of events of the use case Withdraw Cash is given in Figure 19-6. The messages in the diagram are completely informal, as is the specification of the use case in a natural language. The purpose of the diagram is to convey the scenario in a compact and visual way that is easy to understand, and not to be completely formal and unambiguous.



**Figure 19-6: Interaction diagram for the basic scenario of the use case Withdraw Cash for the ATM system**

### Section Summary

- ❑ The extent and the level of detail required for the specification of use cases depend on the complexity of the system under construction, as well as on the degree of formality chosen for the project.
- ❑ Use cases may be specified using templates that can be customized for every particular development team, project, project phase, or use case.
- ❑ The central part of a use case specification is the specification of the behavior, including the basic and alternative scenarios. Those behaviors can be described in any formal or informal way, using UML modeling concepts (most often, interactions), simple narrative text, tables, charts in other diagrammatic languages, or other idiosyncratic ways.

## Managing Use Cases

Complex information systems usually encompass vast numbers of use cases. In order to be tractable, use cases should be carefully managed. First, they must be discovered. Then, they must be specified and classified. Finally, the development process should be carefully planned to prioritize the most risky and demanding use cases.

### **Business Processes and Use Cases**

The principal use cases initially emerge from the analysis of the system's *business processes*. A business process assumes a set of related *activities* that are performed among the business process's participants to achieve a specific goal of a certain importance for the business. A business process is deemed to create value for the business by transforming an input into a more valuable output. Both input and output can be physical things and/or information, while the transformation can be performed by human actors, machines, or both.

Activities assume some behavior of the participants contributing to the behavior and result of the entire process. Activities normally take some time when initiated, are performed by the participants, and are usually performed on physical things (substance, objects, documents, and so on) or pure information. Activities within a process are partially ordered, meaning that one activity may (but need not) depend on another, either by a control dependency (the subsequent activity cannot start before the preceding one has completed), or by flow of things and information between them (one activity produces things or information needed by the other).

Activities can be decomposed into partially ordered sets of more elementary activities. One step within an activity of a higher level can represent an activation of a lower-level activity. This hierarchical decomposition of activities (analogous to procedural decomposition) provides a mechanism for organizing activities into appropriate levels of abstraction and detail. In fact, taken formally, a business process can be regarded just as a high-level activity. This is really true, and the terminological distinction between a process and an activity is just for the sake of human understanding and has no real semantic implications. Which activities are referred to as processes and which are taken to be just parts of them depends on the viewpoint and level of abstraction.

Usually, a clear and simple decisive factor for promoting an activity to a process is the goal. If an activity, taken individually and independently of other activities, has a goal (result, outcome) apparently important and useful for the business, it is a process. Otherwise, it is just a simple activity as a part of a process. The rule should not be taken strictly. Analysts and other stakeholders should not waste too much time and energy in deciding what is and what is not a business process, if it is not particularly useful in the construction of the system.

At the bottom level of decomposition are activities of a business process that do not include any significant sub-steps and are not worth decomposing, even though decomposition would sometimes be possible, such as "Acknowledge the request by e-mail," "Answer the phone," "Fill in the form," or "Print out the receipt."

As an example, let's consider a national information system for issuing and managing personal identification documents, such as passports, ID cards, drivers licenses, visas, and so on. Let's call the system the ID Document Issuance and Management System (IDDIMS). One of the principal business processes in this system is certainly the process of enrollment and issuance of a document.

In very simplified terms, the business process may proceed as follows. An applicant comes to an enrollment station and applies for a document by submitting the application form along with the supporting documents. An enrollment officer receives the submission and captures the applicant's biometric parameters (such as a photograph, finger scan, retinal scan, and/or signature scan). The officer enters all the data into the system. Once the data is collected, the officer prints out the receipt and hands it over to the applicant. The enrollment phase is then over.

## Part IV: Method

---

The application then goes through the approval stage, where an approval officer analyzes the captured data, performs some background checks, and ultimately approves or rejects the application. This process may include several iterations of analysis of the submitted and collected data, as well as an interview with the applicant. Only if the application is approved, will the document personalization be started. The document personalization means printing the personal data of the applicant (including some of the biometrical data) on a document preprinted form. Once the personalization is completed, the document is ready for issuance.

In the meantime, the applicant can check the status of the application. When the applicant sees that the document is ready for issuance, she goes to the registration office to collect the document. The process of issuance can include another authentication of the applicant by matching her finger scans with the ones captured at the enrollment phase, collecting the previous document, and handing over the new document. This act is registered in the information system.

Business processes like this can be modeled using different modeling paradigms, such as classical flowcharts, Structured System Analysis dataflow diagrams, or UML interactions. In UML, they are usually modeled using *activity diagrams*, as shown in Figure 19-7. The diagram represents a part of the model that is logically a graph consisting of nodes and edges. The edges shown in this diagram are control edges that define control dependencies between the nodes — the target node cannot start before the source node completes. The semantics of (the model depicted by) the diagram are defined in terms of passing of control (formally defined as a flow of imaginary control tokens) down the edges between the nodes. Consequently, activity diagrams in UML are descendants of the classical flowcharts in their flavor and notation, but are semantically based on the formalism of Petri nets.

The oval nodes represent either elementary actions that are not further decomposed, or invocations of other complex activities. When each incoming edge of such a node offers a control (that is, a token), the activity of the node is started and the tokens are consumed. When the execution of the node's activity completes, the control is passed on, meaning that the node offers a token to each of its outgoing edges and their target nodes. Depending on the kind of these nodes, the tokens flow down or wait to be accepted by the target nodes.

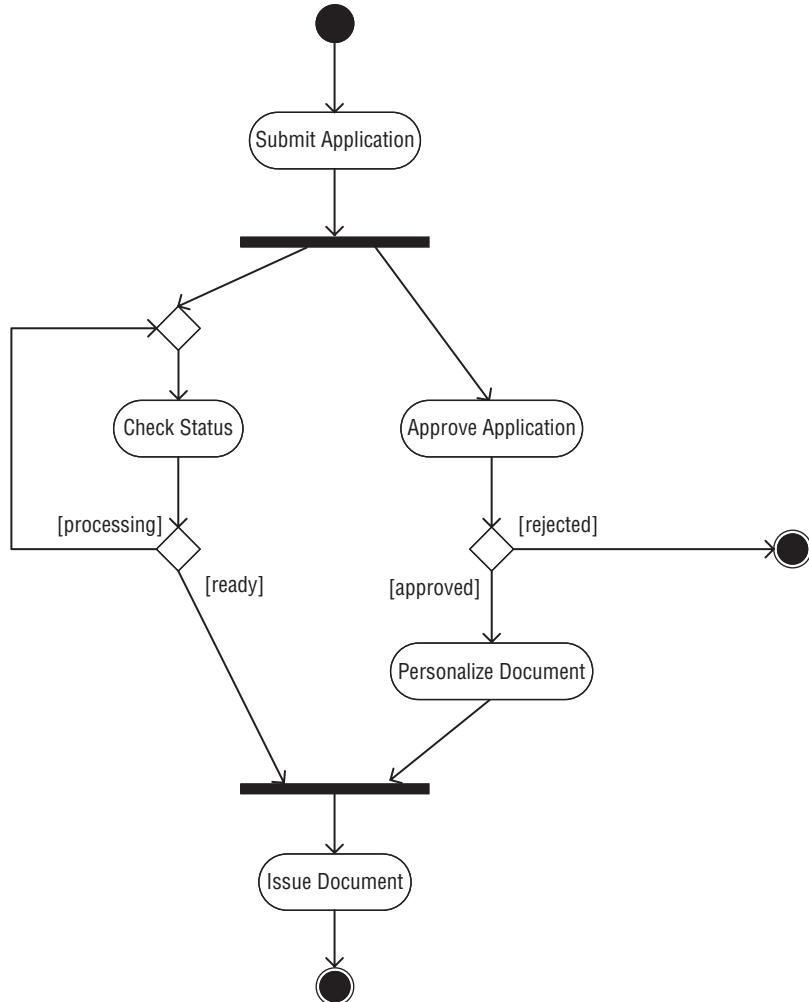
The other nodes in the diagram are control nodes that steer the control flow of the activity. The filled circle on the top of the diagram in Figure 19-7 represents the *initial* node from which the flow is started. Whenever the entire activity is invoked and started (for example, as a sub-activity of another higher-level activity, or as a standalone business process), the initial node offers one control token to its outgoing edge. An activity can have more than one initial node, each of which generates a token when the activity is invoked.

The bar with one incoming and two or more outgoing edges represents a *fork* node. It splits the control flow into two or more concurrent flows. More precisely, when it receives a token on its incoming edge, it generates one token on each outgoing edge and offers the token to their target nodes.

The bar with two or more incoming and one outgoing edge is a *join* node. It is used to synchronize two or more concurrent flows and join them into one. More precisely, the tokens on incoming edges are blocked until there is at least one token on each. When this occurs, one token is consumed from each incoming edge, and a token is generated on the outgoing edge.

The diamond-shaped nodes with one incoming and several outgoing edges are *decision* nodes that represent choices in the flow. They serve as switches of flow because each token on an incoming edge is directed to at most one outgoing edge whose guard condition results to true. There can be an

outgoing edge with the `else` guard that evaluates to `true` if (and only if) all other guards evaluate to `false`. Unlike with fork nodes, arriving tokens are not duplicated; they are just redirected. If none of the guards evaluates to `true`, the incoming tokens are stalled. At most, one guard should evaluate to `true`.



**Figure 19-7: The high-level activity diagram showing the main business process of the ID document issuing and management system**

The diamond-shaped node with several incoming and one outgoing edge is a *merge* node. All tokens offered on the incoming edges are passed on to the outgoing edge. There is no synchronization of flows or joining of tokens.

Finally, the “bull’s eye” node represents the activity’s *final* node. When any token reaches this node, the entire activity is completed, and all other ongoing flows are stopped and existing tokens are discarded. Note that the two bull’s eye symbols in Figure 19-7 represent the same final node as a single model element.

## Part IV: Method

---

UML offers many other advanced concepts for modeling activities, including object flow, structuring, and so on. However, business process, workflow, and UML activity modeling deserve special treatment, but are beyond the scope of this book. This discussion is restricted to these simple, informal explanations, and you are encouraged to explore other resources on these topics. In OOIS UML, activity or other models of business processes serve documentation purposes only, and do not have executable semantics or any particular direct effect on the system under construction. They are used in analysis models and ignored by the model compiler in design models.

In many cases, a business process includes activities that have no impact on and interaction with the system being constructed. In other words, some activities of the business process are performed by the participants other than the system under consideration (or its subsystems) when these participants interact and interchange things (for example, paper documents) and/or information that is not stored or processed by the system being constructed. However, they have an importance for the entire business process (and its goal in particular) which, in some of its parts, interacts with the system or affects it directly. An example in IDDIMS would be the activity of shipment of personalized documents from the personalization center to the issuance station. Almost entirely, it assumes transportation and delivery, but at some point, it does have an interaction with the system under construction (for example when the handovers are registered in the system).

In short, the relationship between a business process and a use case is basically a whole/part relationship. A business process is a set of related activities performed in the entire exploitation environment of the system, while use cases are only parts of it — those that directly affect (“stimulate”) the system under construction. The activities and interactions covered by use cases will affect the design of the system. The rest of them basically will not. The activities not covered by the use cases may simply include interactions between some participants (some of them possibly never interacting with the system), interchanging information, phone calls, talks, paper documents, or other things or materials (that are never handled by the system), and so on.

For example, the Approve Application activity in the described IDDIMS business process may include many participants, sub-activities, and interactions that are not covered by the Approve Application use case. The former encompasses, for example, an interview between an approval officer and the applicant, consulting the authorities (for example, court), checking supporting documents (provided as hardcopies and not stored in the system), multi-level and iterative approval by many people in charge, and, finally, entering the authorization decision (approved or rejected) for the selected application into the information system. Only the final step in this complicated and possibly unstructured procedure may be the actual use case that falls into the scope of the system being built. In other words, the scope of use cases defines the boundaries of the system’s functionality and responsibility. The positioning of use cases within business processes defines the positioning of the system’s responsibilities within the entire business environment.

### Section Summary

- ❑ A *business process* is a set of related *activities* that are performed among the process’s participants to achieve a specific goal of a certain importance for the business. Activities assume some behavior and interaction of the participants contributing to the behavior and result of the entire process.

- A step within an activity of a higher level can represent an activation of a lower-level activity. At the bottom level of decomposition are activities of the business process that are not worth decomposing.
- In UML, business processes are usually modeled using activity diagrams and interaction diagrams.
- The relationship between a business process and a use case is basically a whole/part relationship. A business process is a set of related activities performed in the entire exploitation environment of the system, while use cases are only parts of it — the ones that directly affect (“stimulate”) the system under construction.

### ***Discovering and Engineering Use Cases***

The specific way use cases are managed within the software development process depends on the methodology being followed, as well as the nature and complexity of the system under construction. In some cases, a brief use case survey is sufficient. In some others, use cases may evolve along the project from a list of a small number of principal business use cases, briefly specified with a descriptive sentence each, up to a complex and exhaustive enumeration of all use cases with detailed specifications.

In practice, I most often use the following method of discovering and engineering use cases. The method is again iterative and incremental, especially for more complex projects. The outline of the method is as follows:

- 1.** Identify and model business processes.
- 2.** Identify main business use cases from the business processes.
- 3.** Develop the conceptual model.
- 4.** Driven by the conceptual model, add use cases to cover all necessary structural manipulations of the object space defined by the conceptual model.
- 5.** Analyze the discovered use cases, decompose them, and identify relationships.

These five activities will be elaborated in the remainder of this section.

The analysis of the problem domain usually starts with business processes modeling (BPM), which is the key driver for the development of the software system. BPM may be focused on representing both the current (“as is”) and future (desired) processes of an enterprise so that the current process may be analyzed and improved. For the ongoing example of IDDIMS, let’s focus on the business process of enrollment and issuance of documents described in the previous section.

The second step assumes that you derive use cases from the specified business processes. Such use cases are referred to as the *business use cases* in these discussions. The relationship between business processes and use cases (which is used as a guideline for identifying business use cases) was described in the previous section. A rule of thumb for defining the scope and boundaries of business use cases is that, after execution of a use case, the subject should be in a state in which no further inputs or actions are expected that would continue that use case, and the same or another use case can be initiated again.

## Part IV: Method

---

For the IDDIMS example, the following business use cases can be identified from the considered business process:

- Initiate Enrollment** — Create a new application for a document, capture the data, and submit or save the application for later completion.
- Continue Enrollment** — Select an already created application for a document that has been saved, continue with capturing data, and submit it or save it for later completion.
- Approve Application** — Select a submitted application and change its status to approved or rejected, and optionally enter other data supporting the decision.
- Personalize Document** — Select the approved applications, prepare the data for personalization, start and supervise the personalization process.
- Issue Document** — Select a personalized document and change its status to “issued.”
- Check Application Status** — Search for a submitted application, select it, and see its status.

These use cases are simplified for brevity. In a real project, the activities of the considered business process can be further decomposed into sub-activities, and more fine-grained, elaborate use cases can be discovered. However, in the very initial period of requirements capturing, even such a brief and concise list can be appropriate and serve as a good starting point.

The next step focuses on developing the conceptual model from the identified business processes and business use cases. The process of conceptual modeling was described in Chapter 18. Figure 19-8 shows a simplified conceptual model that could be developed from the first analysis iteration for the IDDIMS example.

A *Person* (see Figure 19-8a) is an individual who can apply for a document and whose personal and biometric data can be captured. To separate the notion of a Person from the data captured for that Person possibly many times during the lifetime of the system, the notion of *Personal Record* is introduced. A Personal Record generalizes a set of textual or biometric data that is captured as a single data package independently from other such packages. Personal Records can be captured many times for the same Person, because the Person can apply for many documents. Whenever a new data set is captured (for example, when the Person changes her name or address, or a new photo or finger scan is taken), a new Personal Record of a specific kind is created.

The most recent (last created) Personal Record of each specific kind is linked as a *current record* of the Person. When a new Personal Record is created, the existing current record of the same kind is moved to the set of *old records* of that Person, while the new one is linked as a current record. In this way, the most recent data known by the system about the given Person can be immediately obtained, regardless of which application this data has been captured for. For example, there can be only one finger scan taken for the Person’s first application ever, while there can be a very recent photo taken for the most recent application. Separate kinds of Personal Records group the data that are changed together as a whole, independently of other data. This approach supports a clear separation of notions of a Person (as an individual with inherent identity) from the data captured and known by the system about that Person, that describe that individual (which can, in an extreme case of a criminal forgery, be false).

In addition, one Personal Record can be reused in different applications or documents without copying, if the business rules allow that. For example, a finger scan taken once can be reused in many documents through the entire life, or a photo can be reused during a period of one year, and so on. As shown in

## Chapter 19: Modeling Functional Requirements

Figure 19-8b, a Personal Record can be linked to more than one *Application for Document*, because it is used (that is, referenced by) all of them. However, exactly one Application for Document will be the one for which the Personal Record was originally created.

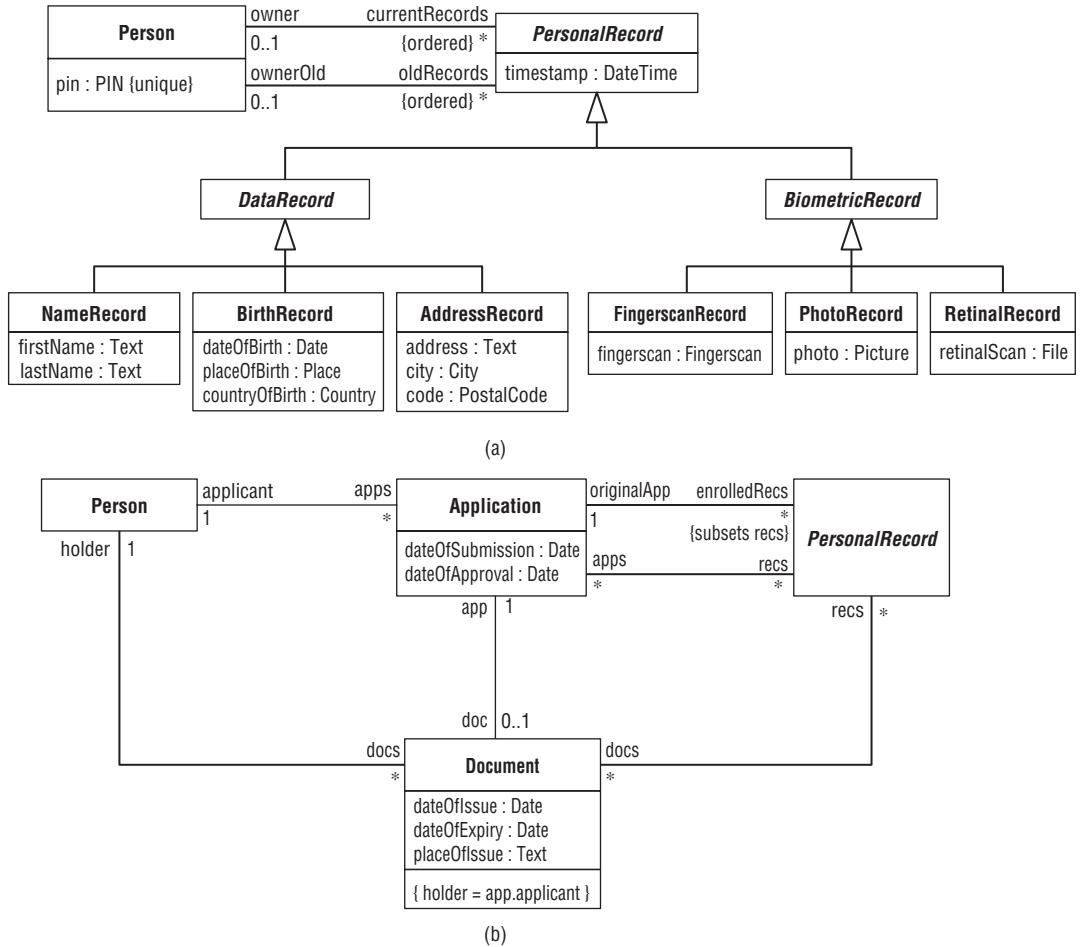


Figure 19-8: A simplified conceptual model for IDDMS

On the other hand, when an Application for Document is created and processed, only some of the Personal Records will be captured and created originally for it (*enrolledRecs*), while all Personal Records referred to by that Application (including those captured and those reused) will be linked to *recs*. Finally, when an Application for Document is approved, a Document (as an object) is created and linked to the Application and its holder, and also to all Personal Records of that Application. These Personal Records will be the source of the data that will be printed on a personalized (printed) physical document.

As you can see, Applications for Documents and Documents pass through different states during their lifetimes. For example, an Application can be created, submitted, approved, rejected, or canceled, while

## Part IV: Method

---

a Document can be created, personalized, damaged, stolen, lost, expired, and so on. This is why their lifetimes can be modeled with state machines. This discussion will not further explore the details of these models, but will simply note that there will be state machines in the conceptual model, associated with the corresponding classes.

Once the conceptual model is created (or at least one of its iterations), it can serve as a vehicle for checking the completeness of the list of planned use cases and for discovering other necessary use cases. Namely, in order to be completely useful for its users, an information system must enable the users to search, view, navigate through, and modify all the information stored in the system to which this is applicable. Note that this does not mean that every user will be allowed to do all these. The accessibility to such features will be restricted by the system's authorization mechanisms and users' privileges, but this is another issue — the system, in general, must provide such features. If the system and its authorization mechanism are properly designed, it is always easier to restrict access to some features than to implement them. Usually, this is just a matter of configuration.

Analysts should use the conceptual model as a kind of a specific checklist for determining whether all necessary navigation, creation, reading, modification, and deletion are covered by the use cases identified so far, or whether new use cases are needed. This process is known as the classical *create-read-update-delete (CRUD) analysis*. In particular, the analysts should take one-by-one a class from the conceptual model and check each of the main structural operations for that class, and each of its properties, as shown in the following table.

| Structural Element | Structural Operations to Check for Coverage                                                                                                                          |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Class              | Search for object and select it.<br>Create new object.<br>Destroy selected object.<br>View object's properties.                                                      |
| Property           | View (read value).<br>Navigate to the linked object(s) (in case of an association end).<br>Add value.<br>Remove value.<br>Modify (replace) value.<br>Reorder values. |

For each of these operations, one of the following answers could be given:

- Not Applicable (N/A)** — The operation is not applicable to the given model element. For example, adding a value to a single-valued property, removing a value from a property with the multiplicity 1..1, or reordering an unordered property, makes no sense.
- Disabled** — The given structural operation is not performed at all in the system. For example, IDDIMS could be constructed so that Persons or Documents cannot be deleted from the system because of legal constraints.

## Chapter 19: Modeling Functional Requirements

- ❑ **Generic/default** — Both the activation and the implementation of the structural operation will be done in a generic way provided by the run-time environment, such as in OOIS UML. In other words, the operation will be activated from the GUI and performed by the run-time environment in a completely generic way, as defined by the semantics of OOIS UML.
- ❑ **Generic/customized** — This means that the activation of the given operation from the GUI will be generic, but its implementation will be redefined. This will be implemented, for example, by customizing a built-in (or designing a user-defined) command and configuring it for the specific GUI action.
- ❑ **Covered by the use case X** — This means that the given structural operation will not be directly accessible from the GUI in a generic way, but will be performed by the implementation of the referenced use case, as part of, for example, a more complex structural manipulation.

The meaning of “generic” in this context is fully flexible. “Generic” denotes any approach of handling standard navigation and structural manipulations in a uniform, general, and automatic way, exploiting the underlying structural model and reflection, without need for manual design or coding. It can (but does not have to) be the OOIS UML generic GUI. For example, in IDDIMS, the system can offer a separate management application that allows the user to navigate over the object space according to her privileges. For the currently observed object (for example, a Person), the application can offer a navigation panel that leads the user to the view of the objects linked to that object over all its properties — for example, to the Person’s current Personal Records, old (archived) Personal Records, all created Applications for Documents, Documents, and so on.

As an illustration, the following is the “CRUD table” for the class `Person` from the IDDIMS conceptual model shown in Figure 19-8.

| Operation         | Means                                                    | Comment                                                                                                                                                                                     |
|-------------------|----------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Search and select | Generic/customized                                       | A Person is searched by entering the PIN or any combination of the (possibly partial) values of selected attributes of his or her current records (for example, name, address, birth date). |
| Create object     | Covered by the use case <code>Initiate Enrollment</code> | If the Person comes to enroll for the first time, an object of <code>Person</code> is created.                                                                                              |
| Destroy object    | Disabled                                                 | Persons cannot be removed from the system.                                                                                                                                                  |
| View properties   | Generic/customized                                       | For the selected Person, the form/dialog will display a selection of the attributes of his or her current records (for example, PIN, name, photo, address).                                 |

Following is the “CRUD table” for the property `Person::pin` from the IDDIMS conceptual model in Figure 19-8.

## Part IV: Method

---

| Operation                 | Means                                       | Comment                                                                                                                                      |
|---------------------------|---------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| View value                | Generic/default                             |                                                                                                                                              |
| Modify value              | Covered by the use case Initiate Enrollment | If the Person comes to enroll for the first time, an object of Person is created and his or her PIN is entered. It cannot be modified later. |
| Add/Remove/Reorder values | N/A                                         |                                                                                                                                              |

Following is the “CRUD table” for the properties `Person::currentRecords` and `Person::oldRecords` from the IDDIMS conceptual model in Figure 19-8.

| Operation                | Means                                                                               | Comment                                                                                                                                    |
|--------------------------|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| View value               | Generic/default                                                                     |                                                                                                                                            |
| Navigate                 | Generic/default                                                                     |                                                                                                                                            |
| Add/Remove/Replace value | Covered by the common part of use cases Initiate Enrollment and Continue Enrollment | When a new Personal Record is created, the current one of the same kind goes to old records, and the new one is linked to current records. |
| Reorder values           | Disabled                                                                            | These properties are ordered in the chronological order of linked records, and cannot be reordered.                                        |

Note that the described approach does not insist on explicit and formal specifications of the CRUD tables. Depending on the level of formality taken in the project, the result of the CRUD analysis can be specified in writing, or it can be accomplished mentally and just implemented. It is important, however, that the analysis is accomplished and that all the structural operations are checked against the coverage by the system’s features in order to discover use cases not implied from the business processes. If this is not done adequately, the application could suffer from the lack of management capabilities, usability, and comfort.

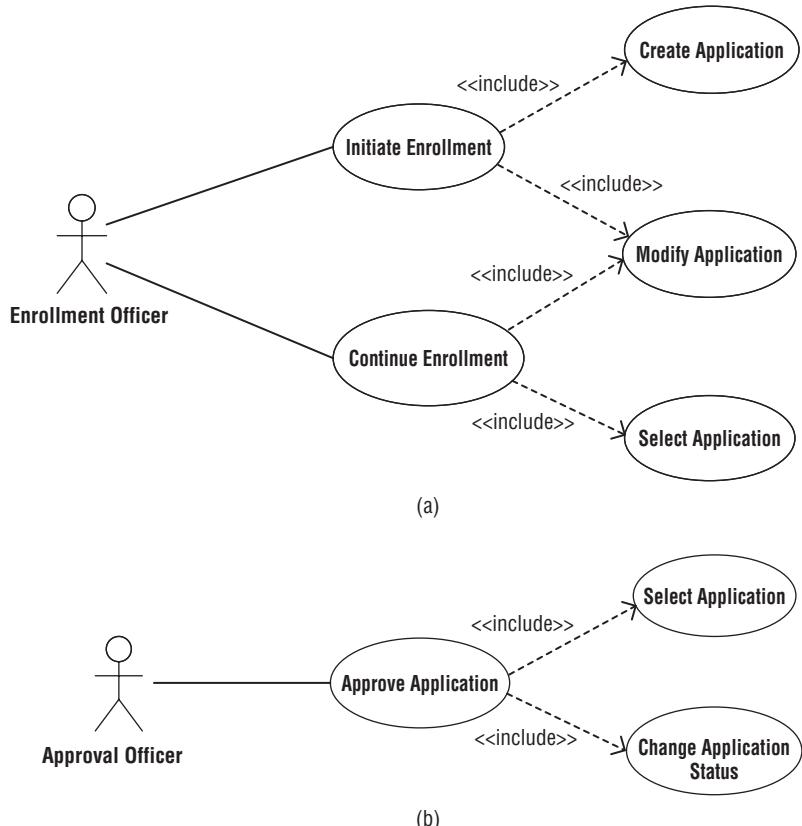
In short, the proposed method uses both the business-driven and model-driven (that is, data-driven) approaches to discovering use cases in a combined, iterative, incremental, and synergistic way. This approach guarantees that you reach a fully featured system that meets the core business needs of users, but also provides full navigability, manageability, and comfort in a balanced manner. In addition, it helps in defining uniform, consistent, and fully featured user interfaces in a more efficient and straightforward way. Finally, the approach leads to a significant reduction in the set of use cases that must be specified and implemented because of the orientation toward generic implementation.

In particular, all the structural operations that are marked as “Generic/default,” and most of those marked “Generic/customized,” do not have to be specified further. The former do not require any

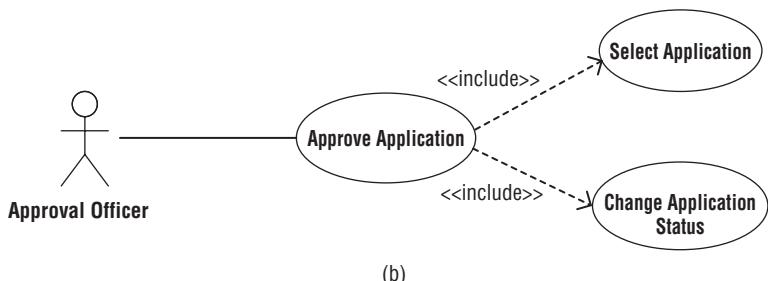
additional effort for the implementation, while the latter usually need small implementation and customization efforts. In practice, such operations usually constitute the major part (about 70–80 percent or more) of the functionality of business applications. Note that a more traditional approach would require specifying and implementing separate low-level use cases for all such operations.

As a final activity, the discovered use cases should be analyzed and decomposed by the «include» and «extend» relationships. Their common fragments are extracted for reuse.

For example, in IDDIMS, several use cases are shown in Figure 19-9. From the enrollment activity, two business use cases are identified, *Initiate Enrollment* and *Continue Enrollment*, as shown in Figure 19-9a. They are further decomposed into more elementary use cases, some of which are being reused. For example, the use case *Initiate Enrollment* is decomposed into *Create Application* and *Modify Application* that follows it. In *Continue Enrollment*, an Application is first selected (*Select Application*) and then modified (*Modify Application*). On the other hand, the use case *Approve Application* is identified from the corresponding business activity. It is further decomposed into *Select Application* and *Change Application Status*. The latter can be reused in other business use cases, too. The identified use cases are finally fully specified by describing their scenarios, explained earlier in the chapter.



(a)



(b)

**Figure 19-9: Selected use case diagrams for IDDIMS.** (a) Decomposed use cases related to the enrollment business activity. (b) Decomposed use cases related to the approval business activity.

### Section Summary

- ❑ Following is an outline of the suggested use case engineering process (iterative and incremental):
  - ❑ Identify and model business processes.
  - ❑ Identify main business use cases from the business processes.
  - ❑ Develop the conceptual model.
  - ❑ Perform the “create-read-update-delete” (CRUD) analysis. Driven by the conceptual model, add use cases to cover all necessary structural manipulations of the object space defined by the conceptual model.
  - ❑ Analyze the discovered use cases, decompose them, and identify relationships.

## Planning Iterations

The discovered and specified use cases represent the main framework for planning iterations in the overall iterative and incremental process. In each iteration, a small set of use cases is selected for implementation, testing, and delivering. More critical use cases are selected for earlier iterations.

In particular, those use cases that represent the core functionality of the system should be prioritized. These are the main business use cases, those that the system is designed for and the users are most interested in. Such use cases should be addressed early, before other less relevant and supporting use cases. For example, in IDDIMS, the principal business use cases (such as document enrollment and issuance) should be scheduled before the supporting, system use cases (such as managing users and their privileges, or the log-in function).

In addition, those use cases that carry the major risk should also be addressed early enough. Such use cases are, for example, those whose implementation includes an algorithm or another technical solution that is not fully understood, is performance-critical or resource-consuming, or requires a technology that is not familiar to the development team.

Another important criterion for scheduling use cases is that the development process should produce a first running environment as early as possible. Such an environment can be a very simple skeleton of the system and cover just its core functionality, without any unnecessary cosmetics or advanced features. The project planners should carefully select the minimal set of use cases that can cover the core part of the main business process(es), a kind of a main thread of functionality that goes from the beginning to the end of the process with minimum needed functionality and development effort.

Such an early environment can be called the *prototype* of the system, and can appear in many different forms, from simple mock-ups or click-dummies, to a completed subsystem. Some authors and methodologies even suggest discarding prototypes completely as soon as they accomplish their purpose, and not using them as the starting point for the true system development. Such an approach is not required,

however, because the proposed OOIS UML technology enables you to develop good and flexible prototypes that represent a stable foundation for the entire system. On the other hand, developing prototypes in all but very small projects is strongly recommended.

Such an approach is important and beneficial for several reasons.

First, it allows the development team to get feedback early on from the users. The early feedback is important because it can eliminate the risk of misunderstandings and help the parties avoid investing resources, time, and money in further wrong development. With the prototype, the users get the first impression and idea of their future system and its look-and-feel, and can even test its core functionalities and give suggestions for corrections or improvements.

In addition, the development team can get an early platform for easier development of other functionalities and for integral testing.

Finally, it has some important psychological side effects. Once they have tackled the core functionality of the system, the members of the development team gain self-confidence and motivation for further work. On the other hand, the users gain confidence in the developers and their capabilities. This is very important for building mutual trust between the parties. It can significantly reduce the impact of the usual mid-project morale-dip problem, too.

After the prototype has been delivered as a result of the first or few first development iterations, later iterations can also deliver partially completed systems. It is recommended that these incremental deliveries be performed over a short period of time, between a couple of days or weeks for smaller projects, and a couple of months for very large projects. In general, functional increments implemented and delivered in later iterations usually represent one of two types:

- ❑ **The start of the implementation of a new set of use cases** — Such iterations are generally more critical. For that reason, the use cases that are addressed in them should be carefully selected. The set of newly addressed use cases is usually small, and only the basic scenarios of these use cases are implemented.
- ❑ **The completion of the implementation of the started use cases** — In these iterations, the auxiliary and exceptional variants of the already partially implemented use cases are completed. In general, such iterations are less risky and problematic.

### Section Summary

- ❑ In each iteration of the development process, a small set of use cases is selected for implementation, testing, and delivering. More critical use cases should be selected for earlier iterations.
- ❑ An important criterion for scheduling use cases is that the development process should produce a first running environment, or a prototype, as early as possible. Prototyping has many benefits — both technical and nontechnical.
- ❑ An agile development process assumes short development iterations with frequent deliveries of functional increments.



# Part V

# **Supplemental**

**Chapter 20:** Characteristics of Information Systems

**Chapter 21:** Process and Principles of Software Development

**Chapter 22:** The Relational Paradigm

**Chapter 23:** Structured Analysis

**Chapter 24:** Introduction to the Object Paradigm



# 20

## Characteristics of Information Systems

This chapter describes the characteristics of software for information systems. Typically, the characteristics of (and requirements for) a successful information system may be classified into three major categories: domain-related, usability-related, and deployment-related.

### Domain-Related Characteristics

Each information system covers a certain problem domain, often very specific and significantly different from other problem domains. For example, the problem domain of a healthcare information system is different from the domain of an accounting system. They deal with different concepts and they have different functionality. Additionally, even systems from the same problem domain constructed for different customers may have different functionalities and complexity.

#### ***Complexity***

One of the major characteristics of all but trivial information systems is their essential complexity. Although even different implementations of systems from the same problem domain may have different accidental complexity, their essential complexity is almost always large. It is not unusual for a system to have hundreds of units of functionality and dozens of interacting software modules of high complexity. (However, note that, in many existing systems, a major part of that overall complexity is accidental, unfortunately.)

Sometimes, however, a system to be constructed seems to have small essential complexity. This may be true at the beginning of the system's exploitation, but if the system is successful and useful, it will almost certainly encounter further improvements and extensions that will increase its complexity. Almost all successful systems that live longer encounter new demands for more complex functionality. Therefore, information systems are complex systems by default. This represents the key obstacle for their construction, a solution to which is the central point of all software engineering methods and tools.

## Part V: Supplemental

---

The main part of this book is almost completely devoted to the discussion of the solution to this problem — that is, how to reduce accidental complexity and deal more successfully with essential complexity.

### Conceptualization

Each problem domain relies on some concepts that form the vocabulary of the domain. These concepts are called the *key abstractions* of the domain. An abstraction is the essential characteristics of an entity that distinguish it from all other kinds of entities. For example, in a school information system, the entities *course* and *person* are significantly different from each other, as well as from other entities in the system, and are, therefore, abstractions.

An abstraction is the result of neglecting some differences from entities in the real world that are irrelevant for the particular purpose, in favor of generalizing their commonalities that are relevant. For example, you do not need to distinguish between the concrete courses that will exist in the school information system according to, for example, the average number of words that are pronounced by the teachers during the courses, or the number of books that are given out to the students for the courses, simply because they are not relevant or needed by the users of the system.

On the other hand, you emphasize the commonalities that *are* important for the users of the system, such as the fact that each course has its name or duration. Similarly, a person in the real world is almost an indescribable ontological notion — we all have our name, birth date, hair color, shoe size, nose shape, marital status, and many, many other properties that make us identifiable entities in the real world. But almost all of these properties are simply irrelevant for the school information system. The system probably simply needs to store names and a few other bits of administrative data about persons.

Therefore, an abstraction is a *simplification* of a real-world entity, with a boundary defined relative to the perspective of the viewer. It is obtained by reducing the information contents from the real world in order to retain only the information that is relevant for the particular system. This process of simplification by focusing on what is important is one of the essential tools that helps humans to make the inherent complexity of problem domains more tractable.

A key abstraction is an abstraction that is important enough to be incorporated into the system's conceptual model, regardless of its concrete implementation. In other words, for the sample school information system, no matter what the implementation will look like, the system should enable the user to manipulate with courses and persons, and to maintain their properties. That is why *course* and *person* are key abstractions in the school information system's problem domain.

The set of key abstractions, their semantics, responsibilities, properties, and relationships, precisely defined using a specific language, is called the *conceptual model* of the given problem domain. The process of defining the conceptual model (called *conceptualization*) is one of the central activities in building information systems.

Building the domain's conceptual model for a new information system is the first critical step in analyzing requirements. It brings considerable risk to other elements of the system, most notably to its functionality. Namely, the information system can only be useful if the conceptual model is properly defined and if it expresses the right users' needs and the real state of the problem. If the conceptual model does not include a certain key abstraction needed for a certain functionality, there is no way to support that functionality in an acceptable way.

There is no functionality or user interface that can repair a significant flaw or deficiency of the conceptual model. For example, if the mentioned school information system does not distinguish the concepts of *teachers* and *students* in some way, the system will not be able to support the functionalities that need to be separate for the two, such as to find all the teachers that teach a certain course. If, on the other hand, the conceptual model clearly defines concepts and crisply separates their concerns, the system is likely to be flexible enough to meet future modifications and enhancements of its functionality.

A system analyst (that is, a person who has to analyze the problem domain by observation and interviews with its participants, and define the conceptual model) thus has a very difficult and responsible task. The task is difficult because there is almost never a single solution to a problem, even for one particular problem domain. The system analyst must identify abstractions, discover possible alternatives in defining the conceptual model, and find out which one is most suitable for particular user's needs. Consequently, the process of conceptualization is very sophisticated and risky. Chapter 18 is devoted to the process of developing information systems, and conceptualization in particular.

Because a conceptual model should be formally defined in order to build software from it, the vehicle for specification that is available to a system analyst is very important. This vehicle is called the *language for conceptual modeling*. The language represents the tool that is available to the system analyst to record requirements of the users and specify how the system to be developed should look. The language should be expressive enough and conceptually close to the problem domain, so that the system analyst does not need to expend unnecessary effort in making the specification.

On the other hand, the language should be simple enough to be understandable by users and developers. It is also desirable that the specification made in that language can be mapped automatically or semi-automatically to the system's implementation. In other words, the language should be carefully balanced. A major part of this book is devoted to these issues.

### **Large-Scale Dynamic Instantiation**

While being exploited, an information system is populated by lots of *instances* of the abstractions from its conceptual model. For example, the school information system stores information about concrete persons and courses: John Smith, Vanessa Vanderbilt, or "Programming in C++" are all instances of the corresponding key abstractions. Usually, information systems encompass at least dozens (but more often hundreds, thousands, or even millions and billions) of instances of certain abstractions. Most often, the number of instances is not limited by any predefined value, but can be simply estimated according to certain natural characteristics. For example, it can be expected that a banking system may contain hundreds of instances of the bank account abstraction, or that a governmental system may contain millions of instances of the citizen abstraction, but the expected number or upper limit is rather estimated than precisely known in advance.

In addition, instances are predominantly created and destroyed dynamically, during exploitation of the system, according to the needs of their users.

### **Functionality**

Functionality defines how the system is expected to perform (that is, behaving and reacting on stimuli occurring from its environment). Until recently, functionality was expressed in terms of *function points*, whereby a function point represents a unit of functionality that can be identified as important for the user, and that assumes the reaction to a certain stimulus from the system's environment. For example, one

## Part V: Supplemental

---

function point of an information system for a book publishing company may be the system's capability to answer the question, "How many samples of a certain title have been sold so far?" Recent approaches have proposed advanced concepts for expressing functionality, such as use cases explained in this book.

Functionality is the characteristic most visible to the users and developers of an information system. It directly affects the usability of the system. For the example of a book publishing company, the system should probably provide the information about the stock amount of a certain title as one function point. However, the same system is much more useful if it also gives the average selling rate of the same title and calculates the expected duration of the stock, so the sales manager can make a proper decision about reprinting the title at the right time.

Additionally, functionality is usually by far the most complex aspect of system development. Even small information systems have very complex functionality, expressed in a huge number of functional points, so that they require a significant effort even to be specified. Therefore, many methodologies have focused on techniques to cope with that complexity. Although sometimes very different, the methodologies are all based on another fundamental principle of coping with complexity apart from abstraction — *decomposition*. The methodologies differ in the concepts and approaches that support decomposition of functionality.

## ***Evolution***

Each information system faces constant demands for modifications throughout its exploitation. These demands have very different sources. First, software designers often misunderstand their users, or simply make mistakes when building software. Software is rarely realized completely correctly at the first shot and, therefore, should be changed later. Second, users constantly require new features, improved usability, or better performance of a system. Third, circumstances change and force adaptations of the system. For example, changes in tax laws affect the calculation algorithms built in an accounting system, or modifications in the organizational structure of a company may need to be accompanied by appropriate modifications of the information system. Fourth, permanent advances in hardware and software technology force systems to keep up with new trends. For example, the emergence of Internet technologies has had a great influence on the architecture of information systems and their development approaches.

Although it is one of the most challenging aspects of system development, evolution is not a destructive or undesired characteristic. On the contrary, it may be regarded as a natural tendency of all successful systems. It is often said that almost each successful system has been built on top of another system that worked well [Booch, 1999]. Therefore, a successful system inevitably evolves over time.

### **Section Summary**

- ❑ Domain-related characteristics are associated with the system's problem domain:
  - ❑ Complexity
  - ❑ Conceptualization

- Large-scale dynamic instantiation
- Functionality
- Evolution

## Usability-Related Characteristics

Information systems are developed to be usable and useful. Thus, they have a number of important characteristics, as discussed in the sections that follow.

### ***Interactivity***

Information systems are predominantly *interactive* (or *reactive*) systems, meaning that their main purpose (or one of the main purposes) is to interact with their environment and react to external stimuli during their execution. In other words, information systems are not *transformational* systems, whose main purpose is to accept some input, perform some computation, produce some output, and complete their execution afterward.

Although transformational systems might still interact with users for one reason or another (for example, in cases of reporting and handling errors), this is usually not their main purpose. For example, a scientific program that finds a numeric solution to a given differential equation may be a transformational program. On the other hand, information systems run for longer periods and constantly react to external stimulation during that time. For example, a banking information system constantly responds to requests from ATMs and performs financial transactions according to those requests.

Information systems are generally *transaction-based* systems, which means that such a system accepts a request from a user, performs some computation, and provides some answer in a sequence, whereby such a sequence may affect the state of the system. For example, a transaction initiated by the user of an ATM includes a request to withdraw some cash, return the cash, and change the balance in the user's account. What is also important is that information systems often perform huge numbers of transactions concurrently. For the mentioned example, hundreds of transactions may take place at the same time, from different ATMs connected to the same banking system.

Although information systems (or, more precisely, some of their subsystems) often interact with other computer-based systems, their ultimate users are generally humans. More precisely, the source of the stimuli depends on where the boundary of the system is placed. For the mentioned example of a banking system, if ATMs are assumed to exist outside the boundaries of the core banking subsystem, the core subsystem interacts with another computer-based system (the ATM). However, if the entire system is observed as a whole, its sources of stimuli and consumers of results are humans (the users of ATMs).

Therefore, the user interface of an information system plays a key role in its usability, as will be discussed later.

### ***Appropriateness***

*Appropriateness* is a quality of a system that ensures that the system provides the information precisely needed by its users. An information system should not offer just any piece of data to its users, but the

## Part V: Supplemental

---

one that is useful. For example, in an information system for registering working hours of employees, different types of users are interested in different kinds of information. A manager of a department is interested in the total sum of all regular work time and overtime for all employees in that department, and one particular employee may be interested just in his or her own work time. Note that both kinds of information are based on the same core factual data — the records of the check-in and check-out time of all employees in a certain period.

### **Timeliness**

*Timeliness* means that a system should provide the proper information at the right time, just when it is needed. Although time is not the primary factor in using most information systems, it is usually not negligible. For example, the success of a stock-market broker highly depends on whether the system provides the needed information at the right time; if the system is slow and delays the information, it has not accomplished its purpose.

### **Availability and Location Independence**

*Availability* means that information should be available whenever needed by users, regardless of when that need occurs. For example, in a hospital information system, the information about the blood group of a patient in critical condition should be available 24 hours a day and 7 days a week, regardless of the business hours of the IT department. Likewise, the system would be useless if that information was unavailable because the system administrator was performing a backup, or the system crashed because of a bug in software or a failure of hardware.

*Location independence* means the availability in space. Some information systems are used on a wide area, and many systems nowadays are used worldwide through the Internet. There are numerous examples of Web applications that provide services to customers all over the world. In addition, it's not surprising that managers of large companies would like to get information about their businesses during their intercontinental flights using on-board computers connected to the Internet.

### **Security**

*Security* ensures that information is provided only to those users who are authorized to possess the information. For example, a military information system must be protected enough to prevent unauthorized access to confidential data about military resources and plans.

Typically, security includes the following:

- **Identification (or authentication)** — This assumes that the system must check whether a user really is whom he or she claims to be. There are many techniques for identification. The most widely used is the username/password approach, but there are also other more sophisticated techniques based on biometrical data (for example, retinal or fingerprint scan).
- **Authorization** — This means that the system should provide only the information that the user is authorized for, and prevent access to any other information. Authorization usually assumes defining *user access rights*, which are settings that define to which operations, data, or features of the system the user does have access.
- **Encryption** — This transforms information so that unauthorized users (who intentionally or accidentally come into its possession) cannot recognize it.

Although security plays an important role in contemporary information systems, it is outside the scope of this book.

### ***Ease of Manipulation***

Information systems must enable simple manipulation with their contents. A system should not force the user to provide more stimuli to accomplish an operation than necessary. Many aspects support such behavior, and it is impossible to list all of them. For example, a system of a trading company should not force the user to remember and enter idiosyncratic codes that are used for merchandise as the only way to select a product. Although this might be the only attribute that identifies a product uniquely, the system should allow the user to search for the product code using another attribute that is closer to the human thinking, such as the name of the product or its producer. Additionally, the method of manipulation should be consistent at all places in the user interface. For the same example, it would be problematic if the system allowed the user to select a product using different search criteria when filling an order, and not to support the same option upon conducting a stock inventory. Even worse, the system might be confusing for its users if the same operation (such as selecting a product) is performed in completely different ways at different places (for example, using the product code in one place and its name at another) unless it is particularly wanted by users and fits their needs.

Consequently, consistency in using a system is one of the most important aspects of its usability. Thus, the development method must pay particular attention to this issue. Methods and tools for development must provide mechanisms for encouraging developers to use the same approach for similar operations at different places, and help them in not expending additional effort if the same thing is to be implemented at several places.

Although the user interface is the most important aspect of a system in terms of its usability, other parts are not negligible. If a certain feature of a system is not implemented correctly, or if some data is not structured and stored in a proper manner, even the best user interface in the world cannot fix that.

Modern modeling methods, frameworks for user interfaces, and development tools have played an important role in improving system usability. Contemporary graphical user interfaces (GUIs) have set very high standards in usability of all computer-based systems, and information systems in particular. However, there is always room for improvement, and much effort is being made to develop the theory of system usability, which may likely require a whole book to elaborate.

### ***Source Versus Derived Information Trade-offs***

Information systems should provide the right information needed by each particular category of users. Very often, different kinds of information are actually derived from the same source data that record certain events or facts from the system's environment. In such cases, it is very important to consider allowing access to, and manipulation of, that very basic fact, probably not to all users, but only to privileged ones. In other words, it is often more convenient to store the basic factual data in the system's persistent storage (that is, database) and to compute the derived information for different kinds of use than it is to store only those derived values.

This is particularly important when the computation is irreversible (that is, when the source data cannot be restored from the derived values). There are several reasons for this. First, keeping several derived values instead of a single source one generally (but not always) requires more storage space. Second, having several related values derived from the same basic fact requires significant effort to keep values consistent: If one value is modified, the others must be updated, too. Third, it is likely that the system

## Part V: Supplemental

---

would encounter requirements for new kinds of information based on the same factual data. In that case, it is much more likely that the new feature can be implemented only if the system really provides the access to (that is, stores) the actual source values.

Let's consider an example of a system for registering work hours of employees in a company. The card readers that are installed at the entrance of the company building provide the source factual data to the system: the records of check-in and check-out time of employees. The users are primarily not interested in that data, but in the derived information, such as the total sum of regular work time and overtime of a particular employee or a whole department.

An implementation of the system might opt for the approach where the source records are not stored in the system's database, and where the daily sum of the work time for each employee is stored instead. The rationale could be that only the derived information is needed, and that it needs less space (an employee may generate several records on the same day). However, this solution would probably be problematic.

First, it is likely that the users will want another feature very soon, which is impossible to implement without having the source records. For example, they might ask for a report on the average number of exits that an employee makes a day. Second, in case of an error (for example, when an employee forgets to check out), the system is unable to make the right calculation without an authorized intervention of the administrator on the source records. This is an example of irreversible computation — the check-in and check-out times cannot be restored from the values of total work times for an employee.

However, there are sometimes arguments for the opposite alternative. First, if the computation of derived information is costly, it may be a better solution to store the derived values. Second, if a system is distributed, it may also be better to keep derived information in local copies or fragments of the persistent storage, instead of constantly downloading the source data from the central storage. The latency of data transfer over communication connections and fault-tolerance reasons may give preferentiality to keeping derived values on local copies of the database.

Anyway, this issue is often an important design trade-off. Although it may look like an exclusively implementation-related issue not important to users, it is not. The opted alternative may significantly affect the usability of a system. This is why users often must be aware of its consequences.

### Section Summary

- ❑ Usability-related characteristics are associated with how a system is exploited and how usable it is:
  - ❑ Interactivity
  - ❑ Appropriateness
  - ❑ Timeliness
  - ❑ Availability and location independence
  - ❑ Security
  - ❑ Ease of manipulation
  - ❑ Source versus derived information trade-off

# Deployment-Related Characteristics

Deployment-related characteristics are associated with technical issues of implementation, deployment, and exploitation of a system. Although they may appear as unimportant to users, they often affect the performance and availability of a system. This is why users must be aware of their consequences. In any case, they are very important to developers.

## Diversity and Quantity of Data

Information systems typically deal with large amounts of data of different kinds and with diverse meaning. This may be one of the key properties distinguishing information systems from other interactive systems. For example, a file-management system may deal with a large number of files, but it generally does not recognize too many concepts (that is, it recognizes just the concepts of a file, a folder, and access rights). Similarly, a word processor knows about characters, words, paragraphs, formatting settings, spelling and grammar, and possibly many other concepts, but its scope is typically bound to a single document. Therefore, these examples of interactive applications are not treated as information systems in the scope of this book.

On the other hand, for example, a governmental information system deals with a diversity of concepts, such as people, different kinds of personal documents (for example, ID cards and licenses), governmental organizational units, legal documents, and so on. Furthermore, each of these concepts implies a huge quantity of data in the information system, like millions of records for people and several times more for their personal documents. In general, hundreds of thousands or millions of records of a certain kind are quite expected in information systems. Information systems must deal with such diversity and quantity of data effectively.

## Scalability

Some information systems may have a large number of users working with huge amounts of data, as described in the previous section. What might be even worse, the number of users who access the system at the same time may be unpredictable during development, while the volume of data can grow over time. For example, an e-commerce Web trading system may be accessed by a virtually unlimited number of users at the very moment of its publishing on the Internet, constantly increasing the volume of data stored in the system. This implies a huge number of concurrent transactions requested from the system over a huge and increasing database. The system must respond adequately because an absence of a proper response may result in users giving up. The hardware and software must deal with this problem effectively. Of course, any real hardware and software have limited throughput and may, therefore, serve just a limited number of transactions in a certain period.

However, the point is that hardware and software should be engineered in such a way that allows throughput to grow quickly when the demand grows, by investing in more hardware. More precisely, software and hardware architectures should be *scalable*, meaning that investment in hardware may ensure greater throughput, if there is a demand for it, without any significant modification of the hardware and software architecture.

Theoretically, an ideal scalability means that the throughput of system is linearly correlated with the amount of hardware processing resources. Of course, in practice, the ideal scalability is difficult to achieve, but developers must strive for this property and try to engineer architectures that achieve the closest-to-linear possible scalability. For example, a scalable architecture of the mentioned Web trading system may encompass layering the software into at least two separate layers: one for receiving

## Part V: Supplemental

---

concurrent requests from the clients through the Web and delivering them to the second layer, and the other for processing the requests. If the former may schedule the requests so that they are performed by the latter on separate servers, a better throughput may be achieved by simply providing more servers that process the requests in parallel.

## Persistence

Information systems deal with *persistent* data, meaning that the data may survive termination of each particular application execution. For example, many users of a librarian information system may activate and terminate the application for searching or ordering a book title. However, the records of books and library members are kept in a persistent storage of a computer system so that they remain accurate for new activations of the application.

Data processing takes place in a computer's processor(s), which can access directly only the computer's operating memory, which is a volatile storage. Because the contents of operating memory are discarded upon application termination, persistence of data assumes that you keep the data in a persistent storage, such as disks. Although many other kinds of interactive applications (such as word processors, spreadsheets, or personal organizers) also deal with persistent data, information systems data persistence has certain unique properties.

First, information systems applications do not deal directly with data organized in files, whose internal structure must be managed by the application itself. As opposed to information systems, for example, word processors or spreadsheets typically store their data in files, and manage internal data structures in the files in their own, application-specific manner. Data handled by information systems is organized into more complex and more general structures. Ultimately, the data may be stored in files on the disk, but the internal structure of the files is not directly visible to the application.

In general, users and applications should be protected from having to know how data is organized in the machine (the internal data representation). Activities of users and applications should remain unaffected when the internal representation of data is changed, and even when some aspects of the external representation are changed. For that reason, there is an intermediate layer of logical data structure, organized in a generic manner, so the structure can be reused by applications in different problem domains. The intermediate data structure is managed by a software system called the *database management system* (DBMS).

A DBMS manages the mapping of a more abstract data structure, visible to the application, into a lower-level internal representation, such as the computer's file system or its proprietary data structures on a raw partition on the disk. Additionally, a DBMS supports generic operations and queries upon that abstract data structure, in a way that the application does not have to worry about how the raw bits and bytes of information are accessed and stored on the disk.

The actual structure of this abstract data layer, along with the set of generic operations upon it (accessible to the application) forms the database paradigm. During the history of computing, different database paradigms have been used and superseded, such as hierarchical, network, relational, or object databases.

Second, other applications with persistent data often work on the "document" basis, meaning that they load the entire data space from a file into the operating memory and work with it from then on, without any interaction with the persistent storage, until the document is explicitly stored to the file. In other words, there are explicit operations of "load document" and "store document" that transfer the entire

data space from and to the persistent storage. Between these two operations, the application works only with the data in the operating memory, and without mapping of the data space into the persistent structure.

On the other hand, information systems cannot generally apply this strategy because of the size of the data space. Instead, information systems work on the “transaction basis,” meaning that they keep consistency of operating memory and persistent storage by sporadically loading parts of the entire data space from the persistent storage, or by updating the persistent storage from the modified operating memory. In other words, information systems work with only parts of their entire data spaces loaded in operating memory at a certain moment — those that are currently needed by the application. Such parts are called the *working sets*.

### **Concurrency Control**

Actions of users or other outer systems that stimulate an information system typically occur asynchronously, sporadically, and concurrently. The actions are *asynchronous*, meaning that they occur in an unpredictable manner. Second, they are *sporadic*, meaning that they may occur in irregular time intervals (that is, without periods). Finally, they are *concurrent*, meaning that stimulations with certain duration may take place at the same time, in parallel.

For example, several users may place their orders to an e-commerce information system, or withdraw cash from their bank accounts in a banking system, simultaneously. This is an inherent property of information systems, as opposed to some other computer-based systems, such as time-driven real-time systems, where events may occur or tasks should execute periodically, at defined time rates. Therefore, information systems must be engineered to accept asynchronous, sporadic, and concurrent stimuli.

On the other hand, it is not sufficient that information systems merely accept asynchronous and concurrent stimuli, but they should also incorporate concurrency of their internal processing. For example, it would be inconvenient if the software of the mentioned e-commerce system required that processing of one order wait for another, totally independent one, to complete, even though the hardware allows their simultaneous processing. In other words, the software should be engineered to support concurrent processing.

Moreover, software should also try to exploit as much concurrency as possible. This generally means avoiding unnecessary synchronization of different processes wherever possible, although asynchronous execution is generally more difficult to achieve and control. More synchronization generally implies less concurrency. For example, in the mentioned e-commerce information system, when a user has placed an order, processing of that order may continue concurrently with other further activities of the same user. The same user may place other independent orders, or activate other functions of the system, while the previously placed order is processed in the background.

The consequence of such orientation to concurrent processing, is very important. If software is engineered to support concurrency, its concurrent processes can be executed simultaneously (physically at the same time), provided that hardware allows that. Consequently, such systems are more likely to be scalable, because parallelism of processing (which generally implies better throughput) can be achieved by adding hardware without modifying the software architecture.

Now, let me emphasize the slight conceptual difference between the terms *concurrency* and *parallelism*. “Concurrency” means that a system (or, more precisely, software) is partitioned into tasks (or processes)

## Part V: Supplemental

---

that can (but need not) be executed physically at the same time. “Parallelism” means that the tasks are really executed physically at the same time (that is, simultaneously). This is why parallelism is also sometimes referred to as *physical concurrency*, while concurrency is also referred to as *pseudo-concurrency* or *logical concurrency*. Of course, parallelism is possible only if there are hardware processing elements that allow parallel execution of tasks, but also if software allows (logical) concurrency. Parallelism (that is, physical concurrency) implies (logical) concurrency, but the opposite is not necessarily true. For example, if there is a single processing element, software that is engineered to be concurrent will be executed sequentially. Concurrency means potential parallelism.

However, concurrency often eventually leads to the necessity of synchronization. Consider the following simple example. Assume that the application of a banking information system has a procedure for withdrawal of a certain amount of money from a bank account. In principle, the procedure may look like this:

```
procedure cashWithdrawal (ba : BankAccountID; sum : real)
begin
 var currentBalance : real;
 readAccountBalance(ba,currentBalance);
 if currentBalance<sum then return; // withdrawal canceled
 currentBalance := currentBalance - sum;
 writeAccountBalance(ba,currentBalance);
end;
```

Now, assume that two users simultaneously activate a withdrawal of money from the same account. Therefore, the presented code is executed concurrently by two different processes, A and B. Generally, nothing can be assumed about the speed of execution of these two independent processes. In other words, there is no implied synchronization between these two inherently asynchronous processes. (This is one of the fundamental assumptions in the theory of concurrent computing.)

Let's assume also that the initial account balance was 100, and that processes A and B want to withdraw 80 and 60 units of money, respectively. Consequently, it may happen that the two processes execute their operations in the following order:

| Process A Executes                    | A's<br>CurrentBalance | Process B Executes                    | B's<br>CurrentBalance | Account<br>Balance |
|---------------------------------------|-----------------------|---------------------------------------|-----------------------|--------------------|
| readAccountBalance                    | 100                   |                                       |                       | 100                |
| if currentBalance<sum                 | 100                   |                                       |                       | 100                |
|                                       | 100                   | readAccountBalance                    | 100                   | 100                |
|                                       | 100                   | if currentBalance<sum                 | 100                   | 100                |
| curretBalance:=<br>currentBalance-sum | 20                    |                                       | 100                   | 100                |
| writeAccountBalance                   | 20                    |                                       | 100                   | 20                 |
|                                       |                       | curretBalance:=<br>currentBalance-sum | 40                    | 20                 |
|                                       |                       | writeAccountBalance                   | 40                    | 40                 |

## Chapter 20: Characteristics of Information Systems

---

In the end, the account balance has an inconsistent value because the two processes seem to have completed correctly (both have withdrawn their money), but the account balance is much greater than what is correct.

Obviously, the code of the given procedure should not be executed concurrently by asynchronous processes because such execution may lead to an inconsistent system state. Such a part of code is called a *critical section*. In other words, the processes must be explicitly *synchronized* in a deterministic manner, so that the described and other incorrect orders of execution cannot ever occur.

One strategy of protecting a critical section is *mutual exclusion*. It entails surrounding the critical section by some kind of synchronization primitives that prevent more than one process from entering the section. For example, the protected critical section might look like:

```
procedure cashWithdrawal (ba : BankAccountID; sum : real)
begin
 var currentBalance : real;
 critical_section_begin
 readAccountBalance(ba, currentBalance);
 if currentBalance < sum then ...; // withdrawal canceled,
 // raise an exception
 currentBalance := currentBalance - sum;
 writeAccountBalance(ba, currentBalance);
 critical_section_end;
end;
```

A process may enter the critical section only if there is no other process already executing the section's code. When a process enters the section, it "closes the door behind itself," so that other processes cannot enter the section. When a process comes to a section that is already closed by another process, it can be blocked until the section is abandoned, or canceled in some way (for example, an exception may be raised), depending on the underlying synchronization policy. The concrete synchronization primitives, constructs, and policy available to an application depend on the underlying operating system and programming language.

However, the described approach of protecting the code of the critical section has a drawback: The code cannot be executed by several processes that work on completely different, independent accounts. For example, if process P1 is withdrawing from account A1, another process P2 cannot perform a withdrawal from account A2 simultaneously, even though their concurrent execution cannot corrupt the system's state. This is why information systems often base their concurrency control strategies on the concept of *locks*, which are associated with the object of operation (for example, a data record) and not with the code itself. A process may lock a data entity, which prevents other processes from doing the same. As in mutual exclusion, when a process tries to lock an entity that is already locked, the process is blocked or an exception is raised.

For the given example, the critical section may use a strategy in which bank accounts are locked. Such a solution allows different processes to work concurrently on different accounts, but only one process on the same account:

```
procedure cashWithdrawal (ba : BankAccountID; sum : real)
begin
 var currentBalance : real;
 lock(ba);
```

## Part V: Supplemental

---

```
readAccountBalance(ba,currentBalance);
if currentBalance<sum then begin
 unlock(ba);
 return; // withdrawal canceled
end;
currentBalance := currentBalance - sum;
writeAccountBalance(ba,currentBalance);
unlock(ba);
end;
```

An in-depth explanation of various concepts of concurrency control is beyond the scope of this book. The theory of concurrent computing is complex and very mature, and many textbooks have been devoted to it. Moreover, this theory is applicable not only to information systems, but to many other application domains. This brief overview serves just as an illustration of the main problems of concurrency control that must be treated in information systems.

### **Distribution**

Information systems are most often deployed on several computers that do not have a shared memory, but have their own processing elements, operating memory, and input/output devices (including persistent storage), and that communicate over an interconnecting network. Such systems (where separate processing elements do not share common memory) are described as *distributed*.

Distribution often assumes several levels of system partitioning. First, an information systems is usually accessible from multiple computers that provide an interface between the system and its users. These computers are called *clients*. Sometimes, there may be a virtually unlimited number of clients, as in Web-enabled information systems. Clients enable simultaneous access to the system by many users, accept their input, present the system's output, and communicate with the rest of the system. Clients may also perform some processing and have their local transient or persistent storage.

Application-specific processing may also be performed on separate computers, in order to support parallelism, as already explained. If processing is performed on machines different from the clients, such machines are then called *application servers*. Distribution of processing improves parallelism but also carries communication and synchronization overheads and makes development of software more difficult.

Finally, the persistent storage itself (that is, the database with the DBMS) is usually deployed on separate computers, called *database servers*. Clients or application servers then usually access the database through low-level, generic interfaces, oriented to transaction processing or querying the database. In some cases, a logically unique database may be physically distributed on several database servers. For example, in a wide-area information system, such as an integrated system of an international company, the integral database may be distributed on computers at several geographically dispersed locations, connected through the Internet. The distribution of the data may be influenced by the locality principle, meaning that the data is placed where it is most frequently needed. This may shorten the response time of the system. Another reason for distribution of the database might be increased throughput, fault tolerance, or security, as in military systems.

The concrete way in which the system's hardware is organized and the way in which software is deployed on it determine the *system architecture*. In the past and nowadays, several architectural patterns, as typical and recommended deployment strategies, have been exploited. Entire books could be devoted to this issue, but let's look at a brief overview here.

### Mainframe Architecture

The mainframe architecture (see Figure 20-1) assumes one (or, in special cases, more) central computer with operating memory, persistent storage (disks), and possibly several processors sharing the same operating memory, and many very simple peripheral computers connected to the central computer. The central computer is called the *mainframe*, and the peripheral computers are called *terminals*. The terminals do not run application-specific programs at all, but perform trivial operations, just collecting the input from the users, expressed in simple commands, and presenting the output from the system.

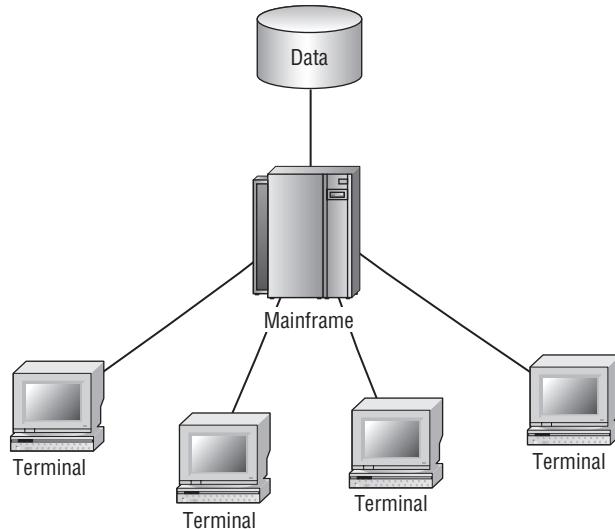


Figure 20-1: The mainframe architectural pattern

Both input and output are transferred between the mainframe and the terminals in raw forms, such as streams of characters or the like. The application itself is executed on the mainframe, where the entire data processing takes place, and where the database is stored, too. The mainframe also handles the raw input from the terminals, performs data conversion and validation, and builds the entire appearance of the output screen presented on the terminals. The DBMS is executed on the mainframe, where the transactions and database querying are performed, too. The mainframe provides concurrent execution of all these tasks for multiple users by means of concurrent processes, which may run in parallel if the mainframe has multiple processors.

This architecture can hardly be regarded as distributed because terminals have a very trivial role in the system. It was popular in the past, but it is mostly considered obsolete nowadays.

### Client/Server Architecture

In the client/server pattern (see Figure 20-2), there is one (or, in special cases, more) *server* computer and many *clients* connected with the server by a network. The server has a massive storage capability (disks), and clients usually do not (or at least, this is not the key factor). Unlike terminals in the mainframe architecture, clients do have significant processing power and operating memory, as well as powerful input/output devices, most notably presentation devices (screens). The database is placed on the server, where the DBMS is executed, too. The clients access the database through query or update requests,

## Part V: Supplemental

---

which are transferred through the network. The server replies to the clients with query results, which are also transported over the network. The entire application-specific processing is performed on the clients. The clients are also responsible for the presentation and input collection. However, the interaction with the user is much more intensive and user-friendly than in mainframe architectures because it is managed by more powerful client machines. It usually includes an intuitive, user-friendly GUI with flexible manipulation.

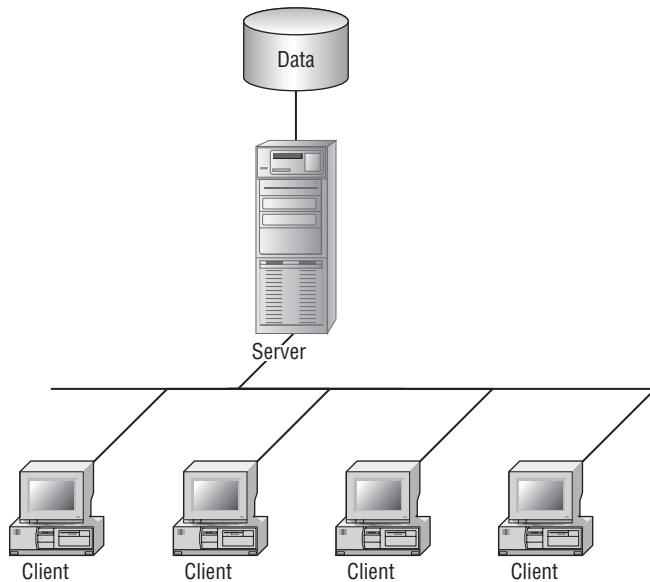


Figure 20-2: The client/server architectural pattern

This architecture is suitable for transaction-intensive applications that require complex user interfaces. Because the communication between clients and servers might be intensive, the network that is used to link these tiers is usually a *local area network (LAN)* (for example, an intranet of a company or department).

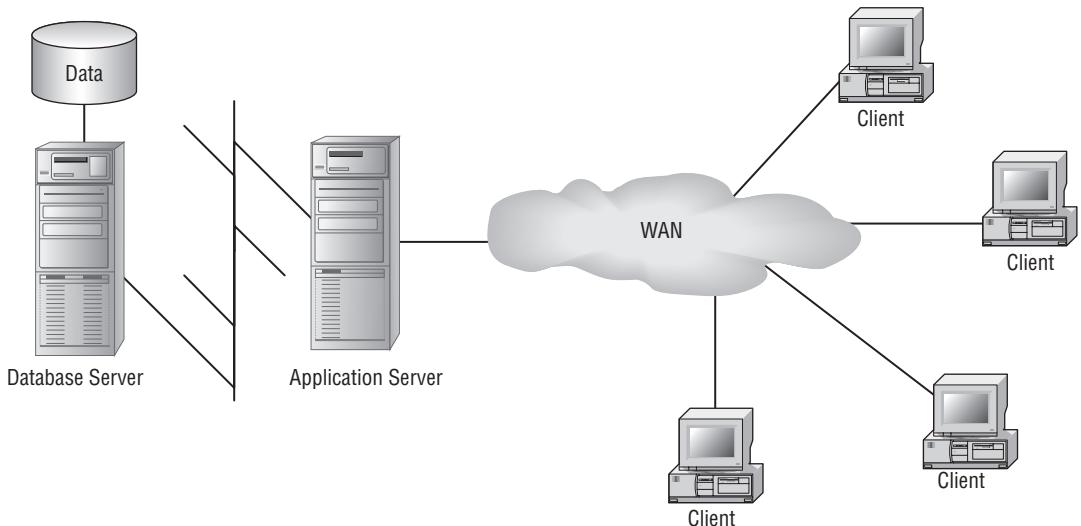
The client/server architecture is more convenient than the mainframe architecture because it allows heterogeneous client machines to access the database server through a unique database access interface. Moreover, because the application-specific processing is distributed on the clients, it has better scalability than mainframes. This is the reason why client/server architectures superseded mainframes.

On the other hand, this architecture is not suitable for wide-area systems, such as Internet-based systems. Because they rely on processing and presentation on the clients, they often suffer from portability problems because the applications are often tied to the client's particular platform (hardware and operating system). Finally, because the presentation and application processing are deployed on the same computer, they are often unnecessarily mixed up. This often leads to a lack of concurrency.

For the given reasons, the client/server architecture has been superseded by architectures that are more scalable and appropriate for wide-area systems. However, client/server architectures are still popular in local-area systems with intensive data flow and with high performance demands.

### **Three-Tier Architecture**

The emergence of Web-enabled information systems has brought the three-tier architectural pattern (see Figure 20-3) as a natural successor of the client/server pattern. In this pattern, there are three tiers: the *database server*, the *application server*, and the *client*.



**Figure 20-3: The three-tier architectural pattern**

The database server has the same configuration and role as in the client/server architecture. It hosts the database, executes the DBMS, and responds to the update and query requests from the upper tier. The upper tier consists of one or more application servers, typically connected to the database server through an intranet. The communication between the application server and database server has the same form and role as in the client/server architecture. However, application servers do not deal with the user interface, but perform the application-specific processing, which is often referred to as the *business logic*.

Clients are computers connected to the application servers through a *wide area network (WAN)*, very often through the Internet. They are responsible for rendering the output transferred from the application servers and for collecting the input. Thus, they generally perform very simple input/output operations and do not have to have extensive processing power. However, unlike terminals in the mainframe architecture, clients usually render GUIs out from a standard textual form provided by the servers. This form is the Hypertext Markup Language (HTML), the standard textual language used on the Web. The clients execute generic programs, specialized for rendering HTML, called *Web browsers*. Additionally, clients are capable of performing input conversion, or even certain data validation. In some cases, clients can perform significant application-specific processing to reduce the communication overhead. Therefore, although clients can perform significant processing, it is generally limited to the user interface and input validation.

Depending on the intensity of the processing on the clients and their roles in the system, this pattern is sub-typed into the so-called *thin client* and *thick client* patterns. Thin clients merely have the role to render HTML on the screen and collect input from the forms. Thick clients perform more complex

## Part V: Supplemental

processing (such as data validation and processing, animation, and so on), executing some application-specific code in the form of, for example, JavaScript or Java applets. Thick clients may even exploit the client's local persistent storage for caching the working set of the data from the server.

On the other hand, application servers are responsible for business logic, as well as for servicing standard Web requests, processing raw input from the clients (usually provided as strings), and preparing output for the clients. The output may be in the form of an entire Web page in HTML, or a partial update to a page in XML, as is the case with the Ajax approach.

This pattern has several important advantages. First, the presentation, application, and database layers are clearly separated, which improves scalability. Second, the presentation is provided in a simple and standard form (for example, HTML), which ensures portability to most heterogeneous clients with small processing capabilities. Finally, because the clients are connected to the servers through the Internet, the system can be easily and widely accessible without the need for the deployment of application-specific software on the clients.

### N-tier Architecture

The N-tier pattern (see Figure 20-4) is a natural generalization of the three-tier approach, whereby the middle tier is divided into two or more separate tiers, each having a certain responsibility. For example, the middle tier can be split into two new tiers, called the *Web server* and the *application server*. Now, the Web server, which is connected to the clients over the Internet, is responsible for receiving Web requests from the clients, analyzing them, and forwarding the requests to the appropriate application server for processing.

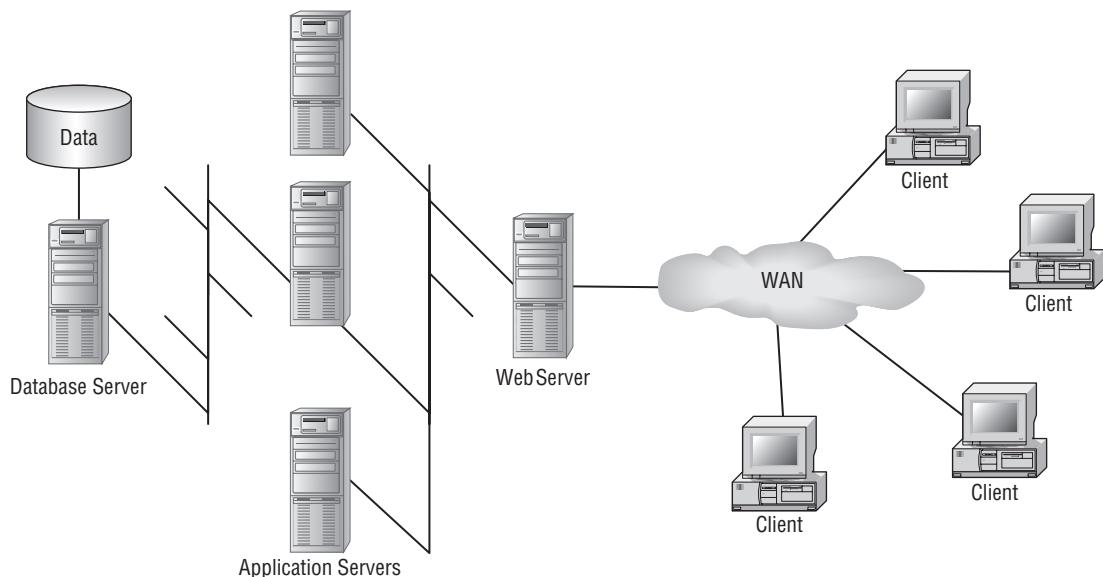


Figure 20-4: The N-tier architectural pattern

There may be several tiers of application servers with different responsibilities (that is, with different specialization of processing). Alternatively, there may be just one application tier with several identical

application servers, which simply exist to enable parallel processing of concurrent requests from the Web server that balances the workload. Application servers reply to the Web server with the information about the presentation of the response on the clients (for example, the succeeding page to be displayed and its contents), but the format of this information is usually not pure HTML. The Web server (or, more generally, one of the tiers) is responsible for generating the HTML page that is returned to the client.

It is obvious that this pattern, as a generalization of the three-tier pattern, has the same fundamental advantages as the latter. Additionally, because it has more clearly separated concerns of the tiers, it has generally better scalability.

### Distributed Architecture

In the distributed approach (see Figure 20-5), the system is distributed completely arbitrarily on the nodes in the network. The distributed application components communicate in a peer-to-peer manner, according to the application-specific needs.

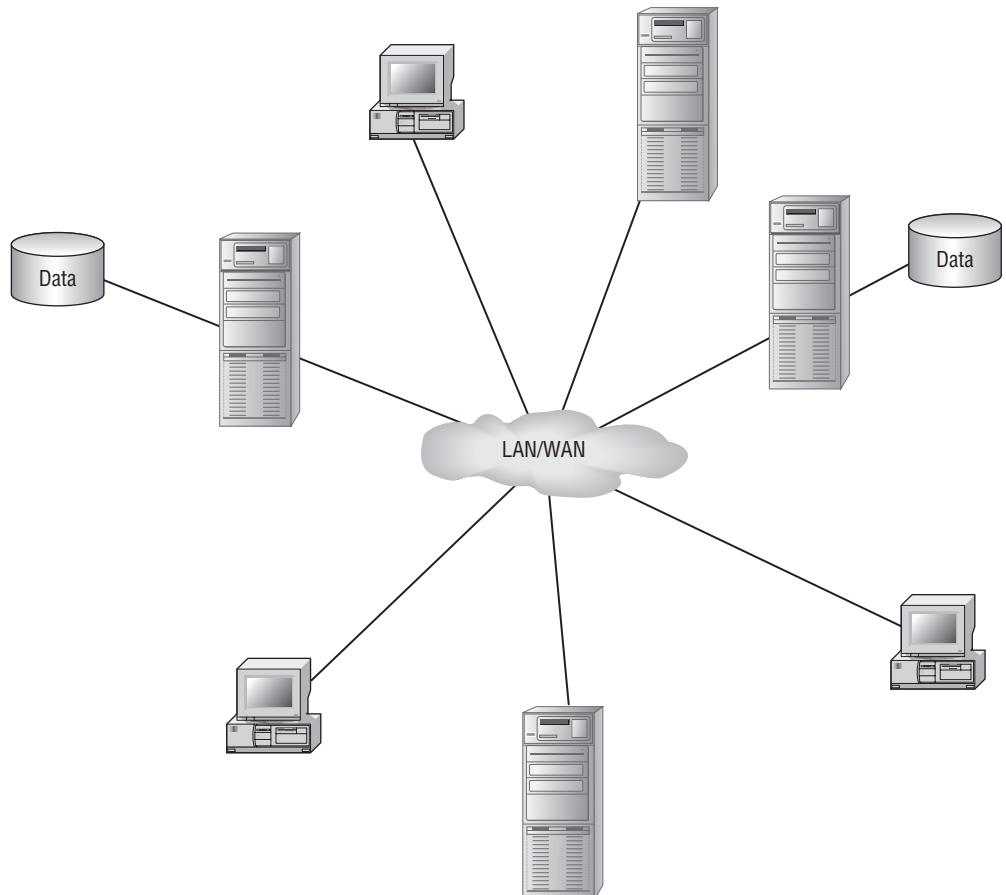


Figure 20-5: The distributed architectural pattern

## Part V: Supplemental

---

In effect, each participant in the peer-to-peer communication can be an entire particular information system of one enterprise, or a separate component of an enterprise information system. It can have an internal architecture according to some of the already described patterns, while the other systems can communicate with it through a machine interface it offers. In that way, the participating systems can communicate in a business-to-business (B2B) manner, exchanging information and providing business services to each other.

*Service-oriented architecture* (SOA) exploits this approach for building globally accessible and distributed information systems that can communicate with each other in a B2B manner. SOA advocates building applications out of software *services* [Wiki]. Services are relatively large, intrinsically unassociated units of functionality (which have no calls to each other embedded in them) offered to their clients. They typically implement functionalities such as filling out an online application form, viewing an online bank account, placing an online product order, or posing a search request for an item in a dictionary.

When a service is accessible to other applications through the Web (that is, over the HTTP protocol), it is referred to as a *Web service*.

Web services can be implemented on an ad-hoc protocol, but also on a standard protocol. Implementation of Web services on a standard protocol ensures their interoperability. One such protocol is SOAP, which uses HTTP as the underlying communication protocol to build up a higher-level, remote procedure call (RPC) paradigm in a platform-independent way. In other words, a client system can call a remote procedure implemented on a server system over the Internet, although the two systems can be deployed on completely different platforms and developed in different programming languages. SOAP uses XML-based messages for the implementation of RPC.

### Selecting a Pattern

The selection of the architectural pattern is a difficult task for a system architect. As described, the decision often must trade off between many issues, such as functionality, portability, availability, performance, scalability, complexity, cost, and many others. An in-depth discussion on these topics falls beyond the scope of this book, but this section emphasizes their importance for the development and exploitation of a system.

### Fault Tolerance

Every real system might break down occasionally. Hardware devices stop working for different reasons. Network connections break down or generate errors, and software components fail because of bugs or exceptions. To be available, an information system must be *fault tolerant*, meaning that it must overcome hardware and software failures in a way that does not endanger its functionality and availability.

The theory and practice of fault tolerance of computing systems is also a mature and complex engineering field, well described in many textbooks. This is why the general techniques for achieving fault tolerance in computer-based systems will not be discussed here. The basic and most often used strategy for achieving fault tolerance is redundancy. For example, a database may be replicated on several disks that work in parallel to prevent data loss on disk failures. Communication lines can be multiplied in order to provide alternative paths on communication breakdowns, and so on.

There is one specific aspect of fault tolerance that is significant for programming information systems. Consider the following simple example. In a banking information system, there is a procedure

## Chapter 20: Characteristics of Information Systems

---

of transferring a certain amount of money from one account to another. The procedure might look like this:

```
procedure moneyTransfer (fromBA, toBA : BankAccountID; sum : real)
begin
 var fromBalance, toBalance : real;
 readAccountBalance(fromBA,fromBalance);
 if fromBalance<sum then return; // transfer canceled
 readAccountBalance(toBA,toBalance);
 fromBalance := fromBalance - sum;
 toBalance := toBalance + sum;
 writeAccountBalance(fromBA,fromBalance);
 writeAccountBalance(toBA,toBalance);
end;
```

Apart from the fact that this code forms a critical section (which we ignore in the interest of simplicity), there is another problem here. Namely, if an execution of the procedure fails when it has already completed the first but not started the second write operation, the state of the system will be corrupted, because the money will be withdrawn from the first account, but not placed into the other. Therefore, this part of code must be protected against failure in a way that prevents the system from passing into an inconsistent state.

The traditionally used solution to this problem is the concept of a *transaction*. A transaction is a sequence of modifications of the system's state that is executed in an atomic manner to guarantee its consistency. The *atomicity* of the sequence simply means that the entire sequence of actions inside a transaction is executed either completely or not at all. More precisely, if the transaction is executed without a failure, the system's state is consistent and will be as determined by the results of all actions executed in the transaction. If the execution of the transaction fails for any reason whatsoever, the system's state remains as it had been before the transaction started. Therefore, the state is consistent again, because it is as if the transaction had not started at all.

For the given example, the code of the procedure should be enclosed in a transaction as follows:

```
procedure moneyTransfer (fromBA, toBA : BankAccountID; sum : real)
begin
 var fromBalance, toBalance : real;
 readAccountBalance(fromBA,fromBalance);
 if fromBalance<sum then return; // transfer canceled
 readAccountBalance(toBA,toBalance);
 fromBalance := fromBalance - sum;
 toBalance := toBalance + sum;
 transaction_begin
 writeAccountBalance(fromBA,fromBalance);
 writeAccountBalance(toBA,toBalance);
 transaction_end;
end;
```

If all actions within a transaction complete successfully, the transaction is *committed*, meaning that the results of the actions are applied to the actual system's data space. Otherwise, the transaction is *rolled back*, meaning that the system's data space is returned to the initial, unmodified state. The implementation mechanisms for transactions differ, and their detailed description is beyond the scope of this book.

### Portability

Information systems often incorporate very different and heterogeneous platforms, whereby a platform encompasses the underlying hardware and operating system. For example, it is not unusual for the client to run one operating system, the application server another, and the database server a third one. Moreover, clients themselves are often very diverse (as in Web-enabled systems, for example). Additionally, the advances of hardware and software technology imply very frequent modifications in platforms. Therefore, software of information systems must be engineered in such a way that ensures its *portability*. Software is portable if it can be migrated to another platform without significant re-engineering efforts and modifications.

Achieving portability is not a simple task. Typically, it is accomplished by splitting the software into layers or parts with crisply defined responsibilities and precise interfaces. If the interfaces are standardized, the problem is easier. For example, the Open Database Connectivity (ODBC) standard defines an interface for accessing relational databases. If an application uses this standard to access the database, it will be easily connected to most commercial DBMSs, because they all support this standard. Similarly, HTML is a standard language recognized by all existing Web browsers. Therefore, if the user interface of an application is oriented toward HTML, it will be portable to literally all platforms.

However, this is not the case with operating systems. Although their concepts and features slowly converge, they still offer very different application programming interfaces (APIs). This fact is one of the most serious obstacles to portability. If an application directly uses non-standard features of an operating system and its API, it will not be portable. Thus, it is wise to place application-specific code upon a layer that hides specific features of operating systems and maps a unified API into the concrete underlying operating system features. Inherently portable programming languages (such as Java and C#), as well as standard libraries, provide such layers, for example. Anyway, portability should always be kept in mind during development of information system software.

#### Section Summary

- ❑ Deployment-related characteristics are associated with technical issues of implementation, deployment, and exploitation of the system:
  - ❑ Diversity and quantity of data
  - ❑ Scalability
  - ❑ Persistence
  - ❑ Concurrency
  - ❑ Distribution
  - ❑ Fault tolerance
  - ❑ Portability

# 21

## Process and Principles of Software Development

This chapter briefly describes the main characteristics of software process models at the project management level. It also explains objectives, techniques, and principles applied during development of not only information systems, but all complex software systems in general.

### Project Management Process Models

In the past, a process pattern called the *waterfall model* was exploited (see Figure 21-1). In that model of software development process, the core activities of software development (that is, analysis, design, implementation, testing, and deployment) follow each other in a strictly sequential manner. That is, the next activity does not begin before the previous one has been entirely completed. Therefore, the waterfall model encompasses a frontal software development. First, the full resource power (personnel, time, and attention) is focused on analysis. When the requirements are captured completely (for the entire system), the design phase begins. Once the system is designed completely, up to all its tiny details, the implementation phase is launched. After the system is implemented, it is tested as a whole. Finally, once the system is tested integrally, it is deployed as a whole.

It turned out that this process model was inefficient and risky because the described frontal approach to software development often led up to a “big bang” of software integration. Namely, complex software is developed in parts, whereby, for example, different people or teams develop different parts. In the waterfall model, these parts are integrated once they are completed. However, this integration carries a very high risk because numerous parts of a complex system rarely fit together at a first shot. They often produce a crash of the system, and, what is worse, this happens very late, when the majority of the system is already implemented. In other words, mistakes are revealed too late, when lots of investments in development have been made. Therefore, fixing mistakes is very costly because it sometimes requires restarting a refinement process from its very beginning, which may also take a long time. Of course, for very simple systems, this model seems to be perfectly feasible, but it is very often unrealistic and doomed to failure, given the complexity of current systems.

## Part V: Supplemental

---

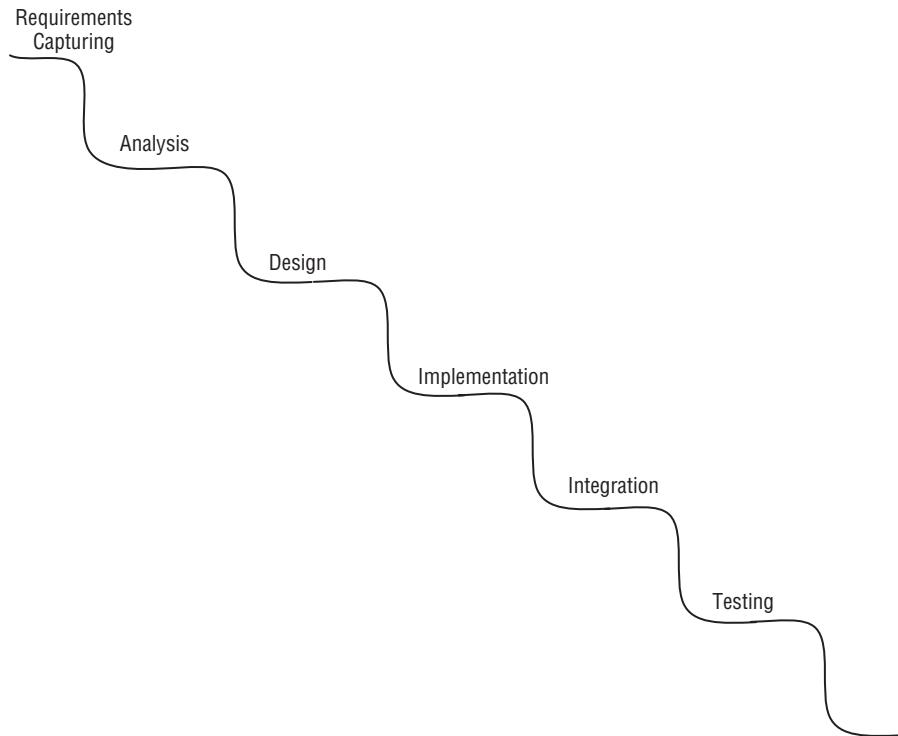


Figure 21-1: The waterfall model of software development process

This is why the waterfall model has been superseded by a newer, more agile and less risky model, called the *spiral model*, shown in Figure 21-2 [Boehm, 1988]. In this model, all core activities of software development (analysis, design, implementation, testing, and deployment) are repeated iteratively, in cycles, whereby each cycle applies all these activities to a small part of the system (that is, to a small increment of its functionality). This way, the system's design is refined and augmented in each cycle, and the system grows to the ultimate solution step by step. This model also allows the project to identify and resolve risks as soon as possible. Additionally, it increases the flexibility of accommodating new requirements or modifications in design decisions.

A contemporary process model, shaped in what is referred to as the *Rational Unified Process* (RUP)<sup>1</sup>, has the following fundamental characteristics [Booch, 1999]:

- The process is *iterative and incremental*. As already stated, a system is not developed frontally and sequentially, but in cycles, whereby each succeeding cycle starts from the product of the previous cycle. In each cycle, the understanding of the system is refined and the product is enhanced with a new set of implemented features. This way, the risk of system integration is reduced, and the sources of risk can be identified and resolved sooner rather than later.

---

<sup>1</sup>RUP was developed by Rational Software Corporation, now part of IBM.

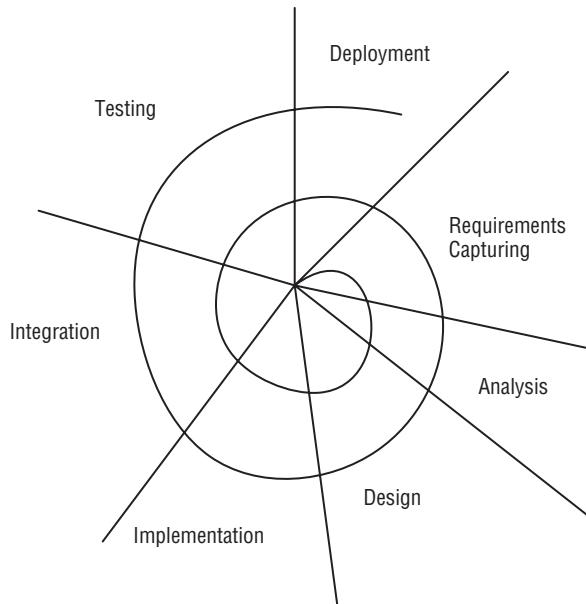


Figure 21-2: The spiral model of software development process

- The development under the process is *architecture-centric*, because the key element of a successful software system is a stable, coherent, clear, and flexible architecture. *Software architecture* is the way in which logical and/or physical components of a software system are organized (that is, how the responsibilities are divided between them and how they collaborate in order to accomplish a broader purpose). The process is oriented toward early development of such architecture, which facilitates software reuse and reduces rework. Actually, the ultimate prerequisite for the iterative and incremental software development to be successful is that the underlying architecture is stable enough to allow you to build complex software — you can build a huge building on top of a small foundation and in small increments only if the foundation is stable enough.
- The development under the process is *use case-driven*. A use case represents a description of a functional requirement of the system. The process is use case–driven because use cases are what drives the process of development in each development cycle. That is, each development cycle focuses on a small set of use cases selected for realization in that cycle. The development focuses on analysis and implementation of the selected use cases, and use cases are also the subject of testing. The notions of use cases and scenarios of their realization are used to direct the process flow from requirements specification to testing, thus providing traceable threads through software development.
- The process is directed to *quality assurance and risk management*. This means that eliminating risks early on and achieving a high-quality product are key criterions for selecting use cases to be captured in a development cycle. The major distinguishing feature of the spiral model is that it creates a risk-driven approach to the software process rather than primarily document-driven or code-driven processes.

### Section Summary

- ❑ The modern understanding of process assumes that it is iterative and incremental, architecture-centric, use case-driven, and oriented to quality assurance and risk management.

# Objectives and Principles

While developing software by creating models and following processes, engineers should always strive to achieve certain software engineering objectives. The objectives stay as ultimate goals to which software engineering has been constantly moving closer, but that do not seem to be fully reachable. Regardless of the paradigm, language, or process model used during development, there are some objectives that engineers should always bear in mind.

To meet these objectives, engineers apply some general software engineering techniques and obey general software engineering principles, independently of the used modeling language, level of abstraction, or development process.

## Objectives

Two important objectives of software development are *flexibility* and *reusability*.

### Flexibility

To fulfill all desired characteristics of complex software systems (such as portability, maintainability, usability, evolution, and so on) — in other words, to face up to permanent demands for modifications — a software system must, in fact, be flexible, meaning that necessary modifications can be incorporated into the system without significant efforts. For example, portability means that an existing system can be ported to another platform, ideally without any (or with just small) adaptations. Similarly, a system is maintainable if it can accommodate error corrections, requirements modifications, or requirements enhancement. Therefore, flexibility is one of the most important engineering properties of a good software system.

Maintenance (as a set of activities aimed at correcting errors or responding to enhanced or modified requirements) implies enhancement or modification of software. However, it is usually less risky to apply extensions than modifications in the existing software because modifications sometimes cause defects in the existing functionality, or even ruin the existing software architecture by a domino effect (when fixing one error causes a set of others to appear). Consequently, it is always an intention to respond to new requirements by extending and not modifying software. Nevertheless, it may be even more beneficial if the response to modified requirements (that is, changes in the initially posed requirements) can be an extension of the software as well, and not its modification. The OO paradigm supports this principle by its notions of inheritance and polymorphism, as explained in this book.

### Reusability

One of the key factors for improving software development productivity is the possibility to reuse development artifacts. It is evident that a solution to a problem or an implementation of a requirement can be

obtained much more easily if it has already been available as a reusable artifact. Depending on the kind of the reusable idea, solution, or artifact, software reusability can be achieved at a number of different levels:

- ❑ **Code pattern-level reusability** — Programmers often reuse some parts of code that represent typical usage of a programming-language construct or an API concept. Usually, the programmer copies an excerpt of available code and customizes it to the particular usage. Such parts of reusable code excerpts are called *code patterns* (or *idioms*). For example, a typical iteration through a composite structure available in a library, or a counter loop in C and other languages that inherited the same approach, can be called patterns at the code level:

```
for (int i=0; i<n; i++) ...
```

This is really software reuse, although at the lowest level of granularity and size. Macros and their parameterized replacement provide a good tool for capturing idiom-level reusability.

- ❑ **Template-level reusability** — Some programming languages (such as C++, C#, or newer versions of Java, for example) support automatic customization of larger parts of code. Typically, the implementation of composite structures of elements, available in libraries, does not depend on the type of the elements itself. In that case, the implementation can be designed generically, as a *template*, whereby the code relies on an abstract type of the element as its parameter. The programmer then instantiates the template at different places, by parameterizing the code with concrete element types. The compiler may generate the code of the concrete structure, replacing the formal parameter of the template with the actual type supplied, as if the programmer did it manually. Instantiation of the same template at different places is another example of code reuse.
- ❑ **Library-level reusability** — It has been always a common practice to reuse some off-the-shelf parts of code, such as procedures, functions, or classes from *libraries* of such parts. Library elements are parts of code or models that perform certain tasks, and are ready to reuse without any modification or customization.
- ❑ **Algorithm-level reusability** — It is sometimes impossible to reuse a part of code without customization, or with just small modifications. However, the solution that already has been applied to a similar problem can be reused. For example, an algorithm for solving a certain kind of problem can be reused, although its implementation must be programmed from scratch.
- ❑ **Design pattern-level reusability** — Similar to the algorithm-level reusability, this is also a kind of solution-based reuse approach. *Design patterns* are elegant and efficient solutions to commonly appearing problems in a certain domain, expressed not through excerpts of code, algorithms, or templates, but in terms of more complex collaborations of structural and behavioral elements [Gamma, 1995]. The structural elements may be expressed in terms of *roles*, as abstract things that will be replaced at the time of pattern application by concrete elements of the software that play these roles. Therefore, similar to algorithms, patterns may not be completely ready-to-use parts, but they are ideas or directions to the solutions of common problems that should be customized for each particular application.
- ❑ **Framework-level reusability** — For certain application domains, or for some categories of applications that share some common characteristics, *frameworks* provide another level of reusability. A framework is a skeletal structure of a program that must be elaborated on to build a complete application. Frameworks often comprise large sets of classes, functions, and other elements that can be used or extended. They often encompass mechanisms of interaction that provide some implemented and reusable functionality. For example, there are a lot of available frameworks for interactive applications with GUIs, Web applications, and so on.

## Part V: Supplemental

---

- ❑ **Component-level reusability** — *Components* are ready-to-use, encapsulated, physical software artifacts that can be instantiated, plugged in with other components, or simply embedded into specific applications to provide certain functionality. Components are used as black boxes, meaning that their internal mechanisms and implementation are not important for their use. The only thing the user of a component should know is its interfaces, which define its services.
- ❑ **Architecture-level reusability** — Another example of solution-based reuse is architecture reuse. It assumes reusing the same idea of how to organize different software systems in the same way by applying the same software architecture.

There may be other unclassified ways to reuse known ideas, artifacts, or solutions to similar problems, existing or upcoming.

At the modeling level, design patterns play a significant role in reusability. They represent high-level solutions that may be immediately used or easily customized for a particular application. This is why patterns play a significant role in some solutions presented in this book.

## Principles

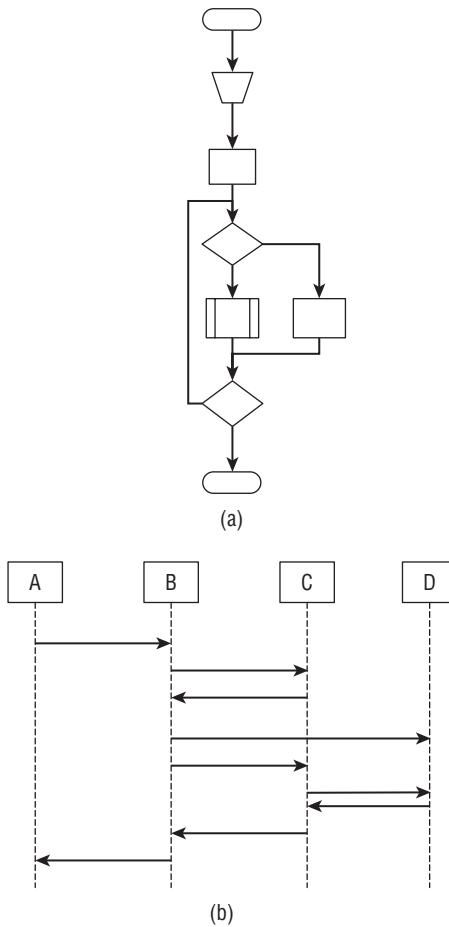
In order to meet the presented objectives, engineers apply some general software engineering techniques and obey general software engineering principles, independently of the used modeling language, level of abstraction, or development process. Following are some of the most important principles.

## Decomposition

One fundamental and powerful weapon available to engineers in the battle against the inherent complexity of software systems (in addition to abstraction) is *decomposition*. In order to cope with the complexity of a system, engineers break the system apart. That is, they break its structure and/or its functionality into parts and attack these parts separately, applying the “divide and conquer” principle. To achieve this, engineers apply the available techniques of decomposition, two of which are historically most important (see Figure 21-3):

- ❑ **Procedural decomposition (also referred to as algorithmic decomposition, Figure 21-3a)** — This kind of decomposition assumes that a certain functionality of the system is divided into smaller parts, or steps. Namely, a functional requirement (that is, a requirement that the system must accomplish a certain task) is broken up into steps. For example, the thoughts in the procedural decomposition might go like this: “In order to accomplish a task *A*, the system must first do step *A*1, then step *A*2, then — if a certain condition is satisfied — step *A*3, and so on.” The decomposition is hierarchical, starting from the highest level of abstraction (the top-level functionality) and going downward iteratively. In each iteration, a step at a higher level of abstraction is refined into smaller steps at a lower level of abstraction, until the model reaches the level of implementation (for example, a statement or procedure in the target programming language). While applying this kind of decomposition, engineers often discover common parts of behavior that are used at several places and localize them into procedures, possibly parameterized to support invocations from different contexts.
- ❑ **Object-oriented decomposition (see Figure 21-3b)** — Object-oriented decomposition is a set of activities, principles, and techniques that lead up to a system decomposed across several dimensions, both structural and behavioral. It is difficult to explain the entirety of OO decomposition in a couple of sentences, and many books on OO technology, including this one, discuss OO

decomposition from different viewpoints. Let's just say that by applying object-oriented decomposition, a system is ultimately broken up into *objects*, as entities that interoperate in complex manners to provide the system's functionality. The interaction is generally fulfilled by *scenarios* of interchanging messages between objects. However, object-oriented decomposition includes many other things, too.



**Figure 21-3: Decomposition. (a) Procedural (algorithmic) decomposition. (b) Object-oriented decomposition.**

It should be emphasized that the object-oriented decomposition is not a revolution or replacement of the procedural decomposition. Instead, it is an evolution and enhancement of the latter. Actually, the procedural decomposition is also one of the techniques incorporated in the object-oriented decomposition. However, the procedural decomposition is applied non-exclusively, and not predominantly, but it is used in different contexts, at a lower level of abstraction. It is basically used in designing scenarios of object interaction, or when implementing objects' services.

### ***Localization of Design Decisions***

The end artifacts of design decisions should be localized and not scattered across wide parts of software. This is a general principle that can be applied at very different design levels. There are many examples, but let's look at just a few trivial and well-known ones.

For example, the implementation of an algorithm should be localized and encapsulated into a procedure, which is then invoked from different places, and not in-lined within all these invocation places. Similarly, the decision that the dimension of an array is, for example, 100 should be localized in a definition of a symbolic constant or a configuration parameter that will be used in the rest of the program, instead of scattered by using the literal 100 in all iterations through the array.

The reason is very simple. Flexibility of the system can be achieved only if this principle is strictly obeyed. In other words, modifications of design decisions (which are quite often in practice) can be accommodated easily only if the manifestations of these decisions are localized. For the given examples, if this principle were violated, the modifications would be very difficult. If the decision about the implementation of the mentioned algorithm were changed, it would be difficult to modify the program at all invocation places. Similarly, if the decision about the dimension of the array were modified, it would be rather difficult to search the program for all occurrences of the literal 100 and replace only those that refer to the same entity and not those that refer to completely different concepts that accidentally coincide with the value 100 (such as dimensions of other, independent arrays).

Hunt and Thomas [Hunt, 2004] call this the DRY principle ("Don't Repeat Yourself"). Every piece of knowledge must have a single, unambiguous, and authoritative representation within the system. Although an implementation of the system may end up with several physical copies of the same piece of information, only one of these must be an authoritative source — that is, the one that should be manually changed when the design decision is changed, and the others should be obtained automatically. Macros and header files in C and C++ represent a simple example of this principle.

### ***Loose Coupling***

The positive effect of decomposition is that separate, smaller parts are easier to cope with than the entire complex system. However, decomposition into small parts carries also a risk of their integration. Namely, it often happens that perfectly implemented and tested parts (usually developed by different persons or teams) produce a malfunctioning system when integrated.

From the rich experience of software engineering, it has been concluded that the main problem of integration of software parts lays not in the parts themselves, but in their interconnections. This is why interconnections of parts require special attention during design. It is also well-known that loosely coupled parts of software (where dependencies between parts are crisply defined) have a greater chance for successful integration. Additionally, loose interconnections can be controlled better and lead to better flexibility, because parts are more independent of each other.

This is why loose coupling is one of the main principles of good decomposition. Put another way, this principle says that parts should "know" as little as possible about each other because they remain more independent of modifications. As Hunt and Thomas metaphorically state [Hunt, 2004], a piece of software should be shy: like a 4-year-old hiding behind a mother's skirt, it should not reveal too much of itself, and should not be too nosy into others' affairs.

### Section Summary

- ❑ The pragmatic objectives of software engineering are:
  - ❑ Flexibility of software
  - ❑ Reusability of software, which can exist at different levels (such as code patterns, templates, libraries, algorithms, design patterns, frameworks, components, architectures, and others)
- ❑ In order to meet these objectives, engineers apply general software engineering techniques and obey general software engineering principles:
  - ❑ Decomposition, most notably the procedural and the object-oriented decomposition
  - ❑ Localization of design decisions
  - ❑ Loose coupling of software parts



# 22

## The Relational Paradigm

This chapter describes the main concepts and principles of the relational paradigm, along with its mathematical foundation and the related technology, including SQL, DBMS support, and development tools.

### Introduction

In the history of databases and information systems, by far the most influential breakthrough was the emergence of the *relational paradigm*, or the so-called *relational database model* [Codd, 1970]. (According to the definitions provided in Chapter 1, the term “model” refers to a representation of a concrete system or application, while the term “paradigm” encompasses a set of concepts used for modeling. This discussion addresses the latter, and thus the term “relational model” will be used only if it refers to a concrete application, and not to the entire paradigm.) Before its appearance, engineering of information systems and databases used some other methods for data representation, such as the *hierarchical* or the *network* paradigms. However, the relational paradigm has proven to be the most practical. Today, the vast majority of information systems rely on underlying relational database models. This is because the relational paradigm is easy to understand and use, the support of database management systems (DBMSs) is stable and standardized, and the entire paradigm has been approved as suitable for successful systems deployment. However, the relational paradigm also has some drawbacks that are discussed in Chapter 2.

To illustrate this, let’s use a simple yet descriptive example of a small information system for an imaginary school named Easylearn. Although a complete information system would also encompass some domain-specific functionality and applications, this examination concentrates only on the structure of the system’s data (that is, the representation of its conceptual model in the relational paradigm), as well as on some simple operations on it.

The Easylearn school offers different courses to its students, such as Cooking for Dummies, Computers for Everybody, and so on. For each course in the school, the information system should keep the following information:

- Name of the course

## Part V: Supplemental

---

- Duration of the course expressed in the number of classes
- Price of the course
- Level of the course (which can be basic, intermediate, or advanced)
- An indicator that tells whether the course is offered on behalf of an associate school with which Easylearn has a cooperation agreement

Additionally, the school stores information about all persons that either attend courses or work for the school (teaching or administration). For each person, the system should provide the following information:

- Name
- Postal address
- Phone number
- E-mail address
- Credit card number

Persons in the Easylearn school also include teachers. In other words, some of the persons are those who teach courses. Moreover, teachers are also a kind of person because teachers also have their names, e-mail addresses, and all other properties of persons, and they can be students of other courses in the school as well. However, teachers are a special kind of person for which the system should also record their Web site addresses (so that the students can be informed about their teachers), titles, and reception hours (so that the students can contact them personally).

### Section Summary

- In the history of databases and information systems, by far the most influential breakthrough was the emergence of the *relational paradigm*.

## Fundamental Concepts

The central concept in the relational paradigm is a *table*. A table is a two-dimensional structure that fully fits into the intuitive human understanding of the concept of a table. A table is a set of *rows*, which are also called *records* or *tuples*. The rows are not ordered (that is, there is no implicitly imposed ordering of the rows). Each row has an ordered set (a tuple) of named values. The number of values, and their names, order, and types are the same for all rows. The values are scalar and non-composite, meaning that a value in a tuple cannot be a vector or another composed record. In other words, the types of the values are scalar types, such as integers, strings, real numbers, or even pictures, audio, or video recordings. Therefore, the values in the rows may be viewed as organized into homogeneous *columns* of the table, where the name of the column can be used to refer to the corresponding values in the tuples. Columns are sometimes also called *fields*, *attributes*, or *properties*. Because columns are named and can be referred to by their names (which must be unique in the scope of a table), their ordering also becomes irrelevant.

Consequently, a table is defined by its columns — that is, by the names of the columns and the types of the values in these columns. This definition of the table is regarded as static, specified at design time, and generally not modified at run-time. However, relational DBMSs support addition and removal of table columns, or modification of the definitions of the columns at run-time, too. Of course, rows (that is, records) can be dynamically added and removed, and values in them can be modified at run-time, during the exploitation of the system.

For the given example of the Easylearn school system, the concepts of course and person can be modeled by the corresponding tables `Course` and `Person`. The following is the `Course` table.

| <b>name</b>              | <b>duration</b> | <b>price</b> | <b>level</b> | <b>outsourced</b> |
|--------------------------|-----------------|--------------|--------------|-------------------|
| Car Mechanics for Ladies | 15              | 45.0         | Basic        | false             |
| Cooking for Dummies      | 20              | 99.9         | Basic        | false             |
| Computers for Everybody  | 50              | 450.0        | Intermediate | true              |
| ...                      | ...             | ...          | ...          | ...               |

The following is the `Person` table.

| <b>name</b>    | <b>postal</b>                    | <b>phone</b> | <b>email</b>        | <b>cardnum</b> |
|----------------|----------------------------------|--------------|---------------------|----------------|
| John Smith     | 45 5th Avenue, New York City, NY | 432002       | john@smith.com      | 455523         |
| Steven Walsh   | 22 45th St., Denver, CO          | 234456       | swalsh@computer.com | 333467         |
| Alice Thompson | 715 23th St., Washington, DC     | 255667       | alice@univ.edu      | 210054         |
| ...            | ...                              | ...          | ...                 | ...            |

A row must be unique within its table, meaning that there cannot be two or more rows in a table with all equal values. This ensures that a single row can be uniquely identified by a tuple of values within the operations that need to modify it because addressing rows by values is one of the key ideas of the relational paradigm. Of course, it is not necessary that each row have unique values in all its fields. For a given example, it may likely happen that several persons have the same names or postal addresses. However, there must exist at least one subset of columns that uniquely identifies each row (that is, a subset of columns on which every row has a unique tuple of values). Such a subset of columns is called a *primary key*. For the given example, the name of the course may be a primary key (assuming it is unique), or the e-mail of a person (assuming it is unique) may be a primary key, or the name of a person and that person's postal address may form a composite, multiple-field primary key, assuming that there cannot be two persons with the same name living at the same address.

Obviously, a table can have several primary keys. At the very least, the entire set of columns of a table is its primary key because every record is unique within the table by default. However, a set of columns that form a key may not be *minimal* (or *nonredundant*), meaning that there may exist its true subset that also forms a primary key. It is easy to see that every superset of a primary key is also a primary key. For

## Part V: Supplemental

---

example, the set of fields {name, duration} of the table Course is not a minimal key, because its subset {name} is also a key. A *minimal* (or *nonredundant*) *primary key* is a primary key any true subset of which does not form another primary key (that is, has no fields that are superfluous in uniquely identifying each row). When a relational model is being specified, usually one of possibly several candidate primary keys of a table is selected to represent *the* primary key of that table in the model.

The concept of a primary key is important when tables should be created for the information that links records from several tables representing concepts from the problem domain. For the ongoing example, the Easylearn system should store the information concerning which persons attend which courses. An obvious approach may be to create another table with records representing linkages between courses and persons. To identify the linkages, the records in this intermediate table should use the minimal information that identifies the records from the tables Course and Person — that is, their primary keys. Such a set of fields in a table that is not a primary key of that table, but refers to a primary key of another table to identify its records, is called a *foreign key*. For the given example, if you assume that the fields name and postal form the composite primary key of the table Person, following is the intermediate table named PersonAttendCourse, which stores the information about courses attended by persons.

| personName     | personPostal                     | courseName               |
|----------------|----------------------------------|--------------------------|
| John Smith     | 45 5th Avenue, New York City, NY | Car Mechanics for Ladies |
| John Smith     | 45 5th Avenue, New York City, NY | Cooking for Dummies      |
| John Smith     | 45 5th Avenue, New York City, NY | Computers for Everybody  |
| Steven Walsh   | 22 45th St., Denver, CO          | Car Mechanics for Ladies |
| Steven Walsh   | 22 45th St., Denver, CO          | Computers for Everybody  |
| Alice Thompson | 715 23th St., Washington, DC     | Cooking for Dummies      |
| ...            | ...                              | ...                      |

In this table, the set of columns {personName, personPostal} represents a foreign key, referring to the records of the table Person, and the field courseName represents a foreign key referring to the table Course.

In the previous relational model, there is an obvious downside to the table PersonAttendCourse. Namely, there is a large redundancy of information — several rows carry the same information about the address of a person because the person attends several courses. For example, three records in the PersonAttendCourse table and one in the Person table carry the same name and address of John Smith, just to show that John attends three courses. This occurs because of the choice to have the set of fields {name, postal} of the Person table as its primary key. Apart from wasting storage space with such redundant data, this database design may cause significant performance penalties because an unnecessarily large amount of data must be compared when a query, such as to find all courses attended by the given person, is executed. Additionally, this design causes difficulties in updating values — for example, when John Smith changes his postal address.

Consequently, multiple-field primary keys are often inconvenient because of the data redundancy, consistency maintenance, and performance penalties issues. On the other hand, selecting a single field of a table to be a primary key is also often problematic. First, it is not always possible because there is no field that uniquely identifies records. Second, even if such a field exists, it is questionable whether the assumption of the uniqueness of the field values will hold forever. For example, you may have concluded for the moment that the course name represents a primary key in the Course table, but in the near future, the school may decide to introduce several courses with the same name, but at different levels (that is, introductory, intermediate, advanced). In that case, the model is no longer correct, and requires a significant redesign to meet the new requirements. In short, selection of the primary key may be based on a domain-specific assumption, and not *a priori* given as such.

This is why modelers often introduce a separate field with artificially generated unique values, completely unrelated to the problem domain and the information from it. The values for this field, often referred to as the *technical ID*, are generated automatically by the DBMS on every insertion of a new record into the table, or by the application program itself. Such an ID is then used as the foreign key in other tables. For the ongoing example, the following shows what the Course table might look like.

| <b>ID</b> | <b>name</b>              | <b>duration</b> | <b>price</b> | <b>level</b> | <b>outsideSource</b> |
|-----------|--------------------------|-----------------|--------------|--------------|----------------------|
| 1         | Car Mechanics for Ladies | 15              | 45.0         | Basic        | false                |
| 2         | Cooking for Dummies      | 20              | 99.9         | Basic        | false                |
| 3         | Computers for Everybody  | 50              | 450.0        | Intermediate | true                 |
| ...       | ...                      | ...             | ...          | ...          | ...                  |

The following shows what the Person table might look like.

| <b>ID</b> | <b>name</b>    | <b>postal</b>                       | <b>phone</b> | <b>email</b>        | <b>cardnum</b> |
|-----------|----------------|-------------------------------------|--------------|---------------------|----------------|
| 1         | John Smith     | 45 5th Avenue, New York<br>City, NY | 432002       | john@smith.com      | 455523         |
| 2         | Steven Walsh   | 22 45th St., Denver, CO             | 234456       | swalsh@computer.com | 333467         |
| 3         | Alice Thompson | 715 23th St., Washington,<br>DC     | 255667       | alice@univ.edu      | 210054         |
| ...       | ...            | ...                                 | ...          | ...                 | ...            |

The following shows what the PersonAttendCourse table might look like.

## Part V: Supplemental

| personID | courseID |
|----------|----------|
| 1        | 1        |
| 1        | 2        |
| 1        | 3        |
| 2        | 1        |
| 2        | 3        |
| 3        | 2        |
| ...      | ...      |

Now, the field `personID` is a foreign key referring to the `Person` table, and `courseID` is a foreign key referring to the `Course` table. Note that now the values of all other fields, except the ID fields, in the tables `Course` and `Person` do not need to have unique values at all. This avoids the necessity that the DBMS control the required uniqueness of the values of the primary key because the values of the ID fields are generated automatically in a sequence, and are, thus, inherently unique.

The described relational model stores the information of relationships between persons and courses in an indirect way. For example, in order to get the names of the courses attended by John Smith, you must *join* the `Person`, `PersonAttendCourse`, and `Course` tables. To “join” means to combine all the records from these three tables so that a new, huge virtual table of records is obtained, then select only those records of this virtual table that have equal values of the corresponding primary keys and foreign keys, and present only the values of the fields `Person.name` (the value of the `name` field from the `Person` table) and `Course.name`. As the result of such a join, you get the following virtual table:

| Person.name    | Course.name              |
|----------------|--------------------------|
| John Smith     | Car Mechanics for Ladies |
| John Smith     | Cooking for Dummies      |
| John Smith     | Computers for Everybody  |
| Steven Walsh   | Car Mechanics for Ladies |
| Steven Walsh   | Computers for Everybody  |
| Alice Thompson | Cooking for Dummies      |
| ...            | ...                      |

### Section Summary

- ❑ A *table* is a set of *rows* (or *records*, *tuples*), each row being an ordered set (a *tuple*) of named values, called *fields*. The values of the same field in all records have the same type.

- A subset of fields whose values uniquely identify records within a table is called a *primary key*.
- A minimal primary key is a primary key of which there is no true subset of fields that also form a primary key. One primary key is selected to uniquely identify rows in a table.
- A subset of fields whose values refer to the primary key of another table is called a *foreign key*.

## Mathematical Foundation

The relational paradigm has a sound (yet simple) mathematical foundation. All the described fundamental concepts have their formal counterparts in the mathematical set theory and relational algebra. Let's take a look at just a few definitions of these concepts, as an illustration. The definitions presented here should not be taken as a complete formal foundation or reference of the relational paradigm, however.

Let  $V_i, i = 1, \dots, v$  be the sets of possible scalar values in the system — that is, the types supported in the modeling environment. For example,  $V_1$  could be the set of integers,  $V_2$  the set of strings, and so on. These sets are also called the *domains* of a relational model. A table  $T$  then is a subset of the Cartesian product (denoted with  $\times$ ) of some group of sets  $V_i$ :

$$T = \{ < v_1, v_2, \dots, v_n > \mid < v_1, v_2, \dots, v_n > \in V_{i1} \times V_{i2} \times \dots \times V_{in} \}$$

Thus, a table is a set of ordered tuples (enclosed in  $<$  and  $>$ ) of a fixed length  $n$ , whereby the  $k$ th component of a tuple is a value from a set  $V_{ik}$ . This is exactly the mathematical concept of *relation*, from where the relational paradigm got its name.

If  $t = < v_1, v_2, \dots, v_n >$  is a tuple from a relation  $T$ , and  $i$  is an index ( $1 \leq i \leq n$ ), let  $v_i$  be denoted with  $v_i = t[i]$ . Without loss of generality, instead of referring to the tuple components by their indices, one can also refer to them by their names that are unique within the scope of a relation, or more precisely, by the notation *RelationName.fieldName*. This allows you to have the same domain several times in the same relation without ambiguity in referring to each component. In that case, the ordering of the components in the tuple and the domains in the Cartesian product becomes irrelevant. Such relations are sometimes called *relationships*, while the name of a component defines the *role* of the domain at that place in the relationship.

The concept of a primary key can be defined as follows. A primary key is a subset of indices  $\{i_1, i_2, \dots, i_k\}$  such that for each  $t_1$  and  $t_2$  from  $T$  holds:

$$(t_1[i_1] = t_2[i_1] \wedge t_1[i_2] = t_2[i_2] \wedge \dots \wedge t_1[i_k] = t_2[i_k]) \Rightarrow t_1 = t_2.$$

This formula expresses the fact that, if two primary keys are equal, then they must belong to the same row. This way, the values at indices  $\{i_1, i_2, \dots, i_k\}$  uniquely identify the tuples within a relation.

For retrieving information from relations, the relational paradigm defines several basic mathematical operations upon relations: *product*, *selection*, *projection*, and *join*.

## Part V: Supplemental

---

The *product*  $\times$  of two relations,  $T_1$  and  $T_2$ , is the set of tuples that have the components from the two relations:

$$\times(T_1, T_2) = \{< u_1, u_2, \dots, u_{n1}, v_1, v_2, \dots, v_{n2} > \mid < u_1, u_2, \dots, u_{n1} > \in T_1 \wedge < v_1, v_2, \dots, v_{n2} > \in T_2\}$$

This operation is used to create another huge virtual table that combines (in a form of a Cartesian product) all tuples from the operand tables, as a step before applying some other operations. For example, this is the first step in joining two tables, as described previously.

A *selection*  $\sigma$  of a relation  $T$  is the subset of the tuples from  $T$  that satisfy a certain logical condition (also referred to as a *predicate or assertion*)  $P$ :

$$\sigma(T, P) = \{< v_1, v_2, \dots, v_n > \mid < v_1, v_2, \dots, v_n > \in T \wedge P(< v_1, v_2, \dots, v_n >)\}.$$

This operation reduces the set of tuples from one table (possibly a virtual table obtained by a product) to the subset of only those that satisfy a certain logical condition.

A *projection*  $\pi$  of a relation  $T$  on a set of indices  $\{i_1, i_2, \dots, i_k\}$  (without loss of generality, it may be assumed that  $i_1 \leq i_2 \leq \dots \leq i_k$ ) is the set of the tuples from  $T$  that comprise only the components  $\{i_1, i_2, \dots, i_k\}$  from the tuples of  $T$ :

$$\pi(T, \{i_1, i_2, \dots, i_k\}) = \{< t[i_1], t[i_2], \dots, t[i_k] > \mid t \in T\}.$$

This operation is used to filter only those columns of a table (possibly a virtual one obtained by other operations, such as products and selections) to only those that are of further interest.

One important property of all these operations is that their results are also sets of tuples (that is, relations). This is why these operations can be combined and nested (that is, the result of one operation can be an operand of another). For that reason, the term “relation” may refer to both a tabular structure that actually stores the data in the database, as well as to a derived “table” that is the result of a relational expression.

For the given example of the query to “find all courses attended by John Smith,” you can write a mathematical expression in terms of the described operations:

```
π (
 σ (
 × (× (Person, PersonAttendCourse), Course)
 ,
 (Person.ID = PersonAttendCourse.personID) ∧
 (Course.ID = PersonAttendCourse.courseID)
)
,
 {Person.name, Course.name}
)
```

As you can see, although completely formal, this notation is not readable enough to be practically useful. This is why more understandable languages have been designed, as you will see later.

Very often, products of several relations are filtered by subsequent selections of only those tuples that meet the condition that the corresponding foreign and primary keys be equal. This was exactly the case with the previous example where the product of three tables was filtered with the criterion of equality of the ID fields. This is a general need when the information is to be retrieved from tables that conceptually link data from the problem domain. For that purpose, the relational formalism defines the concept of a *join*, which is shorthand for a product combined with a selection with the equality of the corresponding foreign and primary keys. Because this concept is intuitive enough, and because the purpose of this section is not to present a fully formal and complete mathematical definition of the relational model, its definition is omitted here.

### Section Summary

- The fundamental concepts of the relational paradigm rely on a formal foundation — mathematical set calculus and relational algebra.
- The concept of a table corresponds to the mathematical concept of a relation. This is how the paradigm got its name.

## Actions Upon Structure

Thus far, the description of the relational paradigm has examined its *structural* components only — that is, how information is logically organized in a system. This is a rather static view to the system. On the other hand, systems are dynamic, and the paradigm should also define *behavioral* elements, that is, operations that can be performed on the data structure.

It should be emphasized how important this issue is. Namely, suppose the relational paradigm had been defined solely with its structural part described so far. In that case, the information of a system could be organized in a structured way, in terms of tables, but developers and users would not be able to do anything useful with it — the information could not be modified or even created. Alternatively, developers and users would have to access the data through the underlying internal (physical) data representation, dealing with files, pointers, and other low-level access mechanisms. Therefore, the model would be almost useless, or at least extremely impractical. Although this seems to be trivial and obvious, it is emphasized here simply because some more advanced paradigms lack exactly this aspect! Even early relational DBMS implementations did not support all operations upon the data structure, at least not in a standard and intuitive way. Similarly, the early specifications of UML also suffered from the same problem of lacking a precise definition of actions upon the structure and their semantics!

The relational paradigm defines several elementary operations upon the presented data structure available to applications. By accessing the database through these abstract operations, applications and users do not have to be aware of the internal data representation. This makes them independent on that internal representation and its potential change. These operations are what actually ensure the relational view to the data in the database. These operations are generic, meaning that they can be applied to any particular table, unless stated differently by some domain-specific or authorization constraints. Such generic elementary operations are often called *actions*.

## Part V: Supplemental

---

The way these actions are concretely specified and issued is a matter of concrete notation used for their specification and its syntax. Such notation is defined by a *surface action language*, which is overlaid upon the underlying semantics of the actions and without affecting them. One such standard language, called SQL, will be described later in this chapter. Concrete implementations of the relational paradigm may define their own proprietary action languages, just as different procedural programming languages provide different notations for the same concept of procedure invocation, for example.

Following are the generic elementary operations (actions):

- **Insert Record** — This operation creates and inserts a new record or a set of records into a given table. The target table is defined as a parameter of the operation. Optionally, the operation may also set the values of some or all fields of the new record(s). These values are provided as other optional parameters of the operation. If the operation does not set a value of a field in the new record, that field will take a default value defined by the description of the table or by its type. Otherwise, if the value is not provided either in the operation, or in the table definition, or by the field's type, it is set to a special `null` value, indicating that the field does not have any concrete value of its type. For all uniqueness and other constraints on the fields of the table (such as the implicit constraint on the primary key), the DBMS controls the creation of a new record. If the field is set to a value that would violate a constraint, the DBMS raises an exception, and the operation is not accomplished. For example, the following command (specified using a notation that will be described later) inserts a record into the `Course` table, setting the fields to the given values:

```
INSERT INTO Course (name,duration,price,level,outsource)
VALUES ('Making Spaghetti Western Movies',
 25, 249.9, 'Advanced', true)
```

There is also a complex insertion command, where an entire set of records, returned by an expression encompassing products, selections, and projections of tables, can be inserted into the given table.

- **Delete Records** — This operation deletes one or more records from a table. The table is given as a parameter of the operation. The records to be deleted are identified by a selection operation on the table (that is, by a condition that is checked for the records of the table). The records that satisfy the condition will be deleted from the table. For example, the following command deletes all records from the `Course` table that have their names set to "Making Spaghetti Western Movies":

```
DELETE FROM Course WHERE name='Making Spaghetti Western Movies'
```

You might note that by using conditions instead of identifying specific records, you can delete potentially multiple records whose values satisfy the condition. This way of deleting or otherwise affecting records in general is common in practice because it frees the programmer from having to know about which specific records contain the necessary data.

- **Update Values** — This operation writes new values to the fields of certain records of a table. The table is provided as a parameter of the operation. The records that will be updated are specified, as in the Delete Records operation, by a condition that will select the records. The fields and the values that will be written are specified as in the Insert Records operation. Similarly, as in the Insert Records operation, the DBMS controls the update of each selected record. If the field is set to a value that would violate a constraint, the DBMS raises an exception, and the operation is not

accomplished. For example, the following operation writes new values to the selected record(s) of the *Course* table:

```
UPDATE Course SET outsourced = true
WHERE name='Making Spaghetti Western Movies'
```

- ❑ **Select Records** — This is a general operation for retrieving information from a relational database. It actually executes an expression encompassing products, selections, and projections of tables, and returns a set of records as the result of the expression. The semantics of the operation were described formally earlier in this chapter. The standard notation of this operation will be presented later, when SQL is introduced.

The described operations act exclusively on the contents of the tables, and are thus called the *data manipulation* operations. The surface language for these is referred to as the *data-manipulation language (DML)*. They do not act on the very definition of the structure, that is, on the model itself or the *database schema*.

As stated previously, the structure of tables and their fields (that is, the relational model or the database schema) is taken to be predominantly static. However, relational DBMSs support operations upon the database schema itself, allowing developers to modify the database schema even during the exploitation of the system, when the tables already contain data.

Following are some of the most notable operations from this category:

- ❑ **Create Table** — Creates a new table with the defined fields and their types. The name of the table, its fields, their types, constraints, and the set of fields constituting the primary key are defined as the parameters of the operation.
- ❑ **Remove Table** — Removes a table from the system.
- ❑ **Insert Field** — Inserts a new field with the given name and type into an existing table.
- ❑ **Remove Field** — Removes a field from the given table. The operation may fail if the removal of the field would violate uniqueness constraints of the existing records.
- ❑ **Modify Field** — Modifies the name and/or the type of an existing field of a table. The concrete effect of the modification of the field's type depends on the concrete case, because some conversions of the existing values into the new type cannot be feasible. Additionally, the operation may fail if the modification of the field would violate uniqueness constraints of the existing records. Anyway, this operation can be highly questionable in practice and should be generally avoided. Different implementations of DBMSs vary significantly in treating this operation.

### Section Summary

- ❑ The relational paradigm supports some generic, elementary operations on the contents of the tables, referred to as *data manipulation actions*, which define behavioral elements of a model:
  - ❑ **Insert Records** — Inserts a set of new records into a table, and optionally sets their field values.

*Continued*

- ❑ **Delete Records** — Removes a set of selected records from a table.
- ❑ **Update Values** — Writes the given values into the given fields of a set of selected records.
- ❑ **Select Records** — Returns a set of records by evaluating a composed expression of retrieval operations (product, selection, and projection).
- ❑ Relational DBMSs support operations upon the database schema itself:
  - ❑ **Create Table** — Creates a new table in the system.
  - ❑ **Remove Table** — Removes a table and all its data from the system.
  - ❑ **Insert Field** — Inserts a new field into an existing table.
  - ❑ **Remove Field** — Removes an existing field from the given table.
  - ❑ **Modify Field** — Modifies the name and/or the type of an existing field.

## Advanced Concepts

Based on the described fundamental concepts, the relational paradigm has been enriched with many other advanced concepts that make modeling and implementation of applications easier, improve usability and performance of systems, or direct design of a database schema. Some of these concepts are briefly described here.

### Views

As already indicated, an operand of a relational operator can be the result of another relational expression (that is, of a retrieval query). In effect, for retrieval purposes, it is completely irrelevant whether the data source is a table that physically stores the data, or a “logical” (or “virtual”) table that is the result of another query. For that reason, relational DBMSs allow definitions of *views* that represent such “virtual” tables (relations) whose content is derived by executing a query. Whenever a view is accessed in the same way as a table, its records are obtained by evaluating the underlying query. Views are defined as part of the database schema and form its integral part along with the tables. Obviously, both tables and views are formally relations.

In that way, views may be used as an additional logical data layer that separates the organization of the data structure in tables from the applications and users that access it. Views can be defined to provide a unified perspective to the data structured and prepared in a way specifically requested by the application or the user, and hide the way that structure is obtained from the underlying table schema. This makes applications even more independent of the modifications of the data model. Tables and views together form a unified relational perspective on the data space, where the data is organized into relations, be they tables or views.

Under certain conditions, a view can be *updatable*, meaning that the contents of the database can be modified by performing data modification actions on the view instead of the tables. The principal precondition

for a view to be updatable is that the source record in an underlying table can be uniquely and unambiguously identified by a record modified in the query. Of course, the modification is propagated to that record in the physical table. If a query is not updatable, modification actions cannot be applied on it. A detailed discussion is beyond the scope of this book.

### **Referential Integrity**

As seen in the example of the Easylearn school system, it is often the case that tables are linked over their foreign key-primary key pairs. However, the basic semantics do not relate the primary and foreign keys by default. In other words, the values of the two keys, belonging to two different tables, are completely independent. This can have some important implications.

Let's consider the same example with persons and the information about their attendance of courses stored in the `PersonAttendCourse` table. If, for example, the value of the primary key in a record from the `Person` table is modified by any means, the values of the corresponding foreign keys in the `PersonAttendCourse` table will not be affected automatically, and will preserve the old, invalid value that will refer to a non-existing or wrong record in the `Person` table. In other words, the foreign keys may become dangling references.

Similarly, if a record from the `Person` table is deleted, the corresponding records in the `PersonAttendCourses` table will not be deleted automatically, and will also represent dangling references. This way, the database will become inconsistent, in the sense that it will contain records with dangling references (that is, invalid foreign keys). It is said that the database's *referential integrity* would be violated in that case.

Of course, it may be a responsibility of the application to preserve the referential integrity of the database, by undertaking necessary actions on modifications or deletions of a primary key. However, it would be much easier and less error-prone if the DBMS took care of the referential integrity. This is why most DBMSs offer the capability to automatically preserve referential integrity. At database design time, developers specify the bindings between primary and foreign keys. The DBMS stores the information about the bindings in the database schema. Then, for each such binding, the developer specifies the way the referential integrity will be kept:

- Whether the value of the foreign key will be automatically updated whenever the corresponding primary key is updated (if this is allowed at all) for all the records that have that value of the foreign key. This feature is often called *cascading update*.
- Whether the records having the value of the foreign key will be automatically deleted whenever the corresponding record with that value of the primary key is deleted (if this is allowed at all). This feature is often called *cascading deletion*.

This way, during the exploitation of the system, the DBMS automatically preserves the referential integrity for all specified bindings, without any additional intervention by the developer or user.

### **Triggers and Stored Procedures**

It is sometimes necessary to react to actions on records or field values with some domain-specific functionality. For example, in the Easylearn school information system, users may want to prevent a person from attending outsourced courses (provided by other schools) if the person does not have a credit card

## Part V: Supplemental

---

because the Easylearn school must have a means to regulate the payments with other schools. Therefore, if a user toggles a value of the field `Course.outsourced` from `false` to `true`, the database must be checked against persons who are enrolled in that course, but who do not have credit cards (that is, that have `null` values in the field `Person.cardnum`).

DBMSs often meet such needs with the concepts of *triggers* and *stored procedures*. At design time, the developer specifies a *trigger* for a generic operation on a certain field or table. For the given example, the developer may specify that the DBMS should raise an event whenever the value of the `Course.outsideSource` field is modified. Then, the developer can attach a *stored procedure* to that trigger. A stored procedure is a piece of code, written in a language supported by the DBMS, and stored as a part of the database definition, which is executed on the trigger. For the mentioned example, the developer may program a stored procedure that checks the described constraint and prevents the modification of the field `Course.outsourced` if the condition is not satisfied, or takes some other action.

Although the concepts of triggers and stored procedures are very useful, they often cause significant difficulties with portability. Although the basic concepts of the relational paradigm are supported by all DBMSs in more or less the same way, it is completely the opposite with triggers and stored procedures. First, the set of available types of triggers is dependent on the DBMS itself. Similarly, the language used to program stored procedures is often vendor-specific. This prevents an application that relies on stored procedures to be easily ported to another DBMS.

Another yet more serious problem arises when triggers and stored procedures are intensively used throughout the application for implementing its business logic. The business logic then becomes cluttered and difficult to understand, follow, and maintain because it is heavily spread across stored procedures associated with triggers linked to different parts of the database schema. Possible inconsistencies of their definitions or even domino effects (an action upon a record triggers a procedure, which updates another record, which then triggers another procedure, and so on) can be difficult to reveal and control, too. As a result, although triggers and stored procedures are concepts simple to understand and define, it is considered to be a bad practice to base the entire business logic of complex applications on them. Instead, they should be treated as a helper tool to efficiently and easily handle domain-specific consistency rules only.

## Indices

An *index* is an auxiliary structure attached to a table (not a view) that improves the performance of accessing its records. Namely, the records of a table may be scattered on the disk, and the retrieval of the information may be too slow if the records are accessed sequentially. For this purpose, the developer may specify that several indices may be created for that table. An index enables a value of a field (or a tuple of values, or even an expression over the field values) to be quickly found in the auxiliary structure, from where the physical position of its record on the disk is immediately obtained. Therefore, indices speed up information retrieval, but may slow down updates because updates of values may also require the corresponding updates of the index structure.

Indices and their structures are not directly visible to developers and users. DBMSs automatically maintain their structures. Developers should only specify for which fields or sets of fields indices should be created. It is wise to create an index for all fields for which the application performs retrievals and searches, and especially for primary keys and foreign keys because they are often used in table joins. Additionally, DBMSs often require that an index be created for each primary key, or for each field with uniqueness constraints, because the only reasonable way to enforce the uniqueness efficiently is through indices. Virtually all DBMSs implicitly create indices on primary keys.

## Normalization

As you may have already noticed, there is virtually an unlimited number of ways a database schema may be defined for a problem domain. For the Easylearn school system, you could have stored the information about persons and courses they attend in several different ways. For example, instead of creating a separate table `PersonAttendCourse`, you could have stored the information about attended courses in the `Person` table, as shown here.

| <b>name</b>    | <b>postal</b>                    | <b>phone</b> | <b>email</b>        | <b>cardnum</b> | <b>courseID</b> |
|----------------|----------------------------------|--------------|---------------------|----------------|-----------------|
| John Smith     | 45 5th Avenue, New York City, NY | 432002       | john@smith.com      | 455523         | 1               |
| John Smith     | 45 5th Avenue, New York City, NY | 432002       | john@smith.com      | 455523         | 2               |
| John Smith     | 45 5th Avenue, New York City, NY | 432002       | john@smith.com      | 455523         | 3               |
| Steven Walsh   | 22 45th St., Denver, CO          | 234456       | swalsh@computer.com | 333467         | 1               |
| Steven Walsh   | 22 45th St., Denver, CO          | 234456       | swalsh@computer.com | 333467         | 3               |
| Alice Thompson | 715 23th St., Washington, DC     | 255667       | alice@univ.edu      | 210054         | 2               |
| ...            | ...                              | ...          | ...                 | ...            | ...             |

Here, the foreign key referring to the attended course is stored together with the data about a person. However, because a person may attend several courses, several records in the `Person` table carry the same, redundant information about a person's data. Obviously, this design is very poor because of a very high level of redundancy, which causes significant run-time penalties and consistency maintenance issues.

The theory of relational models has studied this issue very deeply and formalized it through the concept of *normalization*. Simply said, normalization is a process of transforming the database schema into a so-called *normal form*, which is a form with as little redundancy as possible. The problem is rather complex, and relies on the concept of *functional dependencies*, which are criteria that may or may not be satisfied in a database design. The theory defines several types of normal forms depending on which functional dependencies the database schema satisfies. However, an in-depth elaboration of these concepts falls far beyond the scope of this book.

### Section Summary

- A *view* is a “virtual” table (relation) whose content is derived from a query.
- DBMSs can support automatic preservation of referential integrity by disallowing dangling references of foreign keys to deleted or modified primary keys.

*Continued*

- ❑ Developers may specify significant events in the database (insertion, deletion, or modification of records), called *triggers*, on which procedures stored in the database are automatically invoked.
- ❑ Indices are transparent, auxiliary structures associated with tables or, more precisely, with some of their fields that improve the performance of information retrieval over these fields.
- ❑ Normalization is the process of transforming the database schema into a form that avoids information redundancies.

# SQL

In order to overcome differences in DBMSs and the way they support access to information in databases, a standard language for information retrieval and modification has been invented. This is the *Structured Query Language (SQL)*, often pronounced as “sequel” by practitioners). SQL has been used as a unified language for accessing relational databases from applications or user interface shells in a way that does not depend on the DBMS implementation. It is a standard language supported by most available DBMSs. This brief survey of SQL serves as an illustration of its capabilities and not as a complete reference for the language.

SQL is not a programming language in the traditional sense because it is not used to write entire applications. It is just a kind of a scripting language and an interface between programming languages used for writing applications and DBMSs. More precisely, SQL is a surface action language for the relational paradigm. Applications thus use isolated *statements* written in SQL (or *queries*, as they are often called in SQL). SQL statements are strings of characters constructed within the application or the user interface shell. The statements are then sent to the DBMS for interpretation. The DBMS interprets the statements and performs actions upon the database or returns a result of the query. The result is given in the form of a *record set*, which is a relation as defined before, meaning that it fully complies with the relational paradigm. The record set has the fields specified by the query, and can be iterated by the application in order to retrieve the needed information, or used in another way.

Statements in SQL may be divided into three main groups:

- ❑ **Statements (or queries) that retrieve information from the database (that is, which select records meet certain criteria)** — These are the most important SQL statements for the context of this book, and will therefore be described in more detail here. They are called *select queries*.
- ❑ **Statements that modify data in the database** — These are the statements that support inserting or deleting records, or modifying field values. They will be described briefly here.
- ❑ **Statements that work upon the database schema (that is, upon the metadata of the database)** — The metadata are the data about the database schema managed by the DBMS. In other words, the metadata is the data about the structure of the domain-specific data in the database. These are the specifications of the tables, their fields, properties of the fields (name, type, uniqueness), keys, constraints, indices, and so on. This category of SQL statements supports modifications of the database schema, such as *create/remove a table*, or *create/remove/modify a field*. A more detailed description of these statements is beyond the scope of this book.

SQL statements are case-insensitive. However, the examples here will have the SQL keywords in uppercase and user-defined identifiers in lowercase, with initials in uppercase for table names. In the informal descriptions of the syntax of SQL in the following sections, square brackets [] are used to enclose optional parts of statements; they are not part of the statements. Similarly, angle brackets <> enclose parts of statements that have separate definitions or are described in the text; they are not part of the statements.

### Section Summary

- SQL is a standard language for accessing relational databases.
- SQL is not a complete programming language but, rather, it is a scripting language used within other languages or shells to access a relational database. SQL is a surface action language for the relational paradigm. SQL statements are interpreted by DBMSs.
- SQL supports three kinds of statements:
  - SELECT queries** — Retrieve information from the database without modifying it — that is, select records that satisfy certain criteria. They return record sets.
  - Modification statements** — Insert or remove records, or modify field values in the database.
  - Metadata access statements** — Access or modify the metadata — that is, the database schema itself.

## ***SELECT Statements***

The following discussion examines the SELECT statements in SQL.

### ***Basic Form***

The basic form of a SELECT query is as follows:

```
SELECT <field_list> FROM <table_list> WHERE <where_expression>
```

The query returns a record set containing the fields specified in the *<field\_list>* from the product of the tables (including views) specified in the *<table\_list>*, satisfying the condition given in the *<where\_expression>*. The syntax and meaning of each of these parts will be described later. For example, the following query returns a set of records specified by the join of the Person, PersonAttendCourse, and Course tables:

```
SELECT Person.name, Course.name FROM Person, PersonAttendCourse, Course
WHERE Person.ID=PersonAttendCourse.personID AND
PersonAttendCourse.courseID=Course.ID
```

The result will have two fields representing the names of the persons and their attended courses. For the example given before, this query will return the set of the following records.

## Part V: Supplemental

---

| Person.name    | Course.name              |
|----------------|--------------------------|
| John Smith     | Car Mechanics for Ladies |
| John Smith     | Cooking for Dummies      |
| John Smith     | Computers for Everybody  |
| Steven Walsh   | Car Mechanics for Ladies |
| Steven Walsh   | Computers for Everybody  |
| Alice Thompson | Cooking for Dummies      |
| ...            | ...                      |

More precisely, the result of a SELECT query does not need to be a set in the mathematical sense, where each element is unique. That is, it is not a true projection of a selected product of tables. By default, a SELECT query may return a collection of records with possible duplicates. For example, let's assume that the Course table has the following records.

| ID  | name                     | duration | price | level        | outsourced |
|-----|--------------------------|----------|-------|--------------|------------|
| 1   | Car Mechanics for Ladies | 15       | 45.0  | Basic        | false      |
| 2   | Cooking for Dummies      | 20       | 99.9  | Basic        | false      |
| 3   | Computers for Everybody  | 50       | 450.0 | Intermediate | true       |
| 4   | Computers for Everybody  | 50       | 450.0 | Advanced     | true       |
| ... | ...                      | ...      | ...   | ...          | ...        |

Consider the following query:

```
SELECT name FROM Course
```

This will return a result comprised of four records, two of which have the same values, as shown in the following table.

| name                     |
|--------------------------|
| Car Mechanics for Ladies |
| Cooking for Dummies      |
| Computers for Everybody  |
| Computers for Everybody  |
| ...                      |

If a true set is needed, without duplicated records, the query should be qualified with the keyword DISTINCT following the keyword SELECT. The result of a DISTINCT SELECT query is a true set, without duplicated records. For the previous example, consider the following query:

```
SELECT DISTINCT name FROM Course
```

The result of this query will be as shown in the following table.

| name                     |
|--------------------------|
| Car Mechanics for Ladies |
| Cooking for Dummies      |
| Computers for Everybody  |
| ...                      |

### Table List

The *<table\_list>* part of a SELECT query specifies the list of tables whose product will be made. More generally, the elements of the list can be all kinds of record sets (that is, relations), including tables, views, or other nested SELECT statements that also return record sets. As described before, the product will combine all records from all record sets in the list. For example, consider the following query:

```
SELECT Person.name, Course.name FROM Person, Course
```

This will return the set shown in the following table.

| Person.name    | Course.name              |
|----------------|--------------------------|
| John Smith     | Car Mechanics for Ladies |
| John Smith     | Cooking for Dummies      |
| John Smith     | Computers for Everybody  |
| Steven Walsh   | Car Mechanics for Ladies |
| Steven Walsh   | Cooking for Dummies      |
| Steven Walsh   | Computers for Everybody  |
| Alice Thompson | Car Mechanics for Ladies |
| Alice Thompson | Cooking for Dummies      |
| Alice Thompson | Computers for Everybody  |
| ...            | ...                      |

As shown, the *<table\_list>* can be a list of record sets separated by commas. However, a record set in the list may be also given an *alias*. The alias is specified as an identifier following the keyword AS

## Part V: Supplemental

---

after the table name or nested query. The keyword AS may be omitted. The alias can then be used in the *<where\_condition>* or *<field\_list>* as the alternative name of the record set to refer to its fields.

An alias may serve just as a convenient abbreviation of a table name, as shown in the following example:

```
SELECT p.name, c.name FROM Person AS p, Course AS c
```

Aliases are particularly useful (and even necessary) when products are made on several occurrences of a table name in the *<table\_list>* of a query. In such case, each table occurrence must have a unique alias. For example, consider a HousePart table describing some parts of a house, and incorporating the information of the part-subpart hierarchy. Each part may have at most one enclosing part of which it is a subpart, as shown in the following table for HousePart.

| ID  | name               | superpartID |
|-----|--------------------|-------------|
| 1   | House              | NULL        |
| 2   | Dining room        | 1           |
| 3   | Bedroom            | 1           |
| 4   | Lobby              | 1           |
| 5   | Entrance door      | 4           |
| 6   | Entrance door knob | 5           |
| 7   | Garden window      | 2           |
| ... | ...                | ...         |

The following query returns the direct subparts of the house:

```
SELECT subpart.name FROM HousePart AS superpart, HousePart AS subpart
WHERE superpart.ID=subpart.superpartID AND
superpart.name = 'House'
```

As already stated, a record set in *<table\_list>* does not need to be a table or a view defined in the database schema. It can also be a record set obtained by another SQL select query. In other words, queries can be nested. A nested query is specified within parentheses at the place of a table name. For example, this query returns the same result as the previous one:

```
SELECT subpartName
FROM
(SELECT superpart.name AS superpartName, subpart.name AS subpartName
FROM HousePart AS superpart, HousePart AS subpart
WHERE superpart.ID=subpart.superpartID)
WHERE superpart.name = 'House'
```

### Field List

The field list specifies the fields that the resulting record set will have. The list contains the names of the fields from the record sets listed in the `<table_list>`, possibly qualified by the table names or their aliases. For example, the resulting record set of the following query will have the fields named `Person.name` and `Course.name`, and these fields will have the values from the corresponding fields from the selected product:

```
SELECT Person.name, Course.name FROM Person, PersonAttendCourse, Course
WHERE Person.ID=PersonAttendCourse.personID AND
PersonAttendCourse.courseID=Course.ID
```

Instead of specifying a comma-separated list of fields, a query may return a record set containing all fields of a product. This is specified with an asterisk instead of the field list:

```
SELECT * FROM Person, PersonAttendCourse, Course
WHERE Person.ID=PersonAttendCourse.personID AND
PersonAttendCourse.courseID=Course.ID
```

The resulting record set has a fixed set of named fields. The names of these fields are equal to the fields from the product referenced from the field list. However, the fields in the `<field_list>` may be given aliases, too, in the same way as the record sets in the `<table_list>`. The aliases are then used as the names of the fields of the resulting record set. Consider the following example:

```
SELECT Person.name AS personName, Course.name AS courseName
FROM Person, PersonAttendCourse, Course
WHERE Person.ID=PersonAttendCourse.personID AND
PersonAttendCourse.courseID=Course.ID
```

The result of this query will have the fields named `personName` and `courseName`. This feature is particularly useful when queries are nested, as in the following example:

```
SELECT subpartName
FROM
(SELECT superpart.name AS superpartName, subpart.name AS subpartName
FROM HousePart AS superpart, HousePart AS subpart
WHERE superpart.ID=subpart.superpartID)
WHERE superpart.name = 'House'
```

### Where Condition

The `<where_condition>` clause is an expression that must evaluate in a Boolean result (true or false). It is evaluated for each record of the product defined by the `FROM` clause. The resulting record set of the query contains those and only those records from the product for which the expression evaluates to true.

The `<where_condition>` is evaluated for each record of the product and in the context of that record. This means that the expression may refer to the fields from the record sets in the `<table_list>`, possibly qualified by the table names or aliases. Consider the following example:

```
SELECT Person.name AS personName, Course.name AS courseName
FROM Person, PersonAttendCourse, Course
```

## Part V: Supplemental

---

```
WHERE Person.ID=PersonAttendCourse.personID AND
PersonAttendCourse.courseID=Course.ID
```

The expression consists of operands and operators. The operands depend on the operator, but may generally be the following:

- ❑ **References** — These are terms referring to the fields from the record sets in the *<table\_list>*, possibly qualified by the table names or aliases.
- ❑ **Literals** — These are constants like numbers or strings. String-literals are enclosed in single quotes (' ) (for example, 'House'). A string-literal as the right-hand operand of the LIKE operator (to be described later) may contain special wildcards defining regular expressions, such as %, which stands for any (possibly empty) sequence of characters. The set of supported wildcards may depend on the DBMS.
- ❑ **Built-in functions** — The set of built-in functions may depend on the concrete DBMS, and some DBMSs may offer more built-in functions than specified in standard SQL. They are out of the scope of this discussion.
- ❑ **Sub-expressions** — Expressions may be nested, as usual. Whenever the default order of evaluation is not suitable, the order of evaluation may be changed by enclosing sub-expressions in parentheses.

SQL is rich in operators. The set of supported operators may, however, depend on the DBMS. Some DBMSs may also offer more powerful operators and built-in functions than specified in standard SQL. Following are the most important operators:

- ❑ **Arithmetic operators**, working with numeric operands and producing numeric results (such as the usual arithmetic operators +, -, \*, and /).
- ❑ **Relational operators**, working with numeric operands and producing Boolean results (<, >, =, <=, >=).
- ❑ **Logic operators**, working with Boolean operands and producing Boolean results (AND, OR, XOR, NOT).
- ❑ **String operators**, working with strings and producing Boolean results. These are: = (for full matching of strings), <, >, <=, >= (for comparison of strings in lexical order), and LIKE (returning true if the left-hand operand matches the right-hand operand given as a regular expression). For example, the operation Person.name LIKE '%John%' returns true for all records that have a substring 'John' in their names.
- ❑ **Set operators**, working with set operands and returning Boolean results. The most notable is the IN operator, which returns true if the left-hand operand (which is a scalar value) exists in the set given by the right-hand operand. The set may be given as a comma-separated list of literals enclosed in parentheses, such as ('Intermediate', 'Advanced'), or even as another select query returning a single-field record set. For example, the operation Course.level IN ('Intermediate', 'Advanced') returns true for all records that have the value of the field level set to either 'Intermediate' or 'Advanced'.

### Union Queries

A *union query* is a combination of two or more SELECT queries. The union query then returns the union of the records from all combined select queries. For example, the following query returns the set of names of all persons and all courses in the Easylearn database:

```
SELECT name FROM Person
UNION
SELECT name FROM Course
```

### Aggregate Functions

Instead of returning a record set, a `SELECT` query may apply an *aggregate function* upon the record set obtained in the usual way, and return the value of the result of that function. The aggregate function is calculated cumulatively for all records from the record set that would be returned by the corresponding query.

For example, you may be interested in the number of courses attended by John Smith, not in the names of the courses themselves. The query that would return such a result would be as follows:

```
SELECT Count(Course.name)
 FROM Person, PersonAttendCourse, Course
 WHERE Person.ID=PersonAttendCourse.personID AND
 PersonAttendCourse.courseID=Course.ID AND
 Person.name='John Smith'
```

The `Count` function simply counts the number of records returned by the selected product and returns the number as the result of the query. Although the parameter of the `Count` function may refer to one or more fields, the function disregards the values of the fields, but just counts the number of records. Generally, the result will be equal as in the query (although differences may occur in the way the fields with null values are counted in these two forms):

```
SELECT Count(*)
 FROM Person, PersonAttendCourse, Course
 WHERE Person.ID=PersonAttendCourse.personID AND
 PersonAttendCourse.courseID=Course.ID AND
 Person.name='John Smith'
```

Following are several useful aggregate functions supported by SQL:

- ❑ `Count(<field_list>)` — Counts the number of records in the record set specified in the rest of the query.
- ❑ `Sum(<expression>)` — Returns the sum of the values returned by the expression applied to all records in the record set specified in the rest of the query. The expression may be a compound expression returning a numeric value, and referring to the fields from the record sets in the `<table_list>`.
- ❑ `Avg(<expression>)` — Returns the mean value of the values returned by the expression applied to all records in the record set specified in the rest of the query. The expression may be a compound expression returning a numeric value, and referring to the fields from the record sets in the `<table_list>`.
- ❑ `Min(<expression>), Max(<expression>)` — Returns the minimum or maximum value respectively of the values returned by the expression applied to all records in the record set specified in the rest of the query. The expression may be a compound expression returning a numeric value, and referring to the fields from the record sets in the `<table_list>`.

## Part V: Supplemental

---

### Ordering and Grouping

By default, the result of a `SELECT` query is a collection of records without a specific order. If a specific order of the records in the result set is needed, the query should be qualified with the `ORDER BY` clause at the end of the query. The `ORDER BY` clause follows the (optional) `WHERE` clause and specifies the sorting order of the result according to the fields given in the `ORDER BY` clause. The syntax is as follows:

```
ORDER BY <order_field_list>
```

The `<order_field_list>` is a comma-separated list of elements in the following form:

```
<field_name> [ASC | DESC]
```

The result is sorted by the first field in the list, in the ascending (`ASC`) or descending (`DESC`) order. If `ASC` and `DESC` are omitted, the default order is ascending. Then, within the group of the same values of the first field, the records are sorted according to the values of the second field specified in the list, and so on.

For example, the following query returns the list of names and prices of courses ordered by their price in the descending order:

```
SELECT name, price FROM Course ORDER BY price DESC
```

Another optional specifier may follow the (optional) `WHERE` clause: `GROUP BY <field_list>`. It combines records with identical values in the specified field list into a single record. An aggregate value is created for each record if an SQL aggregate function (such as `Sum` or `Count`) is included in the `SELECT` statement. For example, let's say the table `Course` looks like the following.

| ID | name                     | duration | price | level        | outsideSource |
|----|--------------------------|----------|-------|--------------|---------------|
| 1  | Car Mechanics for Ladies | 15       | 45.0  | Basic        | false         |
| 2  | Cooking for Dummies      | 20       | 99.9  | Basic        | false         |
| 3  | Computers for Everybody  | 50       | 450.0 | Intermediate | true          |
| 4  | Computers for Everybody  | 50       | 450.0 | Advanced     | true          |
|    | ...                      | ...      | ...   | ...          | ...           |

Let's say you then issue the following query:

```
SELECT Sum(price) AS totalPrice FROM Course GROUP BY price
```

This returns the record set shown in the following table.

| totalPrice |
|------------|
| 45.0       |
| 99.9       |
| 900.0      |
| ...        |

### Section Summary

- ❑ A SELECT SQL statement has the following general basic form:

```
SELECT [DISTINCT] [*|<field list>]
 FROM <table_list> [WHERE <where_expression>]
```
- ❑ A SELECT statement returns the collection of records having the fields specified in the <field\_list>, from the product of the record sets in the <table\_list>, selected by the criterion in the <where\_expression>. If the DISTINCT keyword is present, the result is a set without duplicated records.
- ❑ The <table\_list> is a comma-separated list specifying the record sets and their aliases in the product, in the following form:  
`<table_name_or_nested_query> [AS] alias`
- ❑ The <field\_list> is a comma-separated list of references to the fields from the record sets in the <table\_list>, with possible aliases that will be used as the names of the query's resulting fields.
- ❑ The <where\_expression> is a Boolean-resulting expression applied to each record from the set. It consists of operands and operators.
- ❑ Operands may refer to the fields from the record sets in the <table\_list> or may be literals, built-in functions, or sub-expressions enclosed in parentheses.
- ❑ Operators include arithmetic, relational, logic, string, or set operators.
- ❑ A union query is a combination of several SELECT queries that returns the union of the records of the combined SELECT queries.
- ❑ A SELECT query may return, instead of a result set, a value that is the result of an aggregate function applied to the resulting records, such as Sum, Count, Avg, Min, or Max.
- ❑ The resulting records of a SELECT query may be ordered (the ORDER BY clause) or grouped (the GROUP BY clause).

# Data Modification Statements

SQL statements of this kind perform the generic modification operations (actions) upon the database. There are three main kinds of data modification statements: insert records, delete records, and update records.

## Insert Records

This kind of SQL statement inserts one or several records into an existing table. There are two forms of this statement. Following is the first form:

```
INSERT INTO <table_name> [<field_list>] VALUES (<value_list>)
```

This statement inserts a single record into the table or an updatable view given with the *<table\_name>* and sets the fields given by the *<field\_list>* to the values given in the *<value\_list>*, respectively. If the statement does not set a value of a field in the new record, that field will take a default value defined by the description of the table or implicitly by its type. Otherwise, if the value is not provided either in the operation, or in the table definition, or by the field's type, it is set to a special null value, indicating that the field does not have any concrete value from its type. For all constraints on the fields of the table (such as the implicit constraint on the primary key), the DBMS controls the creation of a new record. If the field is set to a value that would violate a constraint, the DBMS raises an exception, and the operation is not accomplished.

For example, the following statement inserts a record into the *Course* table, setting the fields to the given values:

```
INSERT INTO Course (name,duration,price,level,outsourced)
 VALUES ('Making Spaghetti Western Movies',25,249.9,'Advanced',true)
```

The second form is used to insert a set of records into the given table or updatable view:

```
INSERT INTO <table_name> (<field_list>) <SELECT_statement>
```

This statement inserts the entire record set returned from the *<SELECT\_statement>*, where the fields given in the *<field\_list>* are set to the values of the fields from the resulting record set of the *<SELECT\_statement>*, respectively.

## Delete Records

The following statement deletes the records selected by the *WHERE* condition from the given table:

```
DELETE FROM <table_name> [WHERE <where_condition>]
```

## Update Fields

The following statement writes new values into the fields of the records selected by the *WHERE* condition:

```
UPDATE <table_name> SET <assignment_list> [WHERE <where_condition>]
```

The *<assignment\_list>* is a comma-separated list of assignments, each being in the following form:

```
<field_name> = <expression>
```

Here, the specified field on the left-hand side of the assignment will be set to the value returned by the *<expression>*.

### Section Summary

- ❑ An INSERT statement inserts one or more records into the given table, optionally setting the values of the fields of the new record(s). It has two forms:
  - ❑ Single-record form inserts one record into the table:

```
INSERT INTO <table_name> [(<field_list>)
VALUES (<value_list>)]
```
  - ❑ Multiple-record form inserts a record set returned by the SELECT statement:

```
INSERT INTO <table_name> (<field_list>)
<SELECT_statement>
```
- ❑ A DELETE statement deletes a set of selected records from the table:

```
DELETE FROM <table_name> [WHERE
<where_condition>]
```
- ❑ An UPDATE statement writes new values into the fields of the selected records:

```
UPDATE <table_name> SET <assignment_list>
[WHERE <where_condition>]
```

## DBMS Support

Key to the success of the relational paradigm is certainly the strong support of DBMSs. DBMSs serve as run-time environments that support execution of applications based on relational models. The key contributions and roles of the DBMSs are discussed here.

First, DBMSs provide a conceptual mapping from a relational model to the lower-level implementation platform. This encompasses several things. First, developers and users see the relational data model structured into tables, fields, and records, and the DBMS maps that model into an internal data representation built in the target file system or even a raw disk partition. Developers and users do not deal with such subtle details as where the value of a certain field of a certain record of a certain table is stored on the disk, or how it can be accessed. Second, DBMSs support the described generic structural operations and map them into lower-level operations upon the raw data on the disk. Again, developers do not need to worry about how a field value is written to the disk when an Update Field action is issued.

Moreover, DBMSs provide strong support for other elements necessary for the practice, and possibly not strictly falling into the core concepts of the relational paradigm. First, they support distribution. Clients may access databases from remote locations, and even the database storage may be distributed or replicated on multiple servers. Again, the distribution is usually almost completely transparent to developers and users, and they do not need to worry about how the database is remotely accessed from the application over a network.

## Part V: Supplemental

---

Similarly, DBMSs control concurrent, multiuser access to the database. They provide adequate locking mechanisms to ensure consistency of data in the presence of simultaneous access of several users. The locking mechanisms are usually offered on a per-record basis. However, DBMSs differ in the way a lock is acquired. Some DBMSs apply a “pessimistic” approach, meaning that a process should acquire a lock *before* it wants to modify a record. If the record is already locked by another process, the lock is not acquired. The other policy is “optimistic” and assumes that a process tries to modify a record without asking for an explicit lock. The modification will be successful only if there are no other processes that want to modify the same record at the same time. Otherwise, the attempt to modify the record fails.

DBMSs ensure fault tolerance, usually by the described concept of transaction. A critical sequence of actions upon the database is enclosed by a transaction, meaning that it will be either executed as a whole, or not executed at all, in case one of the actions in the transaction fails. In other words, in case of a failure, the actions are rolled back so that the database state remains consistent.

Finally, DBMSs support security through user access rights. User access rights are usually specified in terms of which generic operations for which tables are accessible to which users. For example, a user access right specifies that the user *X* is allowed to insert and read records of the table *T*, but not to update and delete them.

### Section Summary

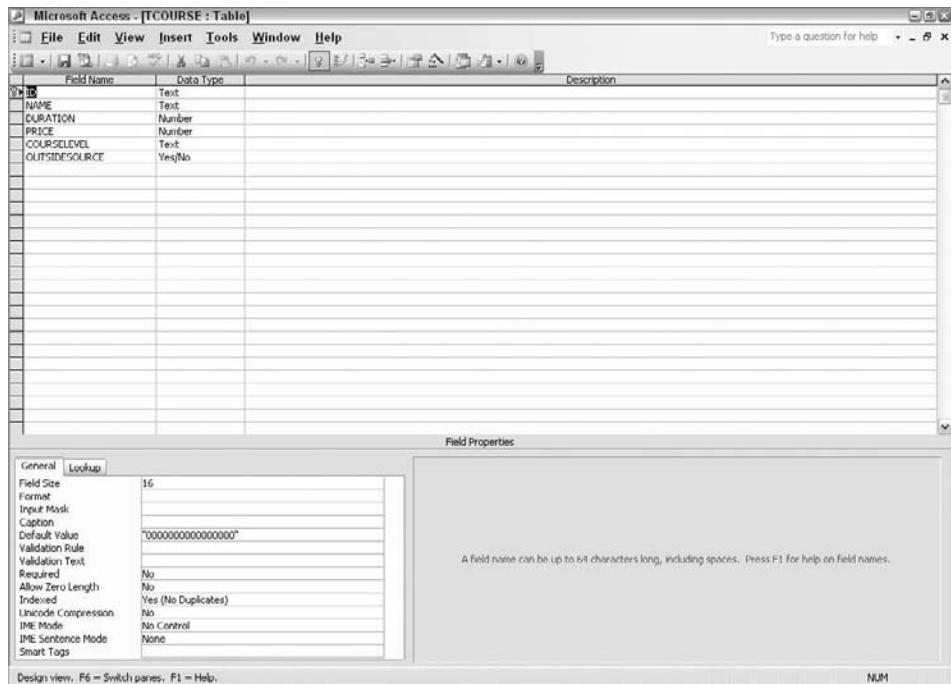
- ❑ Commercial relational DBMSs support all important practical aspects:
  - ❑ Conceptual mapping of the structure and actions from a more abstract relational model to lower-level implementation layers and internal representations of data.
  - ❑ Transparency of distribution of the application and the database.
  - ❑ Concurrent, multiuser access to the database by record-based locks.
  - ❑ Fault-tolerance, by transactions.
  - ❑ User access control, in terms of specifying which user has access to which generic operations upon which tables in the database

## Development Tools Support

Tools that support development of databases and applications based on the relational paradigm may offer significant help to developers. First, they may support database design in an intuitive manner, often through visual database modeling. For example, they help in defining tables, fields, and their properties (names, types, default values, indices, and so on), as shown in Figure 22-1a. Then, they may support visual modeling of relationships between tables, providing a diagrammatic view to the database schema and incorporated referential integrity rules, as shown in Figure 22-1b.

Second, they offer considerable help in developing applications, especially their GUIs (see Figure 22-2). The traditional approach of building GUIs for relational database applications is to organize the GUI into *forms* (or *masks*). The forms present values of fields of records in the corresponding form controls, such

as textboxes, list boxes, checkboxes, and so on. The data from the fields of one record may be presented in the controls of one form (see Figure 22-2, upper part). Alternatively, the data from an entire record set may be presented in a tabular form (see Figure 22-2, lower part).



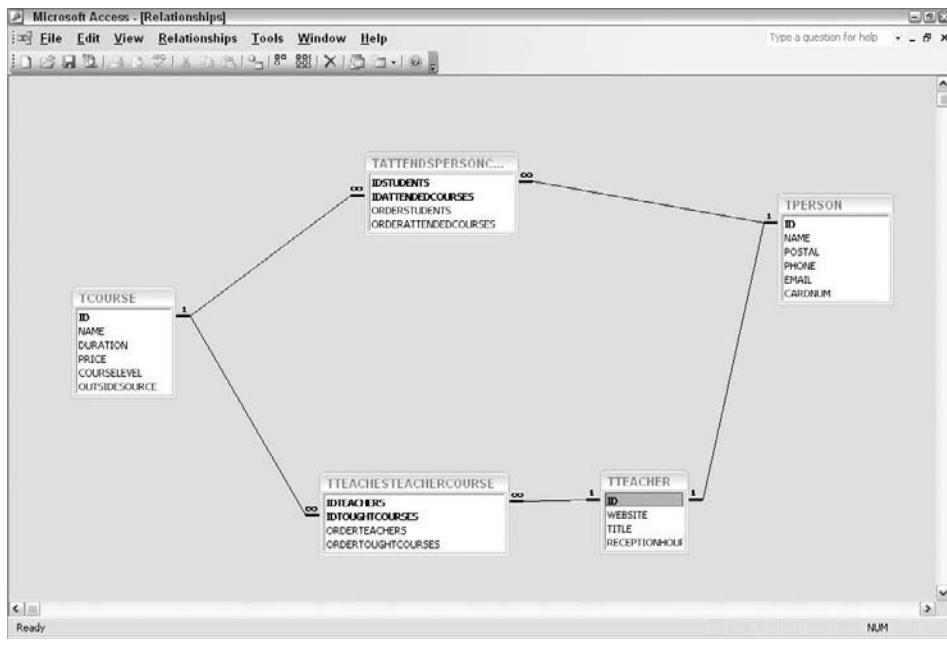
(a)

**Figure 22-1:** An example of a relational database design tool (a) Support for defining tables, fields, and their properties.

Even more complicated generic forms are also possible, such as the popular *master-details* form (see Figure 22-2). A master-details form presents the values of the fields of one record in one subform (see Figure 22-2, upper part), called the *master*, and the values of the records from another table, linked to the master record through a specified join, in another subform (called the *details*; see Figure 22-2, lower part). For example, the master part of a master-details form in the Easylearn school system may show the fields of one course, and the details part may show the properties of the persons that attend that course. Tools and run-time engines support generic navigation through forms so that the developer does not need to worry about it. For example, when the user switches to another record in the master part of a master-details form, the values in the details part are automatically refreshed.

Another important feature of development tools is their reporting capability. Predominantly based on SQL, reporting tools allow developers to design reports similar to the way in which they design forms, in the WYSIWYG ("What-You-See-Is-What-You-Get") manner, by making graphical layouts intuitively, and by connecting the data fields in reports to the fields of the underlying SQL queries as their data sources.

## Part V: Supplemental



(b)

Figure 22-1: (b) Visual modeling of tables, their relationships, and referential integrity rules

The screenshot shows the Microsoft Access Form View for the Course table. The form contains fields for Course details and a list of Students.

**Course**

|              |                                     |
|--------------|-------------------------------------|
| ID           | 00002000000104                      |
| Name         | Cooking for Dummies                 |
| Duration     | 20                                  |
| Price        | 99.9                                |
| Course Level | basic                               |
| Outsourced   | <input checked="" type="checkbox"/> |

**Students**

| ID       | name         | postal | phone     | email            | cardnum |
|----------|--------------|--------|-----------|------------------|---------|
| 00001001 | John Smith   | 32000  | 332211321 | john@smith.com   |         |
| 00001010 | Steven Walsh | 32000  | 432112345 | steven@walsh.com | 12345   |
| 00001011 | David Hardy  | 32000  | 345345345 | david@hardy      | 56789   |

Record: 1 of 3

Figure 22-2: The appearance of traditional user interfaces based on the relational paradigm

The help of the GUI design tools is even more significant in the context of multi-tier Web architectures, when the GUI may be executed on a thin client and plugged with the application layer through a controlled, standard Web protocol. In such cases, the tools provide a transparent view to the GUI design, whereby the developer does not care about how the data is transferred between the application server and the client, and how the controls in a form are updated when their values in the database are modified, or vice versa. In other words, contemporary tools provide a unique view to the relational database application development, regardless of the architecture of the distributed application.

### Section Summary

- ❑ Development tools for relational database applications support easy, intuitive, and often visual modeling of the database schema (that is, tables, fields, their properties, indices, relationships, and referential integrity rules).
- ❑ Development tools also support the WYSIWYG ("What-You-See-Is-What-You-Get") approach to visual design of graphical user interfaces, usually based on the tabular or form views to the underlying record sets, and reports, most often built upon SQL queries.
- ❑ The GUI development tools and run-time engines automatically support generic navigation through records and hide the distributed architecture, so that the developer does not care about the data transfer and consistency.



# 23

## Structured Analysis

One of the methods for designing information systems mostly used in the past is *structured analysis* (SA). It models a system from two viewpoints: conceptual and functional. *Entity-relationship* (ER) models are used for conceptual modeling, and *data flow* (DF) models are used for functional modeling. These are described briefly in the following sections.

To be precise, ER was developed independently of DF and SA. It can be (and has been) used in practice for conceptual (data) modeling independently of DF and SA. The same holds for DF and functional modeling. SA is simply a method that has grouped two parts: ER, for conceptual modeling, and DF, for functional modeling of information systems.

### Entity-Relationship Modeling

The entity-relationship (ER) paradigm [Chen, 1976] supports conceptual modeling at a higher level of abstraction and, thus, more efficiently than the relational paradigm. A model developed using the ER paradigm may be automatically and easily mapped to the corresponding relational model because these two paradigms are conceptually close. This section briefly describes the ER paradigm.

#### Basic Concepts

An *entity* represents a discrete object or a “thing” that can be distinctly identified in the problem domain. In the Easylearn system described in Chapter 22, a specific person (for example, John Smith or Steven Walsh) or a specific course (for example, “Cooking for Dummies” or “Computers for Everybody”) are examples of entities.

A *relationship* is a conceptual association between entities. For example, John Smith and “Computers for Everybody” are in a “student-attends-course” relationship.

## Part V: Supplemental

Entities are classified into sets according to their common properties. For example, *Person* is an entity set in the Easylearn system, whose elements are the entities John Smith, Steven Walsh, Alice Thompson, and so on. A predicate is associated to each entity set to test whether an entity belongs to the set or not. If you know that an entity (for example, John Smith) belongs to an entity set (for example, *Person*), then you know that the entity has the properties common to all entities of that set.

Similarly, relationships are classified into *relationship sets*. For example, all relationships between entities of the *Person* set and entities of the *Course* set that conceptualize the fact that the person attends the course form the *attend* relationship set. Mathematically, an entity set is a set, whereas a relationship set is a relation among entity sets.

A relationship set can be *n*-ary (that is, between *n* entity sets). Each of the participating entity sets plays a certain *role* in the relationship set. The same entity set can play several roles in the relationship set. For example, there can be a *marriage* relationship set between the *Person* entity set and itself, where the same entity set *Person* plays two roles — *husband* and *wife*.

Entities in the same entity set share the same properties, called *attributes* in ER. Attributes are named characteristics of entities. Each individual entity has its own value of each attribute. For example, the name of a person, his or her postal address, and his or her phone number are attributes of the *Person* entity set.

Relationship sets can also have attributes. An attribute maps each relationship from a relationship set to a value.

ER uses a diagrammatic notation for expressing models. In ER diagrams, entity sets are depicted as rectangles,<sup>1</sup> as shown in Figure 23-1. Relationship sets are depicted as diamonds linked with the related entity sets, as shown in the same figure.

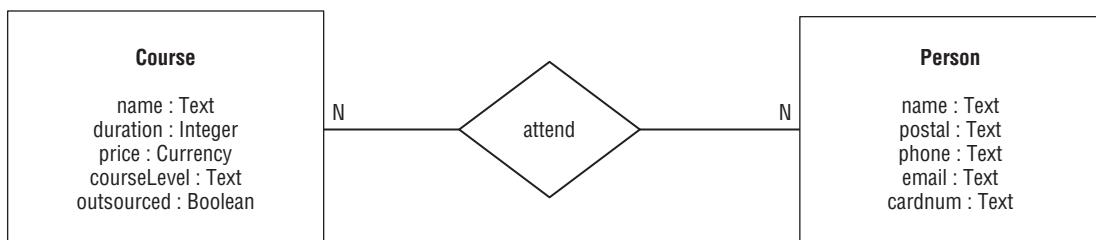


Figure 23-1: An ER model: entity and relationship sets

The *cardinality* of each side (role) of a relationship set denotes how many entities of the entity set at one side may be related to an entity of the entity set on the other side through that relationship. For the given example of a binary relationship (that is, a relationship with two ends), as shown in Figure 23-1, a Person may attend zero to an arbitrary number of Courses, because N denotes “zero to arbitrary many.” Other

<sup>1</sup>Different “dialects” of ER use slightly different notation to show attributes in diagrams. Here, attributes (along with their types) are listed within the rectangle of the entity set. Other notations exist — for example, where an attribute is shown inside an ellipse attached to the entity set rectangle.

specifications of cardinality are also possible, such as 0..1 (zero or one), 1..1 (exactly one), 1..N (one or more), and so on.<sup>2</sup>

### Section Summary

- ❑ An *entity* represents a discrete object or a “thing” that can be distinctly identified in the problem domain. Entities of the same kind are classified into *entity sets*.
- ❑ A *relationship* is a conceptual association between entities. A *relationship set* denotes a set of relationships between entities of the same sets.
- ❑ *Attributes* are named properties of entity or relationship sets. An attribute maps each entity or relationship in a set to a value.
- ❑ The *cardinality* of each side (role) of a relationship set denotes how many entities of the entity set at one side may be related to an entity of the entity set on the other side through that relationship.
- ❑ ER uses a diagrammatic notation, where entities are depicted as rectangles and relationships are shown as diamonds connected with the entity sets by line paths.

## Advanced Concepts

The original version of ER and its later descendants proposed many other concepts that will be briefly mentioned here.

An ER model is a conceptual representation of a problem domain. However, from its very beginning, ER was also concerned with representation of the data conceptualized with an ER model. One mostly used representation of ER models is the relational representation, which is described in the next section.

One concern about relational data representation is identification of entities represented as records. Influenced by the relational representation, ER also introduces the notion of a *primary key*. As in the relational paradigm, a primary key is an attribute or a collection of attributes of an entity set whose values uniquely identify each entity. Unlike the relational paradigm, ER does not require a primary key to be specified at the conceptual level. The primary key must be specified as late as the relational data representation model is defined. If none of the domain attributes from the conceptual ER model can be used as a primary key because of its non-unique nature, a technical ID should be introduced. However, because of its heavy orientation to the relational data representation, the identification of a primary key for an entity set was almost a must. This is why ER conceptual models usually emphasize primary keys. They are underlined in ER diagrams.

Another consequence of the approach where identification of entities at the conceptual level is unnecessarily intermingled with primary key as an implementation-oriented aspect, is the notion of *weak entities*.

<sup>2</sup>Similarly, different notations exist for specifying cardinalities of roles.

## Part V: Supplemental

---

A *weak entity set* is a set of entities that cannot be uniquely identified by its own attributes alone, and, therefore, must use as its primary key a composition of its own attribute(s) and the primary key of an entity it is related to.

For example, in Figure 23-2, a concrete sample of a book in a library is a weak entity, depending on the book title it is a sample of. This means, to fully identify a book sample, you must provide both the identifier (primary key) of the book title, and the identifier of the book sample. In other words, the identifier of a book sample has a scope of its book title and is unique only within that scope.



Figure 23-2: An ER model with a weak entity set

For example, the book title #A can have samples identified as #1, #2, and #3, whereas the book title #B can have its own samples, also identified as #1 and #2. Therefore, to identify a book sample in the global scope, you must say #A#1, and not just #1.

In ER diagrams, a weak entity set is indicated by a bold rectangle connected by a bold arrow to a bold diamond that represents the relationship set with the related entity set that provides the part of the primary key (see Figure 23-2). Double lines can be used instead of bold ones.

A weak entity set means that the entities are *existence-dependent* on the related entities from the master set. In the previous example shown in Figure 23-2, when a book title is removed (deleted) from the database, all its book samples should also be removed.

Apart from the basic, scalar attributes described so far, ER recognizes several other kinds of attributes:

- ❑ **Composite attributes** have values that are records of other composite or scalar values. For example, an address can be a composite of street, city, and state.
- ❑ A **multi-valued attribute** maps an entity to a collection of values. For example, a person can have one or more addresses.
- ❑ A **derived attribute** is an attribute whose value is derived from the values of other attributes. For example, an age attribute is derived from a birth date attribute of a person.

Entity sets do not have to be disjointed. Sometimes one or more entity sets are more specific subtypes (or subsets) of another entity set. For example, in the Easylearn system, the Teacher entity set is a subset, or *specialization* of the Person entity set because each teacher is also a person. In ER diagrams, this is indicated by a triangle with "ISA" written inside, pointing to the more general entity set, as shown in Figure 23-3.

A relationship set and all its participating entity sets can be treated as a single entity set for the purpose of taking part in another relation through *aggregation*, indicated by drawing a dotted rectangle around all aggregated entities and relationships.

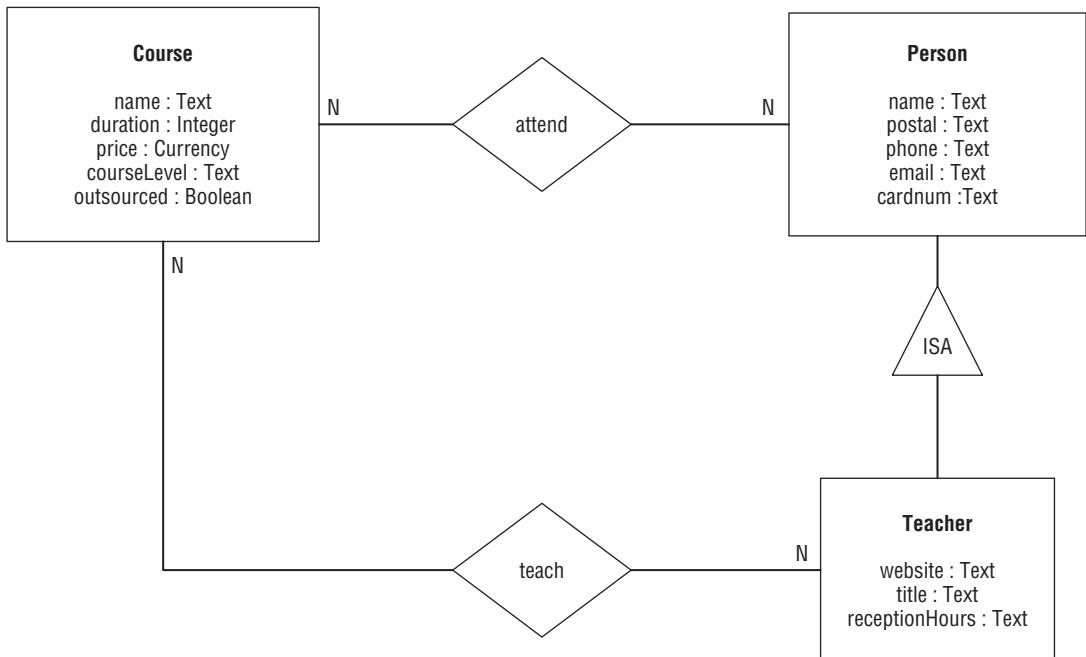


Figure 23-3: An ER model with a specialization

### Section Summary

- ❑ A *primary key* is an attribute or a collection of attributes of an entity set whose values uniquely identify each entity.
- ❑ A *weak entity set* is a set of entities that cannot be uniquely identified by its own attributes alone and, therefore, must use as its primary key a composition of its own attribute(s) and the primary key of an entity it is related to. A weak entity is existence-dependent on its related entity.
- ❑ *Composite attributes* have values that are records of other composite or scalar values.
- ❑ A *multi-valued attribute* maps an entity to a collection of values.
- ❑ A *derived attribute* is an attribute whose value is derived from the values of other attributes.
- ❑ An entity set may be a *specialization* (subset) of another entity set.
- ❑ A relationship set and all its participating entity sets can be treated as a single entity set for the purpose of taking part in another relation through *aggregation*.

### Actions Upon Structure

The very initial proposal of ER [Chen, 1976] defined the set of elementary generic operations (or actions) upon an ER structure and their semantics, without insisting on the concrete syntax of an action language:

- Create an entity of an entity set.
- Create a relation of a relationship set between given entities.
- Insert a new value to a multi-valued attribute of a given entity or a given relationship.
- Change the value of an attribute of a given entity or a given relationship.
- Remove a value from a multi-valued attribute of a given entity or a given relationship.
- Delete (remove) a given entity. As a consequence, delete all existence-dependent entities, all relationships connected to the entity, and all values of its attributes.
- Delete (remove) a given relationship. As a consequence, delete all values of its attributes.

In addition to these data modification actions, ER changes the semantic level of queries, which is now based on the set calculus instead of relational algebra as in the relational paradigm. For example, to retrieve the names of all persons that attend the course “Cooking for Dummies,” according to the ER model in Figure 23-3, the query has the following set semantics:

$$\{ \text{name}(p) \mid p \in \text{Person}, c \in \text{Course}, (p,c) \in \text{attend}, \text{name}(c) = \text{'Cooking for Dummies'} \}$$

Several database query and manipulation languages based on ER, with concrete syntax, have been proposed. One of them is *Entity-Relationship Role-Oriented Language (ERROL)* [Markowitz, 1983]. Although they were implemented and functional, they did not gain wide popularity in practice. Instead, ER has been predominantly used for conceptual data modeling. ER data models are transformed into relational data representations, which are then directly accessed by, for example, SQL.

#### Section Summary

- Generic elementary operations (actions) provide the means to create and remove entities and relationships, modify their attribute values, and retrieve any information from an ER model.
- Although they exist, concrete database query and manipulation languages have not gained wide popularity in practice.

### Mapping to Relational Model

Although very popular for conceptual modeling, the ER paradigm has not been directly supported by widely adopted run-time environments that may interpret ER models. Instead, ER models are, in practice, usually translated into relational models that serve as their implementations. The mapping from an ER

model into the corresponding relational model is generally straightforward, and thus may be performed completely automatically.

Basically, entity sets are mapped into tables. For each entity set in an ER model, a table should be created in the relational database. Attributes of entity sets are generally mapped into fields (columns) of the entity tables. Consequently, entities are represented by the records of the corresponding tables.

Depending on the cardinality of a relationship set, the relationship set's mapping can be done in several ways. Let's consider a relationship set between entity sets *A* and *B*. If the upper bound of the cardinality at the side *B* is a small number *k*, most usually 1, then *k* foreign keys in the table *A* are introduced. These *k* keys will refer to the records in the table *B* representing the related entities. If the lower bound of the cardinality is less than *k*, then these foreign keys can take null values; otherwise, if the cardinality is exactly *k*, the keys cannot be null.

If, however, the upper bound of the cardinality at the side *B* is a relatively large number, so that it is concluded that a large amount of storage space would be wasted for keeping null values in the foreign keys in the table *A*, or the upper bound is unlimited (specified to N), then the opposite side *A* is considered, and the same reasoning is applied to it.

Finally, if the cardinalities at both sides are large numbers or N, then a separate table for the relationship set is introduced, whereby its records will contain foreign keys referring to the related entities. Each record in the relationship table will represent one relationship between entities. The corresponding referential integrity rules (such as cascading update and deletion) are usually introduced for such relationship tables. Clearly, this is the most general case that can be applied to every relationship set, regardless of its cardinalities, including a cardinality of 1. It allows you to change cardinalities of the relationship set without affecting the database scheme.

Finally, specializations can be implemented in a similar way. The table for the special entity set can have a foreign key referring to the table for the general entity set. This way, an entity of the special set is represented by a record in the table for that set, along with the record in the table for the general set, to which it is related.

According to these principles, the relational model obtained from the ER model in Figure 23-3 may be the following:

Table Course  
ID : Integer, Automatically Generated Sequence, Primary Key  
name : Text  
duration : Integer  
price : Currency  
courseLevel : Text  
outsourced : Boolean

Table Person  
ID : Integer, Automatically Generated Sequence, Primary Key  
name : Text  
postal : Text  
phone : Text  
email : Text  
cardnum : Text

## Part V: Supplemental

---

### Table Teacher

```
ID : Integer, Automatically Generated Sequence, Primary Key
website : Text
title : Text
receptionHours : Text
IDPerson : Integer, Foreign Key to Person.ID, Not NULL
```

### Table Attend

```
IDPerson : Integer, Foreign Key to Person.ID, Not NULL
IDCourse : Integer, Foreign Key to Course.ID, Not NULL
```

### Table Teach

```
IDTeacher : Integer, Foreign Key to Teacher.ID, Not NULL
IDCourse : Integer, Foreign Key to Course.ID, Not NULL
```

#### **Section Summary**

- Entity sets are mapped into tables and attributes into their fields.
- Relationship sets may be mapped in different ways, depending on their cardinalities. In a most general case, a table is introduced for each relationship set, having columns for the foreign keys referring to the primary keys of the related entity sets, possibly with enforced referential integrity rules. One record in the table represents one relationship.

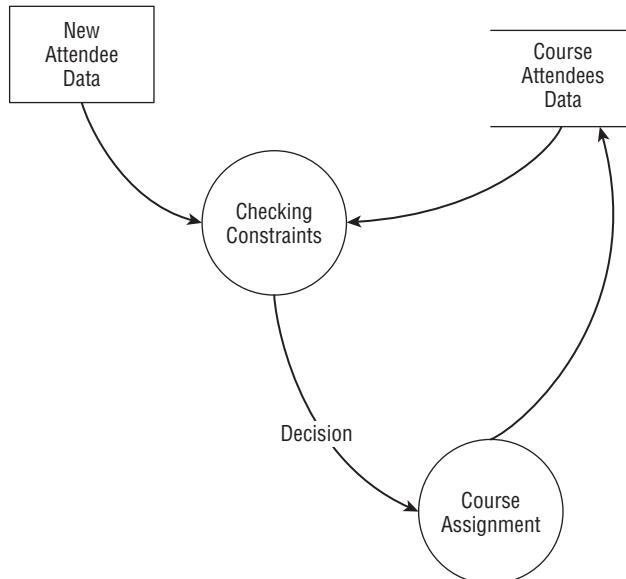
## Data Flow Modeling

The *data flow* (DF) paradigm is used in Structured Analysis to model the functionality of the system. A DF model is used to document the logical flow of data through a set of processes or procedures. For example, Figure 23-4 shows a dataflow model for assigning a new attendant to a course.

A DF model usually includes external sources and destinations of data, activities that transform the data, and stores where the data is held. More precisely, the DF paradigm includes the following main concepts:

- A **process** is a composite set of activities that transform some input data to produce some output data. Processes can be further decomposed using DF diagrams of a lower level. This way, a hierarchical procedural decomposition is undertaken until the processes reach the level of details supported by the implementation programming language. In that case, a leaf process is implemented by an implementation language *procedure*. Processes are depicted as circles in DF diagrams.

- ❑ A **data store** represents a file or a part of the database where the data is held. It represents a persistent source or destination of a data flow. It is denoted with two horizontal parallel lines in the diagrams. The notation also allows specifications of the structure of the data store, but it is out of scope of this discussion.
- ❑ An **external interactor** represents a source or destination of a data flow outside the scope of the presented diagram, which usually represents an enclosing process. It is depicted as a rectangle in the diagram.



**Figure 23-4: A DF model for assigning a new attendant to a course**

- ❑ A **data flow** represents a flow of a structured package of data from a source to a destination. The source and destination can be the described elements (process, data store, or external interactor). It is denoted with an arrow from the source to the destination element.

In summary, a DF diagram is a graph whereby the nodes represent either processes that transform data, data stores that store persistent data, or external inputs and outputs. The edges of the graph represent data flow between the nodes. The semantics of the data flow are intuitively clear. Data packages flow between the nodes and are transformed, generated, stored, or interchanged with the environment in the nodes. A process is activated when all the input data is supplied over the incoming data flow edges. When a process is running, it may supply the data to its outgoing data flow edges. Thus, a DF diagram defines a partial ordering of the execution of processes, which may generally be executed concurrently, unless constrained by implicit ordering implied by a data flow edge.

### Section Summary

- ❑ The DF paradigm is used to model functionality. A DF diagram is a graph whose edges represent data flow between the nodes that can be the following:
  - ❑ Processes, as composite activities that transform input data into output data
  - ❑ Data stores, as pieces of persistent storage of data
  - ❑ External interactors, as inputs from or outputs to the enclosing context of the diagram
- ❑ A DF diagram specifies a partial ordering of process execution. Processes may execute concurrently, unless constrained by implicit ordering implied by data flow edges.
- ❑ A DF diagram usually represents a refinement of an enclosing process. A process in a diagram can be further decomposed by another data flow diagram. This is how hierarchical functional (procedural) decomposition is accomplished.

# 24

## Introduction to the Object Paradigm

This chapter summarizes some fundamental object-oriented (OO) concepts and principles. It also describes the basic concepts supported in some popular OO programming languages.

### Fundamental Object-Oriented Concepts and Principles

The object paradigm is a result of a software technology *evolution*, not *revolution*. It is built upon the concepts and principles from the procedural and other preceding programming paradigms that had been recognized as successful in software engineering practice of building reliable and flexible programs. Those concepts were implicitly recognized and utilized by earlier procedural programming experience, but were often not explicitly and directly supported by the procedural paradigm and languages. On the other hand, they have been explicitly promoted to fundamental object-oriented (OO) concepts and principles, and directly supported by OO programming and modeling languages and methods. The following are the most important concepts and principles described in this section:

- Abstraction (including abstract data types)
- Encapsulation
- Inheritance
- Polymorphism
- OO decomposition

This chapter provides only a brief overview of the concepts.

### **Abstraction and Abstract Data Types**

*Abstraction* is the act of mentally discarding individual aspects of some phenomenon until what remains is small enough to be comprehended [Selic, 1994]. In other words, abstraction refers to the process or a result of the process of reducing the information contents until what remains is relevant for the particular purpose [Wiki]. Namely, the prevailing problem of real-world domains to be modeled by software systems is their complexity, which is not expressed through the quantity of elements existing in a domain, but through their diversity.

One important and effective form of abstraction is the process of classifying elements that share some commonalities into sets, capturing the diversity on a coarser level, among the sets, instead of among individual elements. In other words, it is a conceptual technique where differences among individual elements of some phenomenon are neglected for the sake of highlighting their commonalities. It is, along with decomposition, one of the basic and most powerful software engineering techniques for coping with complexity. It is a general principle, not specific to any problem domain or paradigm. It may be applied at different levels and with different scopes, as it is described throughout this book. Although abstraction is an old principle, and by no means specific to the object paradigm, it is mentioned here because the object paradigm has clearly identified it and promoted it to one of its fundamental principles.

One of the manifestations of abstraction is the notion of *abstract data type*. Abstract data type is a generalization of the concept of type in programming languages. The classical notion of *type* in programming languages refers to a (usually static) specification of a set of (usually dynamic) instances created at runtime. Purely conceptually, the notion of type includes a data structure and a set of services that can be required from instances of that type. The coherent union of the structure and services (or operations) is crucial for the notion of type, and none of them can be treated separately. They co-exist as a whole and cooperate synergistically. For example, an integer type built in a certain programming language on a certain platform does not merely assume a set of, for example, “32-bit integer values in the 2’s complement,” but also the set of operations upon that representation defined in the language, such as addition, subtraction, multiplication, division, and so on. Although classical types in older programming languages did not explicitly package these two aspects, this conceptual union and its importance were clearly recognized later.

Most programming languages have a set of predefined, built-in types. However, it was recognized long ago that there was no way to define a complete set of built-in types of a programming language that would satisfy all needs of problem domains. That is why many languages (even procedural ones) allow programmers to define their own types, called *user-defined* types, as compositions of data structures and operations. The data structures are defined in terms of other elements of built-in or other user-defined types, and the operations are specified using the action part of the programming language.

This is how the notion of type has been generalized into a manifestation of abstraction. Namely, built-in or user-defined abstract data types often denote some abstract things from the problem domain modeled by the program, or abstractions from the solution. This is one motivation for including ‘abstract’ in the term ‘abstract data type.’ For example, the type that represents a point in time during the day (from 00:00:00 to 23:59:59), with the operations such as incrementing the number of seconds or to checking whether one point in time is before or after another, may be represented by an abstract data type. It may be built in a programming language, or it can be defined by the programmer.

Although the data structure of an abstract data type is its important part, what is much more important for its users is the set of available operations upon the instances of the type. In other words, the essential idea behind abstract data types is that the internal data representation and the implementation of

operations are not important for the rest of the system that uses an abstract data type. What is important is its externally visible behavior — that is, the set of operations and their semantics. The purpose of this principle is to avoid some common programming errors. This observation is yet another manifestation of abstraction, and, thus, another reason for the word “abstract data type” in the name.

For example, a stack may be regarded as an abstract data type. It has operations such as “push” (put an element on the top of the stack) and “pop” (fetch an element from the top of the stack), which have well-known semantics. On the other hand, the internal representation of its data structure (that is, the physical storage mechanism, the layout of the elements of the stack in memory, and their linkage) is not important for users of the stack. Users (that is, the other parts of the program that operate on instances of the stack abstract data type) need to know only about the outwardly visible behavior of the stack — that is, for example, that a “pop” following a “push” will return the same pushed element, and so on.

In summary, an abstract data type is defined as:

- ❑ A type, meaning that it specifies a set of instances that exist during the execution of the program
- ❑ A coherent union of data and operations that work upon the data
- ❑ A unit of software organization that has its internal implementation, comprised of the data structure and implementation of operations, and its interface, comprised of the operation signatures (names and arguments of operations) and their outwardly visible semantics
- ❑ A representation of an abstraction from the problem domain or the program solution

The object paradigm has adopted the notion of abstract data types as its first-class concept. In most object programming languages, a *class* defines an abstract data type in the program. Instances of a class are called *objects*. Consequently, from one perspective, objects may be treated as instances of abstract data types.

### Section Summary

- ❑ Abstraction is the act of mentally discarding individual aspects of some phenomenon until what remains is small enough to be comprehended.
- ❑ An abstract data type is defined as:
  - ❑ A type, meaning that it specifies a set of instances that exist during the execution of the program
  - ❑ A coherent union of data and operations that work upon the data
  - ❑ A unit of software organization that has its internal implementation on one side, comprised of the data structure and implementation of operations, and its interface on the other side, comprised of the operation signatures (names and arguments of operations) and their outwardly visible semantics
  - ❑ A representation of an abstraction from the problem domain or the program solution

### Encapsulation

As a result of decomposition, software is divided into parts, the meaning of which depends on the paradigm and decomposition. The parts may be modules, abstract data types, packages, or other concepts recognized by the paradigm or the language. In any case, the principle of abstraction requires that each part has its interface and its implementation clearly separated from each other. *Encapsulation* is the principle of exposing only the interface and hiding the implementation of a part of software. In other words, the environment of a part cannot access the implementation, but only its interface (see Figure 24-1).<sup>1</sup>

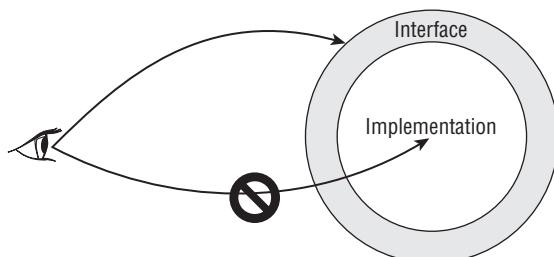


Figure 24-1: Encapsulation

The *interface* of a part is specified as something that the part offers to its environment as a contract, or the obligation that the part promises to fulfill at any time. The *implementation*, however, is something that is private to the part, and that is assumed to be modifiable without affecting external parts. If another external part of the software relies only on the interface of the part, the external part remains stable if the implementation of the latter part changes, because the contract specified by the interface must still be satisfied. Let us imagine that a programmer changes the implementation of the part, assuming that other external parts do not rely on the assumptions posed in that implementation. If an external part violates the encapsulation by relying on the assumptions made in the implementation, that external part will fail after the modification in the former part. This is one of the most frequent causes of errors in traditional software approaches that do not enforce encapsulation.

Encapsulation can be achieved through discipline, by obeying coding rules posed by programmers themselves. However, this approach is highly error-prone. It is crucial, therefore, that the language formally define the rules of encapsulation, and that the modeling tool or compiler check possible violations, in order to avoid errors.

In short, encapsulation is yet another concrete technique that supports the general principle of loose coupling of software parts. It ensures that software parts cannot arbitrarily make interconnections other than as allowed by their contracts. Consequently, it is again a general software engineering principle, not strictly associated with object orientation. However, as with abstraction, the object paradigm has

---

<sup>1</sup>To be precise, *encapsulation* can be treated as separate from *information hiding*. The former is merely the act of aggregating a set of things and dealing with them as a unit. Such a unit may or may not involve information hiding, which means hiding the details of the unit's implementation and exposing its interface only. However, the first one is of much less practical value if not combined with the latter. This is why most popular programming and modeling languages combine these, too, and also why the term “encapsulation” in this book refers to the combination of both aspects.

recognized it as one of its fundamental concepts supported by almost all OO programming and modeling languages and methods. It can be supported at very different levels and by different formalisms. That is why different OO approaches treat it in very different ways, but they all treat it as an important issue.

### Section Summary

- Encapsulation is the principle that an outwardly visible and accessible interface and a hidden, inaccessible implementation can be clearly identified for a software component.

## Inheritance and Subtyping

Sometimes, two abstract data types,  $A$  and  $B$ , may coexist in a relationship that can be expressed with the fact that “ $B$  has the same set of features (that is, the data structure and services of  $A$ , and whatever constitutes the interface and implementation of the abstract type), and possibly has some more features.” In other words, the set of features of  $B$  is a superset of the features of  $A$ . For the example of the school system introduced in Chapter 22, teachers have properties of all persons involved in the system (such as name, postal address, and so on), but they also have some specific properties (such as title or Web address). Instead of defining unrelated types, the object paradigm supports the notion of *inheritance* as a conceptual relationship between abstractions that can be expressed right with the given statements. If a type  $B$  inherits a type  $A$ , it implicitly possesses all features of  $A$  (the inherited features), but may also explicitly add some more features. For the given example, the Teacher abstract type inherits the Person abstract type.

An obvious consequence of the concept of inheritance is reduction of redundancies in a program or model. To define a new type that inherits the features of an existing type, it is not necessary to copy all the features. They are implicitly included in the new type simply by designating it as a descendant of the existing type. Moreover, if a feature of the ancestor type is modified, the modification is automatically propagated to the descendant type. This way, inheritance is applied for the sake of reuse.

However, propagation of features between abstract data types is just one possible aspect of inheritance, although the only one supported by some languages. Another important manifestation is *subtyping*. In such an approach, the descendant type is *a kind of* the ancestor type. This interpretation is fully reasonable, provided that the descendant type inherits *all* features from its ancestor and cannot remove any of them. In that case, everything that can be requested from an instance of the ancestor type can be also requested from an instance of the descendant type. This is why such an approach supports the *substitution rule*: Whenever and wherever an instance of an ancestor type is expected, an instance of the descendant type may be used, without any externally visible difference. In fact, an instance of the descendant type is also an indirect instance of the ancestor type.

For the given example, the Teacher type can be regarded as a subtype of the Person type, because teachers have all features of persons. This is why everything that holds for persons also holds for teachers. For example, teachers may also attend courses as all (other) persons.

Some languages that support only inheritance (without tying it with subtyping) allow the inheriting type to remove some features from its ancestor. In such cases, inheritance can be used for simple reuse of

## Part V: Supplemental

---

implementation, as opposed to subtyping. Most widespread languages, however, do not allow you to remove inherited features in the inheriting type, because they tie the concept of inheritance to subtyping, as explained here. If a type inherits another type, but removes some of its outwardly visible features (that is, part of its interface), then it will obviously not represent its subtype.

### Section Summary

- ❑ Inheritance is a conceptual relationship between two abstract data types that can be expressed with the fact that the descendant type implicitly possesses all the features (structural and behavioral) of the ancestor type, unless they are explicitly excluded.
- ❑ Inheritance is often (but not always) accompanied by subtyping, whereby the descendant type is also a subtype of the ancestor type. This means that instances of the subtype are also instances of the ancestor type.

## Polymorphism

An operation of an abstract data type is only a specification of a service that can be requested from its instances [Booch, 1999]. The concrete behavior activated on such a request depends on the implementation of the operation, which can be regarded as a procedure invoked on a request of the service. The object paradigm offers the possibility that different types provide different implementations of the same operation. Instances of those types will then react on the invocation of that same operation differently, according to the implementations provided by their types. This concept is called *polymorphism*.

One particular and important case of this concept is that a descendant type can *override* an operation from the ancestor type by providing an implementation that is different from the implementation of the same operation in the ancestor type.<sup>2</sup> This concept is particularly useful when a user of an abstract data type works with instances that it treats as instances of the ancestor type, although they may actually be instances of the descendant type because of the substitution rule. The user may request a service from such instances by invoking their operation, which can be overridden in the descendant type. If an instance is of the descendant type, the overridden behavior will be activated. This way, the user is not aware of the specialties of the subtypes in an inheritance hierarchy. The user works with only the general, abstract services of the ancestor type, and is, therefore, independent of the different cases of reaction. The appropriate, special behavior is activated when a general service is invoked; this is called *polymorphism*.

For example, the Person abstract data type may have a service of calculating the discount rate for the person, depending on the number of courses the person attends. The calculation procedure may be overridden in the Teacher type, for example, because the teachers of the school may have greater discounts than other students. The concept of polymorphism allows us to treat all persons and teachers in the same

---

<sup>2</sup>In general, polymorphism does not necessarily imply the use of inheritance and subtyping. Polymorphism simply means that a given operation may be implemented differently by different types (whether or not they are in a subtype or inheritance relationship). However, in most popular programming languages, polymorphism is tightly related to inheritance and subtyping, as described in this text.

way, as the instances from which the service of calculating the discount can be requested. The instances will respond according to their types, although the requests are the same.

The benefit of this approach is tremendous. Clients (that is, other parts of software) of a type inheritance hierarchy may become more independent of the hierarchy. If a new subtype is added to the inheritance hierarchy, for example, clients do not need to be modified if they access the instances as generalized items. The newly added subtype may inherit the implementation of an operation from its ancestors, in which case its instances behave in the same way. On the other hand, the newly added subtype can override the operation, in which case its instances behave differently, although the rest of the software has not been modified. This is how inheritance and polymorphism allow the behavior of software to be modified by *adding* new parts of the software (new subtypes and new implementations of existing operations), and not by *modifying* the already existing parts.

Consequently, inheritance, in combination with subtyping and polymorphism, is yet another manifestation of abstraction. However, they are all among the most powerful concepts of object orientation. They notably distinguish the object paradigm from its procedural predecessor. Their power is again in abstracting common things from some abstract data types, and withdrawing the specialties in subtypes and polymorphic operations. In this way, the tendency of OO modeling and programming is that clients of a certain type hierarchy, defined by the inheritance relationships, rely as much as possible on the generalized types and do not depend on the specialties of the descendants. As explained, the interactions between software parts become weaker and better controllable then.

### Section Summary

- ❑ An operation is a specification of a service that can be requested from instances of an abstraction. The implementation of an operation provides the behavior manifested upon such a request.
- ❑ Polymorphism is the concept that allows different types to provide different implementations for the same operation. Instances of those types will then react to an invocation of that same operation differently, according to the implementations provided by their types.
- ❑ A special (and most important) case of polymorphism occurs when a subtype overrides an operation by providing a different implementation of an inherited operation.
- ❑ Polymorphism ensures that the behavior appropriate for the type of an instance is activated even though the instance is accessed as a general item (that is, as an instance of the ancestor type).

## Object-Oriented Decomposition

Instead of almost exclusively (or at least predominantly) using procedural (algorithmic) decomposition, the object paradigm uses many different methods of decomposition. As with all other OO concepts, the methods of decomposition are not revolutionary approaches that replace all traditional principles. Instead, OO decomposition preserves the principles from the predecessor paradigms that have been recognized as good, promotes some of them, and introduces new ones.

## Part V: Supplemental

---

It is still difficult to enlist all heuristics, methods, and techniques that are considered parts of OO decomposition. It is also very likely that the understanding of what OO decomposition is will further evolve in the future. This is why this discussion briefly describes just some of the major aspects that are central to OO decomposition.

### **Distribution of Responsibilities**

One of the central goals in the design of OO software is a proper distribution of responsibilities among abstractions (that is, classes). There is no cookbook recipe for accomplishing this task. Instead, a lot of heuristics are well-described in a number of books on object-oriented analysis and design. Among the most important is that a class should have a coherent set of responsibilities, loosely coupled to the rest of the system. In other words, one aspect (be it a piece of information or a responsibility for a certain task) should be kept in one place and not spread out or replicated. On the other hand, it is not wise to overload a class with too many unrelated or weakly related responsibilities, because that leads to a cluttered design and inflexible software.

### **Design of Inheritance Hierarchies**

Another important aspect of OO decomposition is the design of inheritance/subtyping hierarchies. Again, there are only suggestions, but not rules concerning how to organize hierarchies. It is well-known that hierarchies that are too large (either too deep or too wide) are not practical because they are difficult to understand and maintain. Moreover, each inheritance relationship should have a strong rationale behind it. Most often, inheritance relationships are introduced to generalize some concepts for the sake of clients who can access the general entities without dealing with specializations.

### **Identification and Design of Structure**

One of the tasks of OO decomposition is the design of structure, which encompasses identification of relationships among conceptual things, and definition of the topology of instances that exist at run-time.

### **Identification and Design of Collaborations**

Another important activity during OO decomposition is identification and design of interactions among instances at run-time. From the OO perspective, objects (as instances of classes) interact during the execution of the system by passing messages or invoking operations of each other. This collaboration of objects takes place in the form of scenarios of interactions. Therefore, objects may be regarded as logical machines that encapsulate a certain structure (defined by their structural features — that is, data) and provide certain behavior (specified by their behavioral features — that is, operations). The internal state of an object may be changed as a side effect of activation of its behavioral features.

### **Procedural Decomposition**

Procedural (algorithmic) decomposition has the same meaning as in the procedural paradigm. However, its scope and role have been changed. First, algorithmic decomposition is no longer the sole decomposition applied during development. It is just part of the entire set of methods and techniques commonly identified as OO decomposition. Consequently, its importance is reduced. Finally, its scope is different. It is now applied when designing interactions among objects in order to provide a certain complex behavior of the system, be it an outwardly visible function of the system, or simply the implementation of an operation of a class.

In summary, OO decomposition is a complex set of activities that encompasses different heuristics and principles, instead of strict rules and recipes. The accumulated practical experience may help in applying

OO decomposition. For example, design patterns play a major role in recognizing directions for OO decomposition.

### Section Summary

- ❑ Object-oriented decomposition is a complex set of activities that encompasses different heuristics and principles, instead of strict rules and recipes. Following are main elements:
  - ❑ Distribution of responsibilities
  - ❑ Design of inheritance hierarchies
  - ❑ Identification and design of structure
  - ❑ Identification and design of collaborations
  - ❑ Procedural decomposition.

# Object-Oriented Programming

OO programming emerged as an evolutionary programming approach that exploits the described OO concepts and principles. In spite of the vast diversity of OO programming languages and their concepts, almost full agreement has been achieved nowadays on what OO programming essentially means. It is an approach to developing software at a lower level of abstraction (compared to higher-level approaches referred to as *OO modeling* and *OO analysis and design*), using textual programming languages. Although there are many “pure” OO languages, the most popular ones have been developed as successors of already existing procedural languages (such as C++ from C) or other OO languages (such as Java following the experience of C++, and C#). This discussion briefly examines just the main elements of these most popular OO programming languages.

## Abstract Data Types and Instances

The concept of abstract data type is supported by the *class* construct in an OO programming language. For example, the Person abstract data type can be implemented in C++ like this (the equivalent Java or C# implementation has just some insignificant syntactical differences):

```
class Person {
public:
 virtual void sendEMail (String message);

private:
 String name;
 String postal;
 String phone;
 String email;
 String cardnum;
};
```

## Part V: Supplemental

---

The class specifies the set of *data members* (also called *fields* or *properties* in the terminology of different OO programming languages), such as `name`, `postal`, and so on, that constitute the structural part of the class, and *member functions* (also called *operations* or *methods* in the terminology of different OO programming languages), such as `sendEMail`, that specify the behavioral part of the class (that is, the set of services that can be requested from the instances of the class). Objects, as instances of classes, are created, destroyed, accessed, and modified at run-time, when the behavioral part of the program is executed. The behavioral part of the program is specified using the action part of the programming language, as described later.

### **Encapsulation**

Basically, encapsulation is supported in C++, Java, and C# at the class level. Members (both data and functions) may be designated either as *public*, meaning that they are accessible from outside the class, or as *private*, meaning that they are accessible from within the class only, or as *protected*, meaning that they are accessible from within the same class, as well as within the descendant classes.

In the previous example, the data members are declared in the private part of the class, and are, therefore, inaccessible from outside the class. The operation `sendEMail` is public and is part of the interface of the class. The compiler controls access rights automatically, and reports violations as compile-time errors. Java also supports a coarser level of encapsulation with the notion of a *package*, which is a set of classes, some of which may be hidden from clients of the package. A similar approach is supported in C# for units of deployment called *assemblies*.

### **Object References**

All three languages support a special built-in data type that represents a reference to an object of a class. (C++ supports two kinds of references, called *pointers* and *references*, but their difference is not significant for this discussion.)

A reference is a handle of an object, its identification in the memory space. Basically, it is implemented as a pointer containing the address of the object in memory. Using a reference, one object can establish a unidirectional link to another object. A set of links is established using a container of references. For example, an object of the class `Course` may have a set of references to the persons that attend the course. In Java, the implementation may be the following:

```
class Course {
 ...
 public Person[] students;
 ...
}
```

The `students` property is a dynamic vector of references to persons.

Such references are used as the medium by which objects can interact with one another. For example, an object that embodies a reference to another object can access the latter object's data fields or invoke its operation, usually as part of a more complex scenario of object interaction.

Note that references in OO programming languages represent unidirectional structural links between objects.

### Inheritance

Inheritance is directly supported in OO programming languages. For example, in order to specify that the type `Teacher` is a descendant of the type `Person`, you can simply declare in Java (and similarly in C#, while the C++ syntax is just slightly more complex because of some subtleties not important to the example here):

```
class Teacher extends Person {
 ...
}
```

In all these languages (except for a subtle exception in C++ that is of less importance), inheritance implies subtyping. In this example, teachers are also kinds of persons. This includes the substitution rule, too. For the given example, the `students` references in an object of the class `Course` may refer to instances of (all) subtypes of `Person`, and not only to instances of strictly that class. In other words, the unidirectional links from an object of `Course` may be established toward teachers, too, without any special modification of the class `Course`.

### Action Language

The behavioral part of a program in a classical OO programming language is exclusively concentrated in operation bodies. Operation bodies are specified using programming constructs almost completely taken from procedural languages. For example, loops, conditional statements, expressions, and so on in these languages are principally the same as in C. The only OO elements are actions upon objects. There are actually only three essential kinds of actions upon objects.

The first is *object creation*. In Java and C#, an object is created using the operator `new` (C++ supports the same operator, although objects in C++ can be created in several other ways, depending on their lifetime category), as shown here:

```
Person aPersonRef = new Person();
```

The second operation upon objects is *object destruction*. In C++, objects (with dynamic lifetime only — that is, those created with `new`) are explicitly destroyed using the `delete` operator. In Java and C#, an object is implicitly destroyed when the last reference linked to it dies.

Finally, the third operation is the *access to an object's member* (data member or operation). For example, an operation of an object in Java and C# can be invoked with the following:

```
aPersonRef.sendEMail("Hello!");
```

Note that a reference may refer to an object of a descendant class. In the case where the descendant class has overridden the operation, the operation invocation is polymorphic. For example, the following code will invoke the overridden implementation of `sendEMail`, if such is provided in the class `Teacher`:

```
Person aPersonRef = new Teacher();
...
aPersonRef.sendEMail("Hello!");
```

### Section Summary

- ❑ OO programming is an approach to developing software at a lower level of abstraction (compared to higher-level approaches such as OO modeling and OO analysis and design), using core OO concepts and principles, and in textual programming languages.
- ❑ Most popular OO programming languages (such as C++, Java, or C#) directly support all fundamental OO concepts and principles.

# References and Bibliography

- [Arnold, 1996] Arnold, K., Gosling, J., *The Java Programming Language*, Addison-Wesley, 1996
- [Arnout, 2003] Arnout, K., Meyer, B., "Uncovering Hidden Contracts: The .Net Example," *IEEE Computer*, Vol. 36, No. 11, November 2003, pp. 48–55
- [Barbier, 2003] Barbier, F., Henderson-Sellers, B., Le Parc-Lacayrelle, A., Bruel, J.-M., "Formalization of the Whole-Part Relationship in the Unified Modeling Language," *IEEE Trans. Software Engineering*, Vol. 29, No. 5, May 2003, pp. 459–470
- [Bjorkander, 2003] Björkander, M., Kobryn, C., "Architecturing Systems with UML 2.0," *IEEE Software*, Vol. 20, No. 4, July/August 2003, pp. 57–61
- [Blaha, 1998] Blaha, M., Premerlani, W., *Object-Oriented Modeling and Design for Database Applications*, Prentice-Hall, 1998
- [Blaha, 2002] Blaha, M., Smith, C., "A Pattern for Softcoded Values," *IEEE Computer*, Vol. 35, No. 5, May 2002, pp. 28–34
- [Blaha, 2004] Blaha, M., "A Copper Bullet for Software Quality Improvement," *IEEE Computer*, Vol. 37, No. 2, February 2004, pp. 21–25
- [Boehm, 1988] Boehm, B. W., "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, Vol. 21, No. 5, May 1988, pp. 61–72
- [Booch, 1994] Booche, G., *Object-Oriented Analysis and Design with Applications*, 2nd ed., Benjamin/Cummings, Redwood City, Calif., 1994
- [Booch, 1999] Booche, G., Rumbaugh, J., Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley, 1999
- [Bourdeau, 1995] Bourdeau, R. H., Cheng, B. H. C., "A Formal Semantics for Object Model Diagrams," *IEEE Trans. Software Engineering*, Vol. 21, No. 10, October 1995, pp. 799–821
- [Chen, 1976] Chen, P. P., "The Entity-Relationship Model," *ACM Trans. Database Systems*, Vol. 1, No. 1, 1976, pp. 9–36
- [Cockburn, 1997a] Cockburn, A., "Structuring Use Cases with Goals" (Part 1), *J. Object-Oriented Programming*, September/October 1997, pp. 35–40
- [Cockburn, 1997b] Cockburn, A., "Structuring Use Cases with Goals" (Part 2), *J. Object-Oriented Programming*, November/December 1997, pp. 56–62
- [Cockburn, 2000] Cockburn, A., *Writing Effective Use Cases*, Addison-Wesley, 2000
- [Codd, 1970] Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, Vol. 13, No. 6, 1970, pp. 377–387
- [Constantine, 1979] Constantine, L. L., Yourdon, E., *Structured Design*, Prentice Hall, 1979.
- [Cypher, 1993] Cypher, A., *Watch What I Do: Programming by Demonstration*, MIT Press, Cambridge, Mass., 1993

## References and Bibliography

---

- [Czejdo, 1990] Czejdo, B., Elmasri, R., Rusinkiewicz, M., Embley, D. W., "A Graphical Data Manipulation Language for an Extended Entity-Relationship Model," *IEEE Computer*, Vol. 23, No. 3, March 1990, pp. 26–36
- [Da Silva, 2003] Da Silva, P. P., Paton, N. W., "User Interface Modeling in UMLi," *IEEE Software*, Vol. 20, No. 4, July/August 2003, pp. 62–69
- [DeMarco, 1978] DeMarco, T., *Structured Analysis and System Specification*, Yourdon Press, 1978
- [EUF] Object Management Group, *Executable UML Foundation*, work in progress at the time of retrieval (February 2009), [www.omg.org](http://www.omg.org)
- [France, 1999] France, R. B., "A Problem-Oriented Analysis of Basic UML Static Requirements Modeling Concepts," *Proc. 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, ACM SIGPLAN Notices, Vol. 34, No. 10, November 1999, pp. 57–69
- [Hunt, 2004] Hunt, A., Thomas, D., "Keep it DRY, Shy, and Tell the Other Guy," *IEEE Software*, Vol. 21, No. 3, May/June 2004, pp. 101–103
- [Gamma, 1995] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns*, Addison-Wesley, 1995
- [Genova, 2002] Génova, G., Llorens, J., Martínez, P., "The Meaning of Multiplicity of Nary Associations in UML," *Software and Systems Modeling*, Vol. 1, No. 2, February 2002, pp. 86–97
- [Genova, 2004] Génova, G., Llorens, J., Fuentes, J. M., "UML Associations: A Structural and Contextual View," *J. Object Technology*, Vol. 3, No. 7, July-August 2004, pp. 83–100
- [Harel, 1987] Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming* 8, 1987, pp. 231–274
- [Harel, 2001] Harel, D., "From Play-In Scenarios to Code: An Achievable Dream," *IEEE Computer*, Vol. 34, No. 1, January 2001, pp. 53–60
- [Harel, 2003] Harel, D., Marely, R., "Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach," *Software and Systems Modeling*, Vol. 2, No. 2, July 2003, pp. 82–107
- [Harel, 2004] Harel, D., Rumpe, B., "Meaningful Modeling: What's the Semantics of 'Semantics'?", *IEEE Computer*, Vol. 37, No. 10, October 2004, pp. 64–71
- [Jacobson, 1992] Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G., *Object-Oriented Software Engineering*, Addison-Wesley, 1992
- [Koskimies, 1998] Koskimies, K., Systa, T., Tuomi, J., Mannisto, T., "Automated Support for Modeling OO Software," *IEEE Software*, Vol. 15, No. 1, Jan./Feb. 1998, pp. 87–94
- [Lauesen, 2001] Lauesen, S., Harning, M. B., "Virtual Windows: Linking User Tasks, Data Models, and Interface Design," *IEEE Software*, Vol. 18, No. 4, July/August 2001, pp. 67–75
- [Lauesen, 2002] Lauesen, S., *Software Requirements: Styles and Techniques*, Addison-Wesley, 2002
- [Lauesen, 2003] Lauesen, S., "Task Descriptions as Functional Requirements," *IEEE Software*, Vol. 20, No. 2, March/April 2003, pp. 58–65
- [Lieberman, 2000] Lieberman, H., "Programming by Example," *Communications of the ACM*, Vol. 43, No. 3, March 2000, pp. 73–74
- [Malhotra, 1990] Malhotra, A., Markowitz, H. M., Tsalalikhin, Y., Paziel, D. P., Burns, L. M., "An Entity-Relationship Programming Language," *IEEE Trans. Software Engineering*, Vol. 15, No. 9, September 1989, pp. 1120–1130
- [Markowitz, 1983] Markowitz, V. M., Raz, Y., "ERROL - An Entity Relationship Role Oriented Query Language," in *Entity Relationship Approach to Software Engineering*, Davis, G. C. et al. (eds.), North-Holland, October 1983, pp. 329–345
- [Mellor, 2002] Mellor, S. J., Balcer, M., *Executable UML: A Foundation for Model-Driven Architecture*, Addison-Wesley, 2002
- [Mellor, 2003] Mellor, S. J., Clark, A. N., Futagami, T., "Model-Driven Development," *IEEE Software*, Vol. 20, No. 5, September/October 2003, pp. 14–18
- [Meyer, 1992] Meyer, B., "Applying Design by Contracts," *IEEE Computer*, Vol. 25, No. 10, October 1992, pp. 40–51

- [Meyer, 1997] Meyer, B., *Object-Oriented Software Construction*, 2<sup>nd</sup> ed., Prentice Hall, 1997
- [Milicev, 2002a] Milićev, D., "Domain Mapping Using Extended UML Object Diagrams," *IEEE Software*, Vol. 19, No. 2, March/April 2002, pp. 90–97
- [Milicev, 2002b] Milićev, D., "Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments," *IEEE Trans. Software Engineering*, Vol. 28, No. 4, April 2002, pp. 413–431
- [Milicev, 2007] Milićev, D., "On the Semantics of Associations and Association Ends in UML," *IEEE Trans. Software Engineering*, Vol. 33, No. 4, April 2007, pp. 238–251
- [Myers, 1992] Myers, B. A., "Demonstrational Interfaces: A Step Beyond Direct Manipulation," *IEEE Computer*, Vol. 25, No. 8, August 1992, pp. 61–73
- [OCL2] Object Management Group, *OCL 2.0 Specification*, V2.0, ptc/2005-06-06, [www.omg.org](http://www.omg.org), June 2005
- [ODTWG] Object Management Group, "Next-Generation Object Database Standardization," mars/2007-09-13, Object Database Technology Working Group White Paper, [www.omg.org](http://www.omg.org), September 2007
- [Øvergaard, 1998] Øvergaard, G., "A Formal Approach to Relationships in the Unified Modeling Language," *Proc. PSMT'98 Workshop on Precise Semantics for Modeling Techniques*, Technische Universität München, TUM-I9803, 1998
- [Øvergaard, 2000] Øvergaard, G., *Formal Specification of Object-Oriented Modelling Concepts*, PhD Thesis, Department of Teleinformatics, Royal Institute of Technology, Stockholm, Sweden, November 2000
- [Riehle, 2001] Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N., "The Architecture of a UML Virtual Machine," *Proc. 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA '01), 2001
- [Rumbaugh, 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., *Object-Oriented Modeling and Design*, Prentice-Hall, 1991
- [Selic, 1994] Selic, B., Gullekson, G., Ward, P.T., *Real-Time Object-Oriented Modeling*, John Wiley and Sons, 1994
- [Selic, 2002] Selic, B., Ramackers, G., Kobryn, C., "Evolution, Not Revolution," *Communications of the ACM*, Vol. 45, No. 11, November 2002, pp. 70–72
- [Selic, 2003] Selic, B., "The Pragmatics of Model-Driven Development," *IEEE Software*, Vol. 20, No. 5, September/October 2003, pp. 19–25
- [Selic, 2004] Selic, B., "On the Semantic Foundations of Standard UML 2.0," *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT 2004)*, Bertinoro, Italy, September 13–18, 2004, *Springer-Verlag Lecture Notes in Computer Science*, Vol. 3185/2004, pp. 181–199
- [Stevens, 2002] Stevens, P., "On the Interpretation of Binary Associations in the Unified Modeling Language," *Software and Systems Modeling*, Vol. 1, No. 1, January 2002, pp. 68–79
- [UML2] Object Management Group, *UML 2.1.1 Superstructure Specification*, Final Adopted Specification ptc/07-02-05, February 2007 and *UML 2.2 Superstructure Specification*, ptc/2009-02-02
- [Wang, 2004] Wang, Y., Patel, S., "Guest Editorial: On Modeling Object-Oriented Information Systems," *Software and Systems Modeling*, Vo. 3, No. 4, December 2004, pp. 258–261
- [Wegscheider, 1997] Wegscheider, E., "Toward Code-Free Business Application Development," *IEEE Computer*, Vol. 30, No. 3, March 1997, pp. 35–43
- [Wiki] Wikipedia, *The Free Encyclopedia*, [www.wikipedia.org](http://www.wikipedia.org)
- [XMI] Object Management Group, *XMI Specification*, formal/07-12-01, December 2007



# Index

Page numbers for index terms in different font styles have the following meanings:

- **bold**: indicates a place with the main definition and/or explanation of the term;
- *italic*: indicates a place with examples for the term;
- normal: indicates a place of mention.

## SYMBOLS

**# (protected)**, 159  
**##**  
 in OQL, 411  
 in pattern object structure,  
 418  
**% operator**, 458  
**% = operator**, 459  
**& operator**, 458  
**~ (package visibility)**, 159  
**~ operator**, 458  
**&& operator**, 458  
 in OCL, 372  
**& = operator**, 459  
**! operators**, 458  
**!= (not equal operator)**, 189,  
 191  
**!= operator**, 459  
 in OCL, 372  
**> operator**, 175, 459  
 in OCL, 372  
**>> operator**, 458  
**>>> operator**, 458  
**>>>= operator**, 459  
**>>= operator**, 459  
**>= operator**, 459  
 in OCL, 372  
**< operator**, 459  
 in OCL, 372  
**<> operator**  
 in OCL, 372  
**<< operator**, 458  
**<<= operator**, 459  
**<= operator**, 459  
 in OCL, 372  
**|operator**, 458  
 for asynchronous call, 439

**||operator**, 458  
 in OCL, 372  
**| =operator**, 459  
**()**, 449  
 (call) operator, 457  
**\* (asterisk)**, 172  
 in OQL queries, 408  
 in SQL, 716  
**\* operator**, 458  
 in OCL, 372  
**\* = operator**, 459  
**+ (public)**, 159  
**+ operator**, 458  
 in OCL, 372  
**++ operator**, 458  
**+ = operator**, 459  
**, operator**  
 in OCL, 386  
**- (private)**, 159  
**- operator**, 458  
 in OCL, 372, 378  
**-> (arrow notation)**, 264  
**-> (slot feature access  
operator)**, 457  
**-> operator in OCL**, 370,  
 372, 376  
**-- operator**, 458  
 in OCL, 386  
**- = operator**, 459  
**. (dot notation)**, 264  
**. (dot)**, 157  
 for operation calls, 435  
 for query results, 393  
 in OQL queries, 403, 407  
 null target instance, 459  
**. operator in OCL**, 372  
**/ operator**, 458  
 in OCL, 372

**/\* \*/ operator**  
 in OCL, 386  
**/\*\*/, comments and**, 446  
**// operator**  
 in OCL, 386  
**//, comments and**, 446  
**/ = operator**, 459  
**= operator**, 459  
 for parameter default value,  
 428  
 in OCL, 372, 378–380  
**== (equal operator)**, 189,  
 191  
**== operator**, 459  
 in OCL, 372  
**? operator**, 459  
**@ operator in OCL**, 372  
**@pre**, 441  
**[ ] for navigated end of  
association classes**, 349  
**% operators**, 458  
**% = operator**, 459  
**(double colon)**, 157  
**\_ (underscore)**  
**in UML**, 156  
**{}, constraints**, 356

## A

**abstract classes**, 71,  
 200  
**abstract classifiers**, 200  
**abstract data type**, 738  
 OOP languages, 745  
**abstract keyword**, 184  
**abstract operations**, 78  
**abstract syntax**, 44, 147

- abstraction**, **50**, 181–182, **664**, **738**  
and types, 738  
commands and, 182  
instance, 50, 665  
key abstraction, 664  
personal organizer, 613  
properties, 51  
vs. class, 58
- abstraction dependency**, 169
- abstraction keyword**, 169
- access control**, 668  
via commands, 127
- access keyword**, 162, 166
- access rights**, 668
- accessibility**, 161
- accumulator in OCL iterations**, 384
- Action class**, 201, 204, 321–323
- action execution specification**  
in interaction, 526  
trace of events, 528
- action language**, **78**  
in entity-relationship, 732  
in OOP languages, 747  
in relational paradigm, 704
- ActionGroup class**, 360
- actions**, **44**, **59**, **78**, **80**  
on association classes, 347  
on association ends, **325**, 328–329  
on attributes, **263**, 278  
on binary associations, 319  
vs. commands, 125  
concurrency control, 263, 325  
concurrency in OOIS UML, 472  
constrained group, 95, 203, 204  
constrained groups, 218  
in entity-relationship, **732**  
exceptions, 468  
groups, 95  
isolated group, 475  
isolation in OOIS UML, 474  
on N-ary associations, 339  
in OOP languages, 747
- pins, 80  
on properties, **325**  
raising exceptions, 477  
in relational paradigm, **704**  
standard UML vs. OOIS UML, 199
- on static association end, 302  
surface language, 81  
on variable, 175
- active state**, **496**
- active state configuration**, **504**
- activities**, **200**, **647**  
vs. business process, 647  
in business process, 647  
decomposition, 647  
ordering, 647
- activity diagram**, **648**  
activity node, 648  
decision node, 648  
final node, 649  
fork node, 648  
initial node, 648  
join node, 648  
merge node, 649  
in OOIS UML, 649
- activity node**, **648**
- actor keyword**, 632
- actors**, **631**  
as classifiers, 632  
generalization/specialization, 632  
names, 632  
notation, 632  
use cases association with, 635
- actual argument of interaction reference**, 543
- acyclic associations**, **314**  
N-ary associations, 338
- acyclic keyword**, 316
- add**, 362  
on attributes, 272
- add value action**  
on attributes, 272, 362  
on properties, 362  
on variables, 362
- addAt**, 362
- on attributes, 272  
in OCL, 388
- addFirst**, 362  
on attributes, 272
- addition**, 458
- addLast**, 362  
on attributes, 272
- aggregate functions**, **717**
- aggregates of N-ary association end**, 338
- aggregation**, **296**  
in entity-relationship, **730**
- AJAX**, 569, 680
- algorithm reuse**, 689
- algorithmic decomposition**, **690**
- algorithmic decomposition of activities**, 647
- aliases**, 713  
in element import, 164  
in OQL, 402, 407
- allInstances**, 371
- allow as multiplicity relationship**, 171
- alt**, 536
- analysis model**, 600, 606  
behavior, 606  
constraints, 363  
structure, 606
- analysis patterns**, 610
- ancestors**, 69  
and **operator**, 458  
in OCL, 372
- anemic domain model anti-pattern**, 587
- annotations**, 151
- anti-pattern**, 587
- anti-symmetry of type conformance**, 178
- any in OCL**, 384
- API (application programming interface)**, 684
- append in OCL**, 379, 380
- applets**, 680
- application architecture**, **586**
- application programming interface (API)**, 684
- application server**, **676**, 679, 680

- applications in OOS UML**, 472  
**applyTo command**, 550  
 in constraints, 367  
 in queries, 396
- appropriateness**, 667
- architecture**, 687  
 in OOS UML, **586**  
 reuse, 690
- argument passing, semantics**,  
 454
- arguments**  
 binding to formal parameters,  
 434  
 checking correctness, 440  
 of interaction, 543  
 matching, 430  
 in messages, 533  
 of OQL queries, 412  
 passing, 429  
 of pattern object structure,  
 419
- arithmetic operators in OCL**,  
 372
- arrow vs. dot**, 264
- AS**  
 in OQL, 402  
 in SQL, 713
- asBag in OCL**, 378–380
- ASC**  
 in OQL, 409  
 in SQL, 718
- ASCII in UML**, 155
- asOrderedSet in OCL**,  
 378–380
- assemblies, in C##**, 746
- asSequence in OCL**, 378–381
- asSet in OCL**, 378–380
- assignments**, 198  
 of arguments, 179  
 of attributes, 271, 279, 362  
 classes, 198  
 compound, 459  
 in OO programming language,  
 453  
 in OOS UML, 179  
 of property, 362  
 of variable, 362
- assignment operators**, 459
- Association**, 321
- association**  
 aggregation, **296**  
 composition, **297**  
 multiplicity, 87  
 in user interface, 35
- association classes**, 68, **343**  
 abstraction, 348  
 actions, 347  
 characteristics, 345  
 vs. class with binary  
 associations, 351  
 constructors, 348  
 create link object actions,  
 347  
 creation of instances, 202  
 destructors, 348  
 external multiplicity, 351  
 instances of, 344  
 internal multiplicity, 351  
 names, 345  
 navigation to linked objects,  
 350  
 notation, 345  
 pseudo-properties, 348  
 specialization, 348  
 uniqueness, 345, 350  
 visibility, 345
- association ends**  
 accessibility of class, 286  
 actions, 328, 329, 339  
**actions**, **325**  
 actions for association  
 classes, 348  
 aggregate, **296**, 338  
 vs. attributes, 68, 193  
 composite, **297**  
 constraints, 357, 358  
 constraints in OOS UML, 193  
 context, 333  
 default, 285  
 default multiplicity, 172  
 default uniqueness, 173  
 derived, 338  
**derived**, **307**  
 and existence dependency,  
 239  
 freezing, 338  
**frozen**, **306**  
 multiplicity, 87
- multiplicity, 284, 358  
 multiplicity, **287**, **335**  
 as multiplicity elements, 284  
 name, 285  
 navigability, **294**, 301, 338  
 navigable, 328  
 navigated ends, 348  
 navigation in OCL, 370  
 navigation in OQL, 403, 407  
 notation, 285, 286  
 notation variants, 193  
 of N-ary associations, 333,  
 338  
 ordering, 284, **287**  
 ownership, 285, 294, 301,  
 338  
 propagated destruction, 236,  
 338  
 as properties, 284, 285  
 read with filter action, 330  
 read-only, **303**, 338  
 redefining, **316**, 338  
 redefining multiplicity, 318  
 redefining name, 318  
 redefining type, 318  
 redefining visibility, 318  
 relink action, 330  
 reorder actions, 329  
 replace action, 330  
 semantics, 193, 284, **287**  
 static, **300**, 338  
 subset, **310**  
 substitution, 177  
 as typed elements, 284  
 union, **312**  
 uniqueness, 284, **287**, 345  
 visibility, 286
- association ends**, **282**
- associations**, **62**, **281**  
 actions, 319  
 acyclic, **314**  
 association classes. See  
 association class  
 vs. attributes, 193  
 binary, **281**. See also binary  
 associations  
 as classifiers, 283  
 conceptual modeling,  
 620

### **associations (*continued*)**

constraints, 358  
constraints in OOIS UML, 193  
create link, 320  
derived, **307**  
destroy links, 323  
extent, **287**  
implementation in relational database, 68  
instances of. See link  
interactive manifestations, 66, 67  
modeling history, 68  
multiplicity and, 68  
N-ary, **331**. *See also* N-ary association  
as named elements, 283  
names, 62  
naming, 283  
notation, 62, 283  
notation variants, 193  
as packageable elements, 283  
vs. relationship set, 68  
reflective, **314**  
roles, 62, **282**  
semantics, 63, 193, **287**

**asterisk (\*) in SQL**, 716

**async**, 439

**asynchronous actions**, 673

**asynchronous message**, 532

**asynchronous operation calls**, 438, 532

at

- in OCL, 379, 380, 388
- on attributes, 266

**ATM**, 667

- extend relationship, 641
- generalization/specialization, 642
- include relationship, 638
- interaction diagram, 645
- use cases, 633

**atomic execution**, 479

**attribute value specifications**, **222**

- conditions, 227

**attribute values**, 54, 56, 247

- access through reflection, 277

**attributes**, **53**, 247

- access through reflection, 277
- actions, **263**, 278
- add value action, 272, 362
- assignment, 362
- vs. association end, 68
- built-in types, 54
- clear action, 362
- clear actions, 269
- composite, in ER, **730**
- conceptual modeling, 614
- constraints, 357, 358
- constraints in OOIS UML, 193
- count action, 266
- default multiplicity, 172
- default uniqueness, 173
- default value redefinition, 262
- default values, 54, **249**
- derived, 247, **255**
- derived, in ER, **730**
- in entity-relationship, **728**, 730
- as features, 248
- frozen, **255**
- initialization, 202
- initialization, **249**
- insignificant, 615
- interactive manifestation, 56
- iterations, 276
- mapping to relational model, 733
- modifiable, 254
- multiplicity, **248**
- as multiplicity elements, 247, **248**
- multiplicity redefinition, 261
- multivalued, in ER, **730**
- name, 53
- name redefinition, 261
- as named elements, 248
- navigation in OCL, 370
- navigation in OQL, 403, 407
- notation, 249

notation variants, 193

persistence, 263

as properties, 247

read action, **264**, 265, 279, 362

read size actions, 264

read value action, 265, 266

read-only, 247, **252**

as redefinable elements, 248

redefinition, 109, **260**

in relational paradigm, 696

remove value actions, 269, 270, 362

semantics, 54, 193, **249**

set actions, 271, 279, 362

significant, 615

slots, 64

static, 248, **250**

as structural features, 247

substitution, 176

type, 54

type redefinition, 261

as typed elements, 247, **248**

types, **248**

unique, **188**

visibility redefinition, 261

vs. association ends, 193

write actions, **268**

**authentication**, 668

**authorization**, 668

- of commands, 551
- via commands, 127

in relational databases, 722

**availability**, 668

Avg

- in OQL, 409
- in SQL, 717

## B

**B2B**, 682

**background processing**, 439

**backward error recovery**, **478**

- recovery point, 478
- rollback, 478
- transactions, 479

Bag

- in OCL, 375  
operations in OCL, 379  
**bag string**, 285  
**base classes**, 75  
destructors, 235  
**base collection of tuples**, 402  
**base-classes**, 69  
**basic fact/derived information trade-off**, 258  
**behavior**, 146, **444**  
of collaboration, 521  
as methods, 444  
modeling with data flow, 734  
state machines, 487  
transition effects, 491  
**behavior diagram**, 150  
**behavioral features**,  
  **classifiers**, 183  
**binary associations**, 281  
actions, 319  
notation, 283  
**binary trees, by creational specification**, 230  
**binding**  
  of GUI component, 574  
  of GUI widgets, 575  
**BLOB**, 244  
**blocks**  
  in if, 459  
  in loops, 462  
  nesting, 456  
  in OOIS UML, 456  
**transactions**, 479  
**in try-catch**, 469  
**Booch, G.**, 27  
**booking seats**, 305  
**Boolean in OCL**, 373  
**Boolean data type**, 244  
**Boolean operators**, 458  
**Boolean type**, 448  
**break in interaction fragment**, 537  
**break statement**, 461, **463**  
**browsers, toolbar commands**, 124  
**built-in commands**, **112, 547**, 552  
applicable types, 553  
  **CmdAddValueToSlot**, 554  
  **CmdClearSlot**, 554  
  **CmdCopySlot**, 554  
  **CmdCreateLink**, 558  
  **CmdCreateObject**, 553  
  **CmdCreateObjectAndLinkTo Object**, 560  
  **CmdCreateObjectAndLink ToTwo**, 560  
  **CmdCreateObjectOfClass**, 553  
  **CmdDeleteLink**, 558  
  **CmdDestroyObject**, 553  
  **CmdRemoveValueFromSlot**, 554  
  **CmdReorderValuesInSlot**, 554  
  **CmdReplaceLink**, 558  
  **CmdReplaceValueInSlot**, 554  
  **CmdSetSlot**, 554  
  create links, 553  
  on slots, 552  
  parameters, 552  
**built-in data types, literals**, 245  
**built-in element of OOIS UML**, 45  
**built-in types**, 241, 244, 738  
  in OCL, 370  
**bulk inputs**, 581  
**business logic**, 587, 679  
  and commands, 127  
**business objects, and commands**, 127  
**business process modeling**, 651  
**business processes**, **604, 647**  
  vs. activity, 647  
  activity diagram, 648  
  discovering use cases, 651  
  modeling, 648, 651  
  vs. use cases, 650  
**business use cases**, **651**  
**business-to-business**, 682  
**byte type**, 448
- C**  
**C##**, 79  
assemblies, 746  
OOP concepts, 745  
**C++**, 79  
declaring exceptions, 471  
OOP concepts, 745  
transactions, 480  
virtual functions, 86  
**call**, 169  
**call dependency**, 169  
**call events in state machines**, **488**  
**call operation actions**, **429**  
argument binding, 434  
as state machine triggers, 488  
asynchronous, 464, 472  
asynchronous calls, 438  
in creational specifications, 228  
in detail-level language, 430  
method resolution policy, 444  
notation, 229  
null target instances, 435  
operation specifications, 430  
polymorphism, 230  
recursion, 230  
results, 436  
synchronous, 464, 472  
synchronous calls, 438  
target instances, 434  
termination, 464  
through reflection, 433  
thrown exceptions, 468  
**callOperation**, 433  
**calls**  
  operations, **429**  
  through reflection, 433  
**car manufacturing example of creators**, 210–213  
**car trading**, 623  
**cardinality**, 87, **171**. *See also multiplicity*  
  in entity-relationship, 733  
  in entity-relationship, **728**  
**Cascade Style Sheet**, 564

# cascading deletion

---

- cascading deletion**, 707  
in OVIS UML, **236**
- cascading update**, 707
- case**, 461
- case sensitivity in UML**, 155
- casting**  
in OCL, 371  
operators, 457
- casting**, **449**  
catch, 466, **468**  
char **type**, 448
- character sets in UML**, 155
- checkInput Pins Coverage  
**command**, 550
- children**, 69
- choice pseudostate**, **501**
- city routes**, 305
- Class **command**, 553
- class diagram**, 51
- class **keyword**, 184
- class utilities**, **424**
- classes**, **51**, 183  
abstract, 71  
and abstract data types, 739  
vs. abstraction, 58  
assignment, 198  
association classes. See  
    association class  
base classes, 75  
conceptual modeling, 612  
constraints, 357, 358  
constraints in OVIS UML, 193  
constructors, **206**  
copy semantics, 197  
vs. data types, 61, 185, 193,  
    612  
destructors, 234  
encapsulation, 159  
vs. entity set, 60  
equality, 192  
extent. See *itshape extent*  
identification, 58  
implementation in relational  
    database, 60  
instances, 185–189  
direct/indirect, 71  
interactive manifestation, 56  
invariants, 357, 359
- lifetime of objects, 198  
multiple classification, 619  
as namespace, 156  
notation, **147, 184**  
object reclassification, 59  
ObjectOf Class, 113  
objects  
creation, 56  
deleting, 56  
as instances, 52  
in OOP languages, 745  
operations, 195, 196  
Query, 136  
responsibilities, 613  
roles, 613  
semantics, 52, 193  
specification, 612  
state machines, 487
- classification**, 738  
explicit, direct, 627, 628  
implicit, indirect, 624  
value-based, 624
- classifier instances**, **141**  
equality, 191  
multiple classification, 619  
structure, 193
- Classifier object type**, 202
- classifiers**, 65, 183, **141**  
actors, 632  
associations, 283  
collaboration, 521  
constraints, 357  
constructors, **206**  
features, 182, **183**  
instances, 182  
invariants, 357  
multiple classification, 619  
as namespaces, 183  
notation, 184  
as packageable elements,  
    183  
semantics, 181  
as types, 183  
use cases, 635
- clear**, 362  
on attributes, 269
- clear actions**  
on attributes, 269, 362
- on properties, 362  
on variable, 362
- client/server architecture**,  
    **677**
- clients**, **676**, 679  
of dependency relationship,  
    168  
thick, 679  
thin, 679
- clones**, 367, 625  
of command, 549  
in queries, 396
- clone operation**, 197
- cloning**, 197  
objects, 187
- cloning**, **449**
- CmdAddValueTo Slot  
**command**, 554
- CmdApplyConstraint To  
    Object **command**, 368
- CmdApplyQueryToObject  
**command**, 397
- CmdClearSlot **command**, 554
- CmdCopySlot **command**, 554
- CmdCreateObjectAndLink  
    ToObject **command**, 550
- CmdCreateLink **command**,  
    113, 553, 558
- CmdCreateObject **command**,  
    553
- CmdCreateObjectAndLinkTo  
    Object **command**, 112,  
    560
- CmdCreateObjectAndLink  
    ToTwo **command**, 560
- CmdCreateObjectOf Class  
**command**, 553
- CmdDeleteLink **command**,  
    558
- CmdDestroyObject  
**command**, 553
- CmdEvaluateConstraint  
**command**, 368
- CmdEvaluateQuery  
**command**, 397
- CmdRemoveValueFrom Slot  
**command**, 554

**comments**

- 
- CmdReorderValuesInSlot  
**command**, 554  
 CmdReplaceLink **command**,  
   558  
 CmdReplaceValueInSlot  
**command**, 554  
 CmdSetSlot **command**, 554  
**code generator**, 9  
**collaboration**, 520, 744  
   behavior, 521  
   as a classifier, 521  
   connector, 520, **521**  
   implementing a mechanism,  
   523  
   implementing an operation,  
   523  
   notation, 520, **521**  
   owned by a classifier, 523  
   owned by a package, 523  
   property, 521  
   roles, 520, **521**  
   role binding, 522  
   semantics, 521  
**collaboration**, 517, **521**  
**collaboration use**, **523**  
   and design pattern, 523  
   role binding, 523  
**collect in OCL**, 382  
**Collection**, 451  
   in OCL, 375  
   operations in OCL, 377  
**collection literals**, 376, 386  
**collection operations in OCL**,  
   387  
**collections**  
   as instance of multiplicity  
   element, 172  
   iterators in OCL, 381  
   in OCL, 370, 375, 387  
   operations in OCL, 377  
   sub-collection, 312  
**columns in relational**  
   paradigm, **696**  
**combined fragment**, **536**  
   alt, 536  
   alternative, 536  
   break, 537  
   coregion, 538  
   critical, 538  
   critical region, 538  
   loop, 540  
   neg, 542  
   negative traces, 542  
   operand, 536  
   operator, 536  
   opt, 537  
   option, 537  
   par, 538  
   parallel, 538  
   ref, 543  
   reference, 543  
   strict, 539  
   strict sequencing, 539  
**Command class**, 109, **548**  
**commands**, **109**, **547**  
   abstractions and, 182  
   access control, 127  
   vs. actions, 125  
   add value to slot, 554  
   application on collections,  
   110  
   applyTo, 550  
   in architecture, 587  
   authorization, 551  
   background processing, 439  
   browser toolbar, 124  
   built-in, **112**, 547  
   checkInput Pins Coverage,  
   550  
   as classes, 109  
   clear slot, 554  
   clone, 549  
   cloning, 113, 549  
   Command class, **548**  
   configuration parameters,  
   549  
   context menus, 125  
   copy slot, 554  
   create and link object, 560  
   create link, 113, 558  
   create object, 553  
   create object and link to  
   object, 112  
   customization of GUI,  
   114–115  
   delete link, 558  
   description, 548  
   destroy object, 553  
   domain-specific, 114, **547**  
   drag-and-drop, 125  
   execute, 551  
   execution, 115, 550  
   features, 109, **548**  
   generic, **547**  
   generic, 112, 547  
   getFavoritePinForFirst  
   Param, 549  
   getFavoritePinForSecond  
   Param, 549  
   getInput Pins, 549  
   getOutput Pins, 552  
   handle, 550  
   handling, 550  
   identification, 548  
   input pin, 109  
   input pins, 549  
   interactive manifestation,  
   124  
   isApplicableTo, 550  
   isApplicableToAll, 550  
   isPrototype, 549  
   issuing from a GUI  
   component, 583  
   logging, 126, 551  
   name, 548  
   one-pin creational, 112, 116  
   OOIS UML API, 135  
   vs. operation, 126  
   output pins, 552  
   preconfigured, 113  
   prototypes, 113  
   prototyping, 549  
   rationale, 126  
   remove value from slot, 554  
   reorder values in slot, 554  
   replace link, 558  
   replace value in slot, 554  
   set slot, 554  
   and transactions, 551  
   for triggering state machines,  
   516  
   two-pin relinking, 113, 121  
**comments**  
   notation, 151

**comments (*continued*)**

in OCL, 386  
in OOIS UML, 446

**comments, 151****commit, 479, 683****compartment, 184****compilation, 143****compiler, 9****complete, 618****complete generalization set, 618****completion event, 501, 510****completion transition, 501****complexity, 663, 666, 738****components, 690**

of query tuples, 391

**composite state, 496**

completion, 501

default entry, 509

default substate, 500

entry points, 511

exit behavior, 504

exit points, 511

history, 500

as a namespace, 496

naming, 496

outgoing transition, 504

redefinition, 513

reuse, 515

submachine state, 513

transition, 507

**composition, 297**

multiplicity, 297

order processing, 297

propagated destruction, 297

user management, 297

**compound statements**

execution, 456

in if, 459

in loops, 462

nesting, 456

in OOIS UML, 456

transactions, 479

**compound transitions, 501, 507**

entry and exit points, 513

execution, 513

**computer-based system, 4****conceptual model, 606, 652,**

**664**

importance, 664

**conceptual modeling, 58, 609, 664**

addressing ambiguities, 611

alternatives, 614

association classes, 351

handling alternatives, 621, 623

identifying associations, 620

identifying attributes, 614

identifying classes, 612

identifying data types, 612

identifying generalizations/specializations, 616

identifying multiplicities, 622

importance, 664

inspection, 610

iterative nature, 611

language, 665

N-ary associations, 340

problems in relational paradigm, 16

process, 609

type-instance relationship, 623

**conceptualization, 664****concrete syntax, 44, 147****concurrency, 673**

in compound statements, 456

logical, 673

in OOIS UML, 438, 472

physical, 673

**concurrency control, 37, 673**

of association end actions, 325

of attribute actions, 263

critical section, 675

locking, 675

mutual exclusion, 675

in OOIS UML, 474

in relational databases, 721

**conditional branch in state machines, 501****conditions, in creational specifications, 227****conformance, 176, 178**

and constrained action groups, 361

implementation in OO programming language, 451

in OCL, 387

of ordering, 178

strong, 362

of type, 177

of uniqueness, 178

weak, 362

**connector in collaboration, 520, 521****consistency rule, 88****constrained action groups, 95, 359**

and conformance, 361

and create link action, 322

in creational specification, 218

and destroy link action, 324

in object initialization, 203

in object destruction, 204

nesting, 360

notation, 360

**Constraint model library package, 366**

ConstraintFailedException, 359

**constraints, 88, 353**

action groups, 359

actions, 357

in analysis model, 363

on association ends, 357, 358

on associations, 358

on attributes, 357, 358

on a class, 357

on classes, 358

constrained elements, 354, 357

constrained group, 95

context, 354, 357

in creational specifications, 227

# creational object structure for GUI configuration

---

- on a data type, 357
- on data types, 358
- deferred evaluation, 359
- in design model, 363
- evaluation, 94, 355, 358
- exceptions, 468
- failure, 95, 359
- informal vs. formal, 363
- on interaction, 535
- invariant, 357**
- invariants, 359
- language, 354
- meaning of self, 357
- as model elements, 353
- modeling rules in OOIS UML, 357
- name in OCL, 370
- names, 354
- namespaces, 354
- notation, 356
- on operations, 358
- as objects, 90, **366**
- in OOIS UML, 357
- operation calls, 94, 357
- as packageable elements, 354
- in pattern object structures, 417
- postconditions, 358
- postconditions, 440**
- preconditions, 358
- preconditions, 440**
- predefined, 354
- presentational and interactive manifestation, 91
- on properties, 357, 358
- scope of evaluation, 157
- semantics, 94, 355, 358
- side effects, 94, 357
- specification language, 357
- in standard UML, 353
- in standard UML vs. OOIS UML, 363
- transition guards, 491
- triggers, 355, 358
- unique, 188
- use, 355
- user-defined, 354
- xor, 354
- constructor operation, 202**
- constructors, 202, 206**
- default, 208
- and initial state, 493
- invocation, 208
- of association classes, 348
- of base classifier, 209
- parameters, 207
- and polymorphism, 209
- and read-only association ends, 303
- and read-only attributes, 253
- context**
- of N-ary association ends, 338
- in OCL, 369
- context menus, commands, 125**
- context of association end, 333**
- context reserved word, 369**
- continue, 463**
- contracts, 443, 740**
- control class, 587**
- control flow in OOIS UML, 459**
- control objects in OOSE, 127**
- control structures, 81**
- copy semantics, 197**
- copy-paste, objects, 187**
- copying of instances, 197**
- coregion in interaction, 538**
- Count in SQL, 717**
- count**
- on attributes, 266
- in OCL, 377
- count actions on attributes, 266**
- coupling, 692**
- covering generalization set, 618**
- create in interaction, 533**
- create classifier instance action, 200**
- semantics, **202**
- create data type instance actions, 198**
- new operator, 457
- create data value action, 200**
- semantics, **202**
- create dependency, 169**
- create keyword, 169**
- Create Link action, 288**
- ordering, 291
- create link actions, 320**
- for association classes, 347
- as link specifications, 222
- for N-ary associations, 334, 339
- OOIS UML API, 110
- and read-only association ends, 303
- by reflection, 321
- rules, 321
- semantics, 322
- semantics, **287**
- create link object actions, 347**
- create message, 533**
- create object in OOP languages, 747**
- create object actions, 59, 198, 200**
- for association classes, 347
- in interaction, 533
- new operator, 457
- as object specifications, 221
- OOIS UML, 110
- and read-only association ends, 303
- and read-only attributes, 253
- semantics, **202**
- in standard UML, 199
- create table, 705**
- createBinding, 574**
- createInstance operation, 201**
- createLink operation, 320**
- CreateLinkObject Exception, 348**
- creating link action, 64**
- creational object structure, 210, 216**
- creational object structure for GUI configuration, 106, 115**

**creational semantics**, 217  
**creational specifications**, 216  
attribute value specifications, 222  
call operation actions, 228  
conditions, 227  
constraints, 227  
by demonstration, 232  
diagrams, 217  
group specifications, 223  
iteration specifications, 224  
link specifications, 222  
as method implementation, 224  
object reference specifications, 220  
object specifications, 221  
ordering, sequencing, 230  
semantics, 218  
substructures, 228  
**creators**, 210, **213**  
by demonstration, 232  
code, 216  
substructures, 228  
**critical**, 538  
**critical region, in interaction**, 538  
**critical section**, **675**  
**CRUD analysis**, 654, 655  
**CSS (Cascading Style Sheet)**, 564  
**Currency data type**, 244  
**customizing behavior**  
by modeling, 114  
by demonstration, 115

## D

**dangling references**, 478  
**DAO (data access objects)**, 587  
**data access object**, 587  
**data definition**, 146  
**data definition action**, 705  
**data definition language**, 705  
**data flow**, **734**  
data store, **735**  
external interactor, **735**

hierarchical decomposition, 734  
process, **734**  
semantics, 735  
**data manipulation actions**  
and constraints, 704  
in entity-relationship, **732**  
in relational paradigm, **703**  
**data members**, 746  
**data modeling**, 146  
**data store in data flow**, **735**  
**data type instances**  
creation, **200**  
destruction, **205**  
equality, 191  
as identifier, 190  
identity, **190**  
semantics, 190  
**data types**, 183  
assignment, 198  
built-in, 241, 244  
conceptual modeling, 612  
constraints, 357, 358  
constraints in OOIS UML, 193  
constructors, **206**  
copy semantics, 197  
enumeration, 241, **242**  
equality, 198  
invariants, 357, 359  
lifetime of instances, 198  
literals, **203**  
notation, **184**  
operations, 195, 196, 426  
primitive, 240, **241**  
reference sharing, 198  
semantics, 193  
standard, 61  
structure, 240  
user-defined, 241, 244  
vs. classes, 61, 185, 193, 612  
**data values**, **185**  
identity, **190**  
lifetime, 198  
**data-manipulation language**  
in entity-relationship, 732  
in relational paradigm, 704

**database management system (DBMS)**, 672  
**database modeling**  
tools, 722  
visual, 722  
**database schema**, 705, 710  
data definition actions, 705  
views, 706  
**database server**, **676**, 679  
**databases**  
hierarchical, 672  
metadata, 710  
network, 672  
object-oriented, 672  
paradigms, 672  
relational, 672  
dataType **keyword**, 184  
Date **data type**, 244  
DateTime **data type**, 244  
**DBMSs (database management system)**, **672**  
roles, 721  
L,D  
705  
**deadlocks**, **476**  
**debugging, model debugging**, 46  
**decision node in activity diagram**, 648  
**declarations**  
in for, 462  
in OOIS UML, 456  
in OOIS UML, **457**  
of variable, 175, 447  
**declarative propagated destruction**, 237  
**decomposition**, **690**  
algorithmic, 690. See also procedural decomposition  
in data flow, 734  
loose coupling, 692  
object-oriented, 690, 743.  
See also object-oriented decomposition  
procedural, 690  
**decrement**, 458

- deep copy**, 197  
**default constructor**, 208  
   of association classes, 348  
**default destructors**, 235  
**default history state**, 498  
**default values**  
   of output parameter, 464  
   of parameters, 428  
   redefinition, 109  
   of variables, 449  
**DELETE, in SQL**, 720  
**delete, in C++**, 747  
**delete field**, 705  
**delete operator**, 206  
**delete records**, 704  
   cascading deletion, 707  
   cascading update, 707  
**delete table**, 705  
**demonstration**  
   of creational specifications, 232  
   of pattern object specifications, 420  
**deparment hierarchy**, 314  
**dependencies**, 167  
   abstraction, 169  
   call, 169  
   clients, 168  
   create, 169  
   derive, 170  
   instantiate, 169  
   notation, 169  
   realization, 170  
   refine, 170  
   semantics, 168  
   suppliers, 168  
   trace, 170  
   usage, 169  
**derive dependency**, 170  
**derive keyword**, 170  
**derived actions, association ends**, 328  
**derived association ends**, 307  
   actions on, 309  
   maintenance strategies, 310  
   of N-ary association, 338  
   notation, 309  
   redefining, 318  
   semantics, 309  
   use, 310  
**derived associations**, 307  
   actions on, 309  
   maintenance strategies, 310  
   notation, 309  
   semantics, 309  
   use, 310  
**derived attributes**, 247, 255  
   actions on, 257  
   implicit update, 257  
   maintenance strategies, 257  
   notation, 256  
   vs. queries, 258  
   redefinition, 262  
   semantics, 256  
   vs. stored procedures, 258  
   uses, 259  
   vs. views, 258  
**derived classes**, 69  
**derived unions**, 312  
   DESC  
   in OQL, 409  
   in SQL, 718  
**descendants**, 69  
**description of command**, 548  
**design**, 594  
**design anti-pattern**, 587  
**design decision**, 692  
**design method**, 12  
**design model, constraints**, 363  
**design patterns**, 689  
   describing by collaborations, 523  
   use in conceptual modeling, 610  
   prototype, 114, 549, 625  
   singletons, 301  
   template method, 551  
**desktop GUI vs. Web GUI**, 569  
**destroy, in interaction**, 533  
**destroy keyword**  
   propagated destruction, 238  
**Destroy Link action**, 289, 290  
   ordering, 291  
**destroy link actions**, 323  
   for association classes, 348  
   for N-ary associations, 334, 339  
   in object destruction, 204  
   and read-only association ends, 303  
   rules, 324  
   semantics, 287, 324  
**destroy message**, 533  
**destroy object in OOP languages**, 747  
**destroy object actions**, 198, 204  
   destructors, 234  
   in interaction, 533  
   in propagated destruction, 237  
   in standard UML, 199  
   semantics, 204  
   by a terminable pseudostate, 501  
**destroy operation**, 204  
**destroy operator**, 323  
**destroying link action**, 64  
**destroyInstance operation**, 204  
**destroyLink operation**, 323  
**destruction, propagation**, 236  
**destruction event**, 533  
**destruction of exceptions**, 468  
**destructor class**, 234  
**destructor operation**, 204  
**destructors**, 204, 234  
   of association classes, 348  
   of base class, 235  
   default, 235  
   invocation, 235  
   parameters, 234  
   polymorphism, 235  
**detail-level language**, 78  
   attribute actions, 278  
   implementation of variables, 451  
**OOIS UML**, 445  
**Development, in relational paradigm**, 19

**development method, 593**

activities, **593**

artifacts, **593**

**development phase**

**discontinuity**, 22, 36

in object-oriented

programming, 26

in OO approaches, 33

**development principles, 688****development process, 12**

activities, **593**

artifacts, **593**

in OOIS UML, 44

planning iterations, 658

prototyping, 658

**development tools**

for relational paradigms, 722

for reporting, 723

**diagram, 147**

class, 51. *See also* class diagram

in entity-relationship, 728

object, 52 *See also* object diagram

sequence, 519, 524

use case, **636**

**dialog tabs, as GUI component**

**component**, 583

**dialogs**

as GUI component, 583

opening, 56

**dichotomies, 141**

basic-derived concepts, 144

classifier-instance, 141

compilation-interpretation, 143

core-extended part, 146

formal-informal concepts, 145

model-diagram, 147

modeling-execution, 142

structure-behavior, 146

**direct classification, 627, 628****direct instances of classes, 71****discontinuity, 20, 36**

development phase, 22

in OO approaches, 33  
in traditional development, 20

**scope discontinuities, 20****semantic, 21****disjoint, 618****disjoint generalization set, 618****DISTINCT**

in OQL, 410

in SQL, **713**

semantics in OQL, 400

**distributed architecture, 681****distributed system, 676****distribution, 37, 676**

in relational databases, 721

**distribution of responsibilities, 744****division, 458****do-while, 461**

doesMatch, 572

doesMatchFully, 572

**domain mapping, 213****domain models, 586****domain-specific commands, 114****domains, 213, 663**

in relational paradigm, **701**

source domains, 213

target domains, 213

**dot operator**

in OCL, 372

in OCL tuples, 375

**dot vs. arrow, 264****dots**

association ends, 285

feature access, 457

for query results, 393

navigation in OCL, 370

null target instance, 459

for operation calls, 435

in OQL queries, 403, 407

**double type, 448****downcasting, 450**

in OCL, 371

in OQL, 130, 403

**DRY principle, 692****dynamic semantics, 46****E****Easylearn school, 41, 695**

associations, 62

command, 108

concurrency control, 474

constraint, 86

constraints, 353, 363

derived attributes, 255

entity-relationship, 727

generalization/specification, 69

multiplicity, 86

object-oriented programming, 745

OCL, 369

operations, 76

OQL, 399

pattern object structure, 415

polymorphism, 84

presentation, 97

queries, 392

query, 128

relational model, 696

SQL, 711

user interface, 35

**effects, 491**

context, 492

execution, 493, 509, 513

notation, 492

**element, 151.** *See also* model element

**element access, 167****Element command, 570****element import, 167****elements**

importing, **162**

aliases, 164

name clash, 164

notation, 162, 164

semantics, 162

visibility, 162

in OOIS UML, 548, 570

**ellipse**

for use cases, 636

**else, 459**

in interaction fragment, 536

**employment tracking**

- association classes, 344, 351  
 unique association classes, 347
- encapsulation, 740**  
 in OOP languages, 746  
 types, 738  
 in UML, 159
- encryption, 668**
- Enterprise Java Beans, 31**
- entities**  
 in entity-relationship, 727  
 weak, 729
- entity classes, 587**
- entity set, 728**  
 vs. class, 60  
 mapping to relational model, 733
- entity-relationship, 16, 727**  
 action language, 21, 732  
 aggregation, 730  
 attributes, 728  
 basic concepts, 727  
 cardinality, 728, 733  
 data manipulation actions, 732  
 data representation, 729  
 diagrams, 728  
 discontinuities, 21  
 entity, 727  
 entity sets, 728  
 ERROL, 732  
 generic operations, 732  
 mapping to relational model, 732  
 notation, 728  
 in practice, 21  
 primary keys, 729  
 queries, 732  
 query language, 732  
 and relational model, 732  
 relationship, 727  
 relationship sets, 728  
 roles, 728  
 specialization, 730  
 surface language, 732  
 and UML, 15  
 weak entities, 729
- entry behavior, 503, 509**
- entry points, 511**
- enumeration, 241, 242**  
 literals, 241, 242  
 in switch, 461
- enumeration **keyword, 243**
- equality comparison of enumeration, 243**
- equality of data values, 191**
- equality of objects, 189**
- equality operators, 459**
- ERROL, 732**
- event dispatching in state machine, 506**
- event occurrence in state machines, 488**
- event ordering, 530**
- event-driven behavior, 483**
- events, 485**  
 completion, 501, 510  
 in state machines, 488
- exception handler, 466**  
 default, 469
- exception handling, 465, 469**  
 obsolete approach, 465
- exceptions, 465**  
 in C++, 471  
 catching, 466, 468  
 checked, 471  
 declaration for operation, 470  
 destruction, 468  
 identity, 468  
 in Java, 471  
 lifetime, 468  
 logging, 468  
 notion, 467  
 propagation, 469  
 sources, 469, 477  
 throwing, 466, 468  
 type, 467  
 unchecked, 471
- excludes, in OCL, 377**
- excludesAll, in OCL, 377**
- excluding, in OCL, 378, 380**
- executable model, 5, 12**
- executable modeling language, 7, 12, 36**
- execute, in queries, 397**
- execute command, 109, 547, 551**
- execution environment. See runtime environment**
- existence dependence, in entity-relationship, 730**
- existence dependency, 238**  
 and multiplicity, 239  
 and propagated destruction, 239
- exists, 89**  
 in OCL, 384
- exit behavior, 503, 508**
- exit points, 511**
- explicit classification, 627, 628**
- explicit lifetime, 198**
- expressions**  
 in OCL, 372  
 in OOIS UML, 457  
 type, 459
- expressive power, 137**  
 of query specifications, 137
- expressiveness, 6, 36, 137**  
 of query specifications, 137
- extend keyword, 642**
- extend relationship**  
 condition, 642  
 extension point, 640  
 as named element, 642  
 notation, 642
- extend relationship, 640, 641**
- extension point, extend relationship, 640**  
 notation, 641  
 rules, 641
- extent, 51**
- extent of association, 287**
- external interactor, in data flow, 735**
- external multiplicity, 342**  
 association classes, 351

**F**

- facade design patterns, 424**
- failure, sources, 477**
- fast prototyping, 36**

**fault tolerance**, 37, **477**, 682  
in relational databases, 722

**feature access operator**, 457  
null target instance, 459

## fields

in OOP languages, 746  
in operation call, 433  
for query results, 394, 397,  
413  
in relational paradigm, 696  
for results of operation calls,  
436

**File data type**, 244

**final node**, in activity diagram,  
649

**final state**, 501

## finalization

destructors, 234  
of link objects, 348  
steps, 238

**first**, 362

in OCL, 379, 380  
on attributes, 266

**flatten**, in OCL, 378, 380

**flexibility**, 688  
by polymorphism, 85, 743

**flight schedule**

multiplicity, 337  
N-ary associations, 331

**float type**, 448

**for statement**, 462

**forAll**, 89

in OCL, 383

**forEach**

on attributes, 276  
for query results, 393, 397

**forEach loop**, in **creational specifications**, 225

**forEachReverse**, on **attributes**, 277

**forEachReverse loop**, in **creational specifications**, 225

**foreign keys**, 698

**forGroup keyword**, 227

**fork node**, in activity diagram,  
648

**form**, 35

**formal concepts**, 145  
**formal parameters**, 426  
argument binding, 434  
binding by name, 436  
default values, 428, 430,  
433, 436, 464  
direction, 427  
in, 392, 427  
input, 432, 433  
of interaction, 543  
matching, 430  
multiplicity, 427  
names, 426  
notation, 427  
of operations, 426  
of OQL queries, 411  
of pattern object structure,  
418  
ordering, 427  
out, 393, 427  
output, 433, 464  
of queries, 391, 392  
returns, 427, 464  
types, 427  
uniqueness, 427

**formal semantics**, 36  
of UML, 29, 30

**forms**, 17, 722  
generation, 22  
master-detail, 723

forName, 451

**forward error recovery**, 478

**Fowler, M.**, 587

**fragment of interaction**, 526.

See also interaction  
fragment

**frameworks**, 689

**freeze**, 275

on association end, 306

**freeze attribute**, 255, 275

**freezing**

of N-ary association end,  
338

state, 493

**FROM**

navigation, 403

nested SELECT in OQL, 404

in OQL, 411

semantics in OQL, **399**, **402**

in SQL, **713**

**frozen association ends**, 306

of N-ary association, 338

**frozen attributes**, 255

freezing, 275

writing, 268

**function point**, **665**

**functional dependency**, 709

**functional requirements**, 606

and use cases, 634

**functionality**, 146, 665

complexity, 666

function point, 665

modeling with dataflow, 734

**functions**, 196

## G

**garbage collection**, 198, 206

**gate of interaction**, 543

**general ordering relationship**,  
530

**generalization set**, 618

complete, 618

covering, 618

disjoint, 618

incomplete, 618

multiple classification, 619

notation, 618

in OOIS UML, 620

overlapping, 618

partitioning, 618

power type, 628

**generalization/specification**

of actors, 633

between use cases, 642

conceptual modeling, 616

design of hierarchies, 744

generalization set, 618

identifying, 75, **616**

implementation in relational  
database, 76

inheritance, 69

interactions, 530

interactive manifestation, 74

method resolution policy, 444

multiple inheritance, 75

- and name resolution, 157  
 notation, 71, 618  
 in OQL, 130  
 pitfalls, 621  
 semantics, 69, 177  
 set interpretation, 70  
 sizing, 75  
 substitution, 70, 176  
 typing, 71
- generalization/specialization**  
 relationship, 69
- generic commands.** *See*  
**built-in command**
- generic OOIS UML GUI,** 45
- getAssociation command,**  
 321
- getClassifier command,**  
 202
- getFavoritePinForFirstParam**  
**command,** 549
- getFavoritePinForSecondParam**  
**command,** 549
- getFeature command,** 572
- getInputPins operation,** 549
- getMatchingRate command,**  
 571
- getOutputPins command,**  
 552
- getOwnedFeature command,**  
 572
- GROUP BY  
 in OQL, 409  
 in SQL, 718
- group **keyword,** 224, 360
- group specifications, 223**  
 conditions, 228  
 as method implementation,  
 224  
 notation, 224  
 semantics, 224
- group transition, 504**
- guards, 491**  
 context, 491  
 evaluation, 493, 507  
 notation, 491
- GUI**  
 background processing, 439  
 configuration, 98
- customization, 37  
 design for relational  
 paradigm, 722  
 development with OO  
 approaches, 32  
 flexibility, 23  
 generation, 37  
 idioms, 22  
 manipulation patterns, 22  
 in object paradigm, 34, 37  
 in OOIS UML, **563**  
 personalization, 107  
 in relational paradigm, 17, 22  
 scaffolding, 37  
 for state machine, 515  
 testing, 23
- Web-oriented,** 108
- GUI class setting,** 571
- GUI collection components,**  
 579
- GUI command components,**  
 583
- GUI component layer,** 587
- GUI components, 564**  
 binding, 574  
 collection, 579  
 command, 583  
 composition, 573  
 container, 583  
 context, 574  
 create and link, 582  
 create object, 581  
 creational, 581  
 dialog, 583  
 dialog tab, 583  
 element, 576  
 element dialog, 583  
 element dialog tab, 583  
 features, 573  
 hot spot, 584  
 library, 576  
 multiple element, 578  
 object association ends tab,  
 583  
 object attributes tab, 583  
 object general tab, 583  
 object identification, 581  
 object tab, 583
- pins, 574  
 query, 579  
 search, 580  
 slot editor, 577  
 slot values, 577  
 slots, 576  
 subnodes, 579  
 tile, 583
- GUI components layer,** 564, 573
- GUI configuration,** 587  
 commands, 124  
 context, 101  
 customizing behavior, 114, 115  
 by demonstration, 127  
 by modeling, 127
- personalization,** 107  
 at runtime, 106  
 for Web, 108
- GUI container components,**  
 583
- GUI context, 101,** 564, **572**  
 of component, 574  
 notation, 101, 103, 104  
 specialization, 101
- GUI create object and link**  
**components,** 582
- GUI create object**  
**components,** 581
- GUI creational components,**  
 581
- GUI dialog components,** 583
- GUI dialog tab components,**  
 583
- GUI element components,** 576
- GUI element dialog**  
**components,** 583
- GUI element dialog tab**  
**components,** 583
- GUI hot spot components,** 584
- GUI item,** 98
- GUI item setting,** **98,** 564, **570**  
 commands, 114  
 extent, 571  
 features, 570  
 inheritance, 572

### GUI item setting (*continued*)

matching rate, 571  
matching rule, 107  
notation, 99  
redefinition, 572  
specializations, 99, 571  
subsetting, 572  
**GUI multiple element components**, 578  
**GUI object association ends tab components**, 583  
**GUI object attributes tab components**, 583  
**GUI object general tab components**, 583  
**GUI object identification components**, 581  
**GUI object setting**, 571  
**GUI object tab components**, 583  
**GUI property setting**, 571  
**GUI query components**, 579  
**GUI search components**, 580  
**GUI setting feature**, 572  
inheritance, 572  
redefinition, 572  
**GUI slot components**, 576  
**GUI slot editor components**, 577  
**GUI slot setting**, 571  
**GUI slot value components**, 577  
**GUI style configuration**, 564, 570  
**GUI style layer**, 587  
**GUI subnodes components**, 579  
**GUI tile components**, 583  
**GUI widgets**  
binding, 563, 575  
construction, 574  
notification, 575  
in OOIS UML, 564  
pins, 575  
responsibilities, 574  
**GUI widgets layer**, 574  
**GUIClassSetting**, 571  
**GUIComponent**, 573

**GUIComponentBinding**, 574  
**GUIContext**, 101, 572  
**GUIItemSetting**, 99, 571  
**GUIObjectSetting**, 571  
**GUIPropertySetting**, 571  
**GUISettingFeature**, 572  
**GUISlotSetting**, 571  
**GUIWidget**, 574

### H

**H, symbol of history pseudostate**, 498  
**handle, in queries**, 397  
handle **command**, 550  
**Hibernate**, 31  
**hierarchical paradigm**, 695  
**high-level transition**, 504  
**history pseudostate**, 498  
deep, 498  
deep, 500  
default, 498  
shallow, 498  
shallow, 500  
**history state, entry**, 510  
**hook method**, 552  
**host language**, 45  
**host object**, 79, 220  
**HTML (HyperText Markup Language)**, 564, 679, 684  
map tags, 584  
**HTTP (HyperText Transport Protocol)**, 682

### I

**ID document issuance**, 647  
business processes, 647  
business use cases, 652  
conceptual model, 652  
CRUD use cases, 655  
**IDDIMS**, 647  
**Identification**, 668  
of object in GUI, 581  
**identifiers**, 188  
data type instances as, 190

### identity, 53

existence-based, 61  
of link objects, 345  
objects, 185–189  
semantics, 191  
value-based, 53, 61  
**idioms**, 689  
**if**, 459  
**if-then-else-endif, in OCL**, 372, 373  
**illustrative concepts**, 145  
**implementation**, 594  
encapsulation, 740  
**implementation form**, 8  
**implementation language**, 8, 45  
**implicit classification**, 624  
**implicit destruction**, 198  
**implicit lifetime**, 198  
**implies operator, in OCL**, 372  
**import keyword**, 162, 166

### importing

elements, 162  
aliases, 164  
name clash, 164  
notation, 162, 164  
semantics, 162  
packages, 165

### IN

notation, 166  
in OQL, 406  
**in**, 427  
argument binding, 434  
for queries, 392  
**InaccessibleNamedElement Exception**, 277  
**include, as multiplicity relationship**, 171  
**include keyword**, 639  
**include relationship**, 637, 638  
as named element, 639  
notation, 639  
**includes, in OCL**, 377  
**includesAll, in OCL**, 377  
**including, in OCL**, 378–380  
**incomplete**, 618

- incomplete generalization set,** 618
- increment,** 458
- indexes**
- in OCL,** 388
- in relational databases, 709
  - `indexOf`, 388
  - in OCL, 379, 380
- indirect classification,** 624
- indirect instances of classes,** 71
- inference,** 233
- infix operator, in OCL,** 372
- informal concepts,** 145
- information,** 4
- derived, 255, 307, 669
  - factual, 669
- information system**
- development, with traditional OO approach,** 31
- information systems,** 3, 4
- appropriateness, 667
  - availability, 668
  - characteristics, 663
  - complexity, 663
  - conceptualization. 664 *See also* conceptual modeling
  - concurrency control, 673
  - deployment-related characteristics, 671
  - development process, 12. *See also* development process
  - distribution, 676
  - domain-related characteristics, 663
  - evolution, 666
  - functionality, 665
  - interactivity, 667
  - location independence, 668
  - persistence, 672
  - scalability, 671
  - security, 668
  - timeliness, 668
  - usability, 669
- usability-related characteristics, 667
- inheritance,** 69, **741**
- design of hierarchies, 744
  - of features, 741
  - method resolution policy, 444
  - namespaces, 157
  - in OOP languages, 747
  - in OQL, 131
  - and polymorphism, 742
  - of state machines, 489
  - of static association end, 302
  - static attributes, 251
  - and subtyping, 741
  - in user interface, 35
- initial node, in activity diagram,** 648
- initial pseudostate, 500**
- initial state, 493**
- initialization**
- of link objects, 348
  - of objects, **253**
  - and read-only association ends, 303
  - and read-only attributes, 253
  - of static association end, 302
  - of variables, 448
- inline queries,** 413
- inout,** 427
- argument binding, 434, 435
- input parameters, 432, 433**
- argument binding, 434
- input pins,** 80
- of command, 109, 549
  - of GUI component, 574
  - of GUI widget, 563, 575
  - vs. ordinary property, 549
- inputPin,** 109, 549
- INSERT, in SQL, 720**
- insert field,** 705
- insert records,** 704
- insertAt, in OCL,** 379, 380, 388
- instances,** 665
- abstractions, 50
  - of association classes, 344
  - of classifiers, 141
  - direct, indirect, 71
- of multiple classes, 58
- multiple classification, 619
- instantiate dependency,** 169
- instantiate keyword,** 169
- instantiation,** 665
- int type,** 448
- Integer, in OCL,** 373
- Integer data type,** 244
- interaction,** 520, **524**, 691, 744
- action execution specification, 526
  - alternative, alt, 536
  - arguments, 543
  - break, break, 537
  - constraint, 535
  - coregion, 538
  - critical region, critical, 538
  - decomposition, 543
  - diagram, 524
  - fragment, 526. *See also* interaction fragment
  - gate, 543
  - generalization/specialization, 530
  - hosting object, 526
  - lifeline, 525
  - loop, loop, 540
  - message, 526, 532
  - method execution, 526
  - negative, neg, 542
  - notation, 524, 543
  - occurrence specification, 526
  - in OOP language, 26
  - operation call, 526
  - option, opt, 537
  - parallel, par, 538
  - redefining, 530
  - reference, ref, 543
  - reuse, 543
  - sd, 524, 543
  - self, 526
  - semantics, **527**, 534, 536
  - strict sequencing, strict, 539
  - in testing, 530
  - trace of events, 527
  - use, 530

# interaction diagram for use case

---

## interaction diagram for use case, 645

## interaction fragment, 526, 534

alternative, alt, 536  
break, break, 537  
combined, 536. *See also* combined fragment  
coregion, 538  
critical region, critical, 538  
loop, loop, 540  
negative, neg, 542  
nesting, 543  
option, opt, 537  
parallel, par, 538  
reference, ref, 543  
state invariant, 535  
strict sequencing, strict, 539

## interactive manifestation, 46, 66, 67

generalization/specialization, 74

## interactivity, 667

## interfaces, 740

## interleaving, of events, 529

## internal multiplicity, 342

association classes, 351

## internal transitions, 503, 507

## intersection, in OCL, 378, 379

inv, 370

## invariants, 357

evaluation, 359

in OCL, 370

## invocation

of operation through reflection, 433

of operations, 429

## invoking operations, 77

## isApplicableTo

in constraints, 367

in queries, 396

## isApplicableTo command, 550

## isApplicableToAll command, 550

## isEmpty, 388

in OCL, 378

isEqualTo **operation**, 191

## isolated action group, 475

transactions, 479

## isolation

action group, 475

in OOIS UML, **474**

## isPrototype, of command, 549

isUnique, in OCL, 384

## iterate, in OCL, 384

## iteration specifications, 224

notation, 227

## iterations, on attributes, 276

## iterative and incremental

process, planning, 658

## iterative and incremental process model. *See* spiral process model

## iterators, in OCL collections, 381

## J

## Jacobson, I., 27

## Java, 79

built-in scalar types, 448

checked exceptions, 471

control flow statements, 459

declaring exceptions, 471

expressions, 457

vs. OOIS UML native

    detail-level language,  
    445

OOP concepts, 745

operators, 457

polymorphic operations, 86

Throwable, 467

transactions, 480

unchecked exceptions, 471

## JavaScript, 680

## join node in activity diagram, 648

## joins, 700, 703

## K

## key abstraction, 50, 664

discovering, 609

identifying, 612

inventing, 610, 612

## keys

foreign, 698

primary, 697

## L

## labels, in OOIS UML, 463

## large-scale instantiation, 665

last, 362

in OCL, 379, 380

on attributes, 266

## leaf, 86, **426**

## leaf operations, 426

## least common ancestor

of states, 508

of transitions, 508

## legacy system, 605

let expression in OCL, 373, 386

## librarian information system, 236

## libraries, 689

## lifecycle modeling, 485

## lifeline, **525**

notation, 525

selector, 525

## lifetime, **198**

## LIKE

in OQL, 406

in SQL, 716

## link objects, **344**

creation, 202

destruction, 348

finalization, 348

identity, 345

initialization, 348

navigation to linked objects, 350

reading, 348

## link specifications, **222**

conditions, 228

crossing group boundaries, 225

notation, 222

owners, 224

- in pattern object structure, 416
- links, 62, 281**
- as a connector, 520
  - creation, 320
  - deleting, 67
  - destruction, 323
  - in OOP languages, 746
  - semantics, 64
- literals, 203**
- built-in data types, 245
  - of collection in OCL, 376
  - enumeration, 241, 242
  - in OCL, 386
  - of tuple in OCL, 375
- local variables, 457**
- localization, 692**
- location independence, 668**
- locking, 675**
- in association end actions, 325
  - of attributes, 263
  - in OOIS UML, 475
  - optimistic, 722
  - pessimistic, 722
  - in relational databases, 721
- locks**
- deadlock, 476
  - exclusive, 476
  - in OOIS UML, 475
  - read, 476
  - scope, 476
  - shared, 476
  - timeout, 477
  - write, 476
- logging of command, 551**
- logical concurrency, 673**
- logical operators in OCL, 372**
- long type, 448**
- loops, 540**
- in OOIS UML, 461
- loose coupling, 692**
- distribution of responsibilities, 744
  - and encapsulation, 740
  - by polymorphism, 743
- lower bound, 171**
- LowerBoundViolationException, 267, 269–270, 273, 322**
- ## M
- M0 level, 142**
- M1 level, 142**
- mailing list, 517**
- mainframe, 677**
- maps, 584**
- master-detail forms, 18, 723**
- matching rate, 571**
- Max in SQL, 717**
- member functions, 746**
- members of a namespace, 156, 164**
- merge node in activity diagram, 649**
- messages, 532**
- arguments, 533
  - asynchronous, 532
  - create, 533
  - destroy, 533
  - in interaction, 526
  - notation, 532
  - in OOIS UML, 532
  - overtaking, 530
  - reply, 532
  - synchronous, 532
  - trace of events, 527
- message passing, 588**
- messaging and notification system, 517**
- metadata in relational databases, 710**
- metamodeling hierarchy, 142**
- methods, 12, 78, 444, 593**
- call of inherited method, 209
  - group specifications, 224
  - hook methods, 552
  - implicit in state machines, 493
  - inheritance, 444
  - in interaction, 526
  - interactive manifestation, 82
  - notation, 78
  - OOIS UML vs. traditional, 594
- in OOP languages, 746
  - vs. operation, 83
  - overriding, 84, 444
  - redefining, 209
  - redefinition, 84, 444
  - reference this, 457
  - resolution policy, 444
  - return, 532
  - setting output parameters, 464
  - side effects, 94
  - specification, 445
  - termination, 463
  - Min in SQL, 717
- mock-up, 658**
- model, 5, 147**
- analysis, 594, 595
  - compiler, 9
  - conceptual. See conceptual model
  - debugging, 46
  - design, 594, 595
  - diagrammatic, 8
  - of a domain, 586
  - executable. See executable model.
  - graphical, 8
  - manifestations of, 46
  - modification at runtime, 60
  - vs. program, 6
  - purpose, 5
  - sequential, textual, 8
  - transformations, 6
  - transformer, 9
  - translator, 9
  - visual, 8
- model compilation, 45**
- model compiler, 9**
- model debugging, 46**
- model editor, 9**
- model elements, 147, 151**
- model library, 44**
- model reuse, 153**
- model transformation, 26**
- model transformer, 9**
- model translator, 9**
- model-centric approach, 147**
- model-driven development, 6**

- model-driven engineering**, 6  
**modeling**, 5  
of behavior, 146  
language. See modeling language  
method, 12  
object-oriented, 25  
paradigm, 9  
of structure, 146  
tools, 9, 722
- modeling by demonstration**  
of creational specifications, 232  
of pattern object specifications, 420
- modeling language**, 6  
desired characteristics, 6  
diagrammatic, 8  
executable, 7  
expressiveness, 6  
graphical, 8  
notation, 8, 147  
reflective, 143  
textual, 8  
visual, 8
- modeling method**, 12
- modeling paradigm**, 9
- modeling tools**, 9  
for relational database, 722
- modifiable association end**, 306
- modifiable attributes**, 254
- modify field**, 705
- Movie data type**, 244
- multi-valued**, 171
- multilevel marketing**  
derived associations, 307  
read-only association ends, 303
- multiple inheritance**, 75, 619  
method resolution policy, 444
- multiple readers-single writer**, 476
- multiplication**, 458
- multiplicity**, 68, 87, 171  
of association actor-use case, 636
- of association ends, 358  
allow relationship, 171  
bounds, 171  
conceptual modeling, 622  
default values, 172  
and existence dependency, 239  
external, 342  
include relationship, 171  
internal, 342  
multi-valued, 171  
of N-ary associations, 335  
notation, 87, 172  
semantics, 173  
single-valued, 171  
static checking in constrained action groups, 361  
unbounded, 172  
validity, 172  
of variables, 175, 358, 447
- multiplicity elements**, 172  
default uniqueness, 173  
implementation in OO programming language, 451  
in OCL, 375  
ordering, 172  
uniqueness, 173
- multiplicity elements**, 171
- multiplicity of attributes**, 248
- multiplicity of composition**, 297
- MultiplicityViolation**  
Exception, 267, 270, 273, 322
- mutual exclusion**, 675
- N**
- N-ary association ends**  
aggregate, 338  
context, 338  
derived, 338  
freezing, 338  
navigability, 338  
ownership, 338  
propagated destruction, 338  
read-only, 338
- redefining, 338  
rules, 338  
static, 338
- N-ary associations**, 68, 331  
actions, 334, 339  
acyclic, 338  
vs. class with N binary associations, 340  
external multiplicity, 342  
internal multiplicity, 342  
links, 333  
multiplicity, 335, 342  
navigation, 340  
notation, 333  
retrieval, 340  
semantics, 333  
use, 342
- N-ary links**, 333
- N-tier architecture**, 680
- name of command**, 548
- named elements**, 155  
accessibility, 161  
associations, 283  
distinguishable, 158  
qualified names, 157  
reference rules, 157  
visibility, 159
- names, length in OOS UML**, 155
- namespaces**, 156  
classifiers as, 183  
constraints, 354  
group specifications, 224  
and inheritance, 157  
member, 164  
members, 156  
name clash with element import, 164  
name uniqueness rules, 158  
as named element, 156  
nesting, 156–157  
in OOS UML, 456  
referring to elements, 157  
without name, 157
- naming conventions in UML**, 156
- navigability**  
association ends, 294

- of N-ary association end, 338  
notation, 294
- navigable association end, actions**, 328
- navigated ends**, 348
- navigation**  
in OQL queries, 403, 407  
pattern object structure, 133  
pictorial, 584
- navigation clause, OQL**, 399, **402**
- neg**, 542
- nesting**  
namespaces, 156, 157  
packages, 153
- network paradigm**, 695
- new**, 457  
for association, 320  
for association class, 347  
for N-ary association, 339  
in OOP languages, 747
- new operator**, 201
- nodes, sub-nodes**, 66
- non-printable characters in OOIS UML**, 155
- NonexistentNamedElement Exception**, 277
- normal forms**, 709
- normalization**, 709
- not operator**, 458  
in OCL, 372
- notation**, 71, **147**  
actors, 632  
association classes, 345  
association ends, 285, 286  
associations, 283  
attributes, 249  
binary associations, 283  
call operation actions, 229  
classes, 184  
constrained action groups, 360  
constraints, 356  
dependencies, 169  
derived association, 309  
derived attributes, 256  
effects, 492  
element import, 164
- entity-relationship, 728  
extend relationship, 641, 642  
formal parameters, 427  
generalizations/specializations, 618  
group specifications, 224  
guards, 491  
GUI context, 101, 103, 104  
GUI item setting, 99  
importing elements, 162  
importing packages, 166  
include relationship, 639  
iteration specifications, 227  
link specifications, 222  
multiplicity, 87, 172  
object reference  
specifications, 221  
object specifications, 221  
operations, 77  
packages, 153  
parameters, 427  
propagated destruction, 238  
pseudostate, 500  
read-only association ends, 306  
read-only attributes, 254  
static operations, 426  
style guidelines, 147  
textual, 55  
transitions, 492  
types, 184  
union association ends, 313  
use cases, 636
- note symbol**, 151
- notEmpty**, 388  
in OCL, 378
- notes. See also comments**  
in pattern object structure, 418  
rendering constraints, 356
- null**, 175, 271, 274, **447**  
in relational database, 704
- O**
- object components**, 567
- object creation action**, 59  
in OOP languages, 747
- Object Data Management Group (ODMG)**, 128
- object destruction**  
in OOP languages, 747  
steps, 238
- object diagram**, 52  
of creational specifications, 217  
for GUI configuration, 101, 103–104  
pattern object structure, 135  
query, 135
- object finalization**, 204
- object identification in GUI**, 581
- object identity**, 53  
primary keys and, 186
- object initialization**, 203  
and read-only association ends, 303  
and read-only attributes, 253
- object member access in OOP languages**, 747
- object paradigm**, 737  
development issues, 36  
key contributions, 743  
traditional development approach, 31  
in UML, 28  
usability, 34  
user interface, 34  
user interface development, 37
- object persistence in OOIS UML**, 57
- Object Query Language (OQL)**, 130
- object reference specifications**, 220  
and conditions, 227  
notation, 221  
to parameter, 220  
*this*, 220
- object references in OOP languages**, 746
- object space**, 51  
in object-to-relational mappers, 32

# object specifications

---

## **object specifications, 221**

conditions, 227  
constraints, 417  
notation, 221  
in pattern object structure,  
    416

## **object structure, 65**

## **object-oriented DBMSs, 60**

## **object-oriented decomposition, 690, 743**

## **object-oriented method, history, 27**

## **object-oriented modeling, 25**

vs. object-oriented  
    programming, 25

## **object-oriented paradigm, 11, 12**

## **object-oriented programming, 745**

deficiencies, 25  
vs. object-oriented modeling,  
    25

## **object-to-relational mapping, 587**

ObjectOfClass **class, 113, 366**

ObjectOfClass **command, 553**

## **Objects, 51, 185, 691, 739**

accessibility, 61  
attribute values, modifying,  
    56

cloning, 187

copy-paste, 187

creating, 52, 56

creation, 200

deleting, 56

destroying, 52

destruction, 204

equality, 189, 192

host, 79, 220

identity, 53, 185–189

instances of classes, 52

interaction, 744

lifetime, 198

link objects. See link object

locking, 475

modeling lifecycle, 485

multiple classification, 619

reclassification, 59

references, in OOP  
    languages, 746

semantics, 53, 191

in user interface, 35

ways to distinguish, 61

working set, 61

## **occurrence specification**

general ordering, 530

in interaction, 526

## **OCL (Object Constraint Language), 88, 354, 369**

@pre, 441

accumulator, 384

addAt, 388

allInstances, 371

any, 384

at, 388

built-in types, 370

casting, 371

collect, 382

collection literals, 386

collection operations, 387

collections, 370, 375, 377,  
    381, 387

comments, 386

conformance, 387

constraint name, 370

context declaration, 369

exists, 384

expressions, 372

forAll, 383

insertAt, 388

invariants, 370

isUnique, 384

iterate, 384

iterators, 381

let, 386

literals, 386

navigation, 370

oclAsType, 371

oclIsKindOf, 371

oclIsTypeOf, 371

oclIsUndefined, 374

OclType, 371

one, 384

OOIS UML dialect, 386

operators, 372

operators in OOIS UML  
    dialect, 386

in OQL WHERE clause,  
    406

postconditions, 441

precedence order, 372

preconditions, 441

reject, 382

reserved words, 374

select, 381

self, 370

sortedBy, 382

string literals, 386

tuple types, 374

tuples, 374

undefined value, 374

variables, 387

oclAsType, 371

OCLConstraint, 90

OCLConstraint **class, 366**

oclIsKindOf, 371

oclIsTypeOf, 371

oclIsUndefined, 374

OclType, 371

**ODBC, 684**

## **ODMG (Object Data Management Group), 130**

**ODTWG, 130**

**OMG, 27, 130**

**OMT, 27**

one in OCL, 384

**OOAD, 27**

**OOIS, 37**

**OOIS UML, 37, 79, 445**

actions, 44, 78

application architecture,  
    586

benefits, 594

block, 456

built-in elements, 45

built-in scalar types, 448

comments, 446

compound statements, 456

concurrency, 438

concurrency control, 474

concurrency model, 472

control flow, 459

- core part, 146  
 declarations, 456, 457  
 design process, 44  
 dichotomies, 141  
**elements, 44**  
 expressions, 457  
 extended part, 146  
 generic OOIS UML GUI, 45  
 vs. Java, 445  
**language, 44, 146**  
 loops, 461  
 method, 594  
 model library, 44, 146  
 namespaces, 456  
 OCL dialect, 386  
 vs. OO DBMSs, 60  
 operation calls, 430, 435  
 operators, 457  
**organization, 44**  
 presentation layer, 563  
 process, 38  
**profile, 44**  
 runtime environment, 44  
 scope, 456  
 semantics, 38  
 statement, 446  
 threads, 438  
 vs. traditional method, 594  
 variables, 175, 447, 456  
 virtual machine, 44
- OOIS UML API**  
 command, 135  
 in Java, 79, 110  
 queries, 135
- OOSE**, 27, 127
- operands of combined fragment**, 536
- operation calls**  
 argument binding, 434  
 asynchronous, 438, 464, 472, 532  
 in constraints, 94  
 effects, 487  
 exceptions, 468  
 in interaction, 526  
 method resolution policy, 444
- null target instances, 435  
 in OOIS UML detail-level language, 430, 435  
 resolution, 430  
 results, 436  
 return values, 464  
 synchronous, 438, 464, 472, 532  
 target instances, 434  
 termination, 464
- operation overloading, 430**, 431  
 distinguishing, 433  
 guidelines, 433  
 resolution policy, 432  
 valid candidates, 432
- operation signatures**, 158  
 matching, 430
- operations**, 76, 423, 742  
 abstract, 78  
 argument passing, 83  
 argument types, 83  
 as behavioral feature, 424  
 call from OQL, 411  
 call from pattern object structures, 419  
 call in constraints, 94  
 call of inherited method, 209  
 call operator, 457  
 call, resolution policy, 444  
 classes vs. data types, 195, 196  
 vs. command, 126  
 constraints, 358  
 data types, 426  
 declaration, 83  
 declaring thrown exceptions, 470  
 distinguishable, 158  
 hiding, 430  
 implementation, 78, 444  
 implementation by state machines, 487  
 interactive manifestation, 82  
**invocation, 429**  
 invocation through reflection, 433  
 invoking, 77
- leafs, 86, 426  
 vs. method, 83  
**methods, 444**  
 as named elements, 156, 423  
 as namespaces, 156, 424  
 notation, 77  
 in OOP languages, 746  
 overloading, 430  
 overriding, 742  
 parameters, 424, 426  
 polymorphism, 84  
 postconditions, 423, 440  
 preconditions, 423  
**preconditions, 440**  
 queries, 392, 426  
 side effects, 195, 196  
 signatures, 83, 423, 470  
 specifying semantics by postconditions, 443  
 static, 392, 424, 488. See also static operations in user interface, 35
- operators**  
% , 458  
% = , 459  
& , 458  
~ , 458  
&& , 458  
&= , 459  
! , 458  
!= , 459  
> , 459  
>> , 458  
>>= , 459  
>>= , 459  
>= , 459  
< , 459  
<< , 458  
<<= , 459  
<= , 459  
| , 458  
|| , 458  
|= , 459  
() (call), 457  
\*, 458

### **operators (*continued*)**

\* =, 459  
+, 458  
++, 458  
+ =, 459  
-, 458  
- >, 457  
--, 458  
- =, 459  
. (dot), 457  
/, 458  
/ =, 459  
=, 459  
==, 459  
? , 459  
% , 458  
% =, 459  
arithmetic, 458  
assignment, 459  
bitwise, 458  
casting, 449, 457  
of combined fragment, 536  
compound assignment, 459  
conditional, 459  
creation, 457  
dot, 457  
equality, 459  
feature access, 457  
grouping, 459

ASC, 409  
base collection of tuples,  
    **402**  
call of query operation, 411  
cumulative functions, 409  
DESC, 409  
DISTINCT, 410  
downcasting, 130, 403  
expressiveness, 137  
FROM, 411  
GROUP BY, 409  
grouping, 409  
inheritance, 131  
inline queries, 413  
navigation clause, 399,  
    **402**  
nested SELECT, 404  
nesting queries, 411  
ORDER BY, 409  
ordering, 409  
output parameters, 407  
parameters, 411  
vs. pattern object structures,  
    135, 137, 416  
polymorphism, 412  
projection clause, 400, **407**  
queries as methods, 411  
query results, 399, 402, 405,  
    407  
query semantics, **399**  
recursion, 412  
result, 131  
SELECT queries, 129, **398**  
selection clause, 400, **405**  
vs. SQL, 128, 129  
tuples, 399  
type specialization, 130  
UNION, 410  
unions, 410  
upcasting, 408  
WHERE, 411  
OQLQuery **class**, 135, 395,  
    413  
or **operator**, 458  
    in OCL, 372  
ORDER BY  
    in OQL, 409  
    in SQL, 718

### **order processing**

association classes, 343  
composite state, 495  
composition, 297  
derived associations, 309  
entry and exit behavior, 502  
enumeration, 242  
history pseudostate, 495  
read-only association end,  
    303  
state machines, 483  
static association ends, 300  
submachine state, 513  
subsetting association ends,  
    310  
unique association classes,  
    345

### **order processing system, bulk input**, 582

#### **ordered**

multiplicity element, 172  
variable, 175, 447

#### **ordered association ends,**     **284, 287**

#### **ordered elements in OCL**, 375

#### **ordered keyword**, 173

OrderedSet  
    in OCL, 375  
operations in OCL, 378

#### **ordering, 172**

conformance, 178  
of creational specification,  
    230

implementation in OO  
    programming language,  
        453

in OCL, 375  
of variables, 175, 447

#### **out**, 427

argument binding, 435  
for query, 393

#### **output parameters**, 433

argument binding, 435  
assignment, 464  
of asynchronous calls, 439  
in OQL, 407  
in pattern object structure,  
    418

- 
- output pins**, 80  
  of commands, 552  
  of GUI component, 574  
  of GUI widget, 563, 575  
**outputPin command**, 552  
**overlapping**, 618  
**overlapping generalization set**,  
  618  
**overriding**  
  methods, 84, 444  
  in OOP languages, 747  
  operations, 742  
**owned members**, 156
- P**
- packageable elements**, 220,  
  **153**  
  associations, 283  
  classifiers as, 183  
  as named element, 155  
**packages**, **153**  
  encapsulation, 159  
  group specifications, 224  
  importing, **165**  
  notation, 166  
  as a namespace, 156  
  nesting, 153  
  notation, 153  
  in OOP languages, 746  
  semantics, 153  
  uses, 153  
  visibility, 159  
**panel components**, 566  
**par**, 538  
**paradigm**, 9  
  object-oriented, 11  
  procedural, 10  
  relational, 10  
**parallelism**, **673**  
**parameters**, **426**  
  argument binding, 434  
  assignment, 179  
  binding by name, 436  
  as a connector, 520  
  in creational specifications,  
    228  
  default multiplicity, 172  
  default uniqueness, 173  
  default values, 428, 430,  
    433, 436, 464  
  direction, 427  
  implementation in OO  
    programming language,  
      454  
  in, 427  
  input, 432, 433  
  as a local variable, 179  
  matching, 430  
  multiplicity, 427  
  names, 426  
  notation, 427  
  object reference  
    specifications, 220  
  of operations, **426**  
  of OQL queries, 411  
  ordering, 427  
  out, 427  
  output, 407, 418, 433, 464  
  of pattern object structure,  
    418  
  returns, 427, 464  
  substitution, 177  
  types, 427  
  uniqueness, 427  
**parents**, 69  
**part-subpart hierarchy**, 238  
**partitioning generalization set**,  
  618  
**pattern matching**, 133  
**pattern object specification by**  
  **demonstration**, 420  
**pattern object structure**, 133,  
  **414**  
  ##, 418  
  actual arguments, 419  
  call of query operation, 419  
  constraints, 417  
  expressiveness, 137  
  formal parameters, 417  
  link specification, 416  
  as method, 418  
  navigation, 133  
  nesting queries, 419  
  object specification, 416  
  vs. OQL, 137, 416  
  output parameters, 134, 418  
  parameters, 418  
  for query object, 418  
  results, 416, 418  
  selection, 134, 417  
  semantics, 414, 416, 417  
  specifications, 414  
  tuple types, 416  
  use, 416  
**patterns. See design patterns**  
**PBD (programming by**  
  **demonstration**), 116  
**PBE (programming by**  
  **example**), 116  
**Persistence**, **672**  
  in OOIS UML, 57, 198  
**personal organizer**, 613  
  abstraction, 613  
  classes and attributes, 613  
  generalizations/  
    specializations, 617  
**physical concurrency**, **673**  
**pictorial navigation**, 584  
**Picture data type**, 244  
**pins**  
  of GUI component, 574  
  of GUI widget, 563, 575  
**pitfall**, 587  
**pointers in OOP languages**,  
  746  
**polymorphism**, **83, 84, 742**  
  benefits, 743  
  in creational specifications,  
    230  
  destructors, 235  
  flexibility, 743  
  interactive manifestation,  
    85  
  in OOIS UML, 444  
  in OOP languages, 747  
  in OQL, 412  
  in user interface, 35  
**portability**, 684  
**post**, 441  
**postconditions**, 358, **440**  
  @pre, 441  
  in OCL, 441  
**power type**, **628**

- pre**, 441  
**preconditions**, 358, **440**  
  in OCL, 441  
**predefined constraints**, 354  
**prepend in OCL**, 379, 380  
**presentation layer**, **563**  
**presentational manifestation**,  
  46  
**primary keys**, **697**  
  composite, 698  
  in entity-relationship, **729**  
  formal definition, 701  
  minimal, **697**  
  nonredundant, **697**  
  and object identity, 186  
  technical ID, 699  
**primitive data types**, 240,  
  **241**  
**primitive keyword**, 242  
**primitive state**, **496**  
  completion, 501  
  entry, 510  
**primitive type in OOIS UML**,  
  448  
**private members in OOP**  
  languages, 748  
**private visibility type**, 159  
**problem domains**, 663  
  vocabulary, 606, **609**  
**procedural decomposition**,  
  **690**  
  in OO decomposition, 744  
  of use cases, 638  
**procedural paradigm**, 10  
**procedures in data flow**,  
  734  
**processes**, 12  
  business. See business  
    process  
  development. See  
    development process  
  in data flow, **734**  
  iterative and incremental. See  
    spiral process model  
  in OOIS UML, 38  
  phases, 685  
  use case-driven, 687  
**products**  
  in OCL, 378  
  in relational algebra, **702**  
**profiles**, 30  
  OOIS UML. See OOIS UML  
**programmatic manifestation**,  
  46  
**programming by contract**,  
  **443**  
  exceptions, 470  
**programming by demonstration (PBD)**,  
  **116**  
  of creational specifications,  
    232  
  of pattern object  
    specifications, 420  
**programming by example**  
  (**PBE**), **116**  
**project management process**,  
  12  
**project risks**, 607  
**projection in relational**  
  algebra, **702**  
**projection clause, OQL**, 400,  
  **407**  
**propagated destruction**, 204,  
  **236**, 340  
  circular, 240  
  of composition, 297  
  declarative vs. imperative,  
    237  
  and existence dependency,  
    239  
  moment of activation, 237  
  of N-ary association end, 338  
  notation, 238  
  recursion break, 240  
  transitive, 238  
**properties**  
  abstractions, 51  
  access through reflection,  
    277  
  actions, 348  
  actions, **325**  
  add value action, 362  
  assignment, 362  
  association ends, 285  
  clear action, 362  
  of collaboration, 521  
  constraints, 357, 358  
  vs. input pin, 549  
  insignificant, 615  
  navigation in OCL, 370  
  navigation in OQL, 403,  
    407  
  in OOIS UML, 63  
  in OOP languages, 746  
  pseudo-, 348  
  read action, 362  
  redefining, 316  
  redefining, **260**  
  in relational paradigm, 696  
  remove value action, 362  
  set action, 362  
  significant, 615  
  static, 300  
  static, **250**  
  in user interface, 35  
**protected members in OOP**  
  languages, 746  
**protected visibility type**,  
  159  
**prototype**, 36  
**prototype design patterns**,  
  114, 549  
  consequences, 625  
  for constraints, 367  
  GUI, 625  
  static operations, 424  
  for type-instance relationship,  
    625  
**prototyping**, 36, 46, 658  
**pseudo-concurrency**, **673**  
**pseudo-properties for**  
  **association classes**, 348  
**pseudostate**, 488, **499**  
  choice, **501**  
  entry points, **511**  
  exit points, **511**  
  history, **498**, **500**  
  initial, **500**  
  notation, 500  
  terminate, **501**  
**public members in OOP**  
  languages, 746  
**public visibility type**, 159

**Q**

**qualified names**, 157  
**quality assurance**, 687  
**queries**, 392, 426  
 actual arguments, 391  
 definition, 394  
 by demonstration, 420  
 vs. derived attributes, 258  
 formal parameters, 391, 392  
 in entity-relationship, 732  
 interactive manifestations, 135  
 invocation, 393  
 as model elements, 392  
 as objects, 134, 394  
 in OOIS UML, 391  
 OOIS UML API, 135  
 operations, 392  
 output parameters in OQL, 407  
 output parameters in pattern object structure, 418  
 pattern object structure, 133–134, 414  
 result, 393, 399  
 resulting tuple types, 391, 393  
 results, 402, 405, 407, 413, 416, 418  
 semantics, 391  
 SQL, 710  
 textual source, 134  
 tuples, 391  
 visual, 133  
**Queries package**, 395  
**Query class**, 134, 395  
**query language**  
 in entity-relationship, 732  
 in relational paradigm, 704  
**query operations**, 426  
 call from OQL, 411  
 call from pattern object structures, 419  
 definition in OQL, 411  
 invocation, 393

method definition, 394  
 results, 393  
**QueryResult**, 394, 413

**R**

**race condition**, 475  
 dangling references, 478  
**rapid prototyping**, 46  
**Rational Software Corp.**, 27  
**read**, 80, 330, 362  
 association class, 348  
 on attributes, 265  
 for link objects, 348  
 for N-ary association, 339  
**read actions**  
 on association ends, 330  
 on attributes, 265, 279, 362  
 on properties, 362  
 on variable, 362  
**read association end action**, 325  
**read attribute actions**, 263, 264  
**read link actions**  
 for association classes, 348  
 for N-ary associations, 334  
 navigability, 294  
 vs. read link object action, 350  
**read link object actions**, 348  
 cardinality, 350  
 multiplicity, 350  
 ordering, 350  
 vs. read link action, 350  
 uniqueness, 350  
**read property action**, 325  
 OOIS UML API, 79, 110  
**read size actions on attributes**, 264  
**read value actions on attributes**, 265, 266  
**read-only association ends**, 303  
 notation, 306  
 of N-ary association, 338  
 semantics, 303, 307  
**read-only attributes**, 252

of class, non-static, 253  
 notation, 254  
 semantics, 255  
 unfreezing, 276  
 writing, 268  
**readOnly modifier**, 254, 306  
**Real in OCL**, 373  
**Real data type**, 244  
**real-time systems**, 673  
**realization dependency**, 170  
 for collaboration use, 523  
**realizations**, 141  
**record sets**, 710  
**records**, 696  
**recovery point**, 478  
**recursion**  
 in creational specifications, 230  
 in OQL, 412  
**redefinable elements**, 248  
**redefines string**, 316  
**redefining**  
 association ends, 316, 317  
 attributes, 260  
 interactions, 530  
 methods, 84, 444  
 notation, 262, 318  
 in OOIS UML, 261, 318  
 semantics, 260, 318  
**redefining association ends of N-ary associations**, 338  
**redefining parameters**, 99  
**redefining settings**, 101  
**redefinition of state machines**, 489  
**ref**, 543  
**reference sharing**, 198  
**reference-based equality comparison**, 192  
**references**  
 in OOP languages, 748  
 variables in UML, 174, 446  
**referential integrity**  
 cascading deletion, 707  
 cascading update, 70  
 in relational paradigm, 707  
**referring to named elements**, 157

- refine dependency**, 170  
**refine keyword**, 170  
**Reflection**, 143, 202, 321,  
  323  
  access to slots, 277  
  link creation, 321  
  link destruction, 323  
  object creation, 201  
  operation calls, 433  
**reflective association classes**,  
  349  
**reflective associations**, 314  
  creation of links, 320  
**reflexivity of type**  
  **conformance**, 178  
**reject in OCL**, 382  
**relation**, 701  
**relational algebra**, 701  
  joins, 703  
  products, 701  
  projection, 702  
  selection, 702  
**relational databases**  
  authorization, 722  
  concurrency control, 721  
  distribution, 721  
  transactions, 722  
**relational model. See also**  
  **relational paradigm**  
  normal forms, 709  
  normalization, 709  
  views, 706  
**relational operators in OCL**,  
  372  
**relational paradigm**, 10, 15,  
  16, 695, 696  
  action language, 704  
  actions, 703  
  advantages, 19  
  basic concepts, 696  
  behavior, 703  
  conceptual modeling, 16  
  data manipulation actions,  
    **703**  
  data-manipulation language,  
    704  
  development issues, 19  
  disadvantages, 20  
  discontinuities, 21  
  drawbacks, 16  
  and entity-relationship, 732  
  formalism, 702  
  generic operations, 703  
  joins, 700  
  query language, 704  
  referential integrity, 707  
  structure, 703  
  surface language, 704  
  usability, 17  
  user interface, 17  
  user interface development,  
    22  
  views, 706  
**relationship sets**, 728  
  vs. association, 68  
**relationships**  
  between use cases, 637  
  in entity-relationship, 727  
  in ER, mapping to relational  
    model, 733  
  extend, 640. *See also* extend  
    relationship  
  generalization/specialization,  
    642  
  include, 637  
  include. *See also* include  
    relationship  
  implementation in OOP  
    language, 26  
  in relational paradigm, 701  
  in user interface, 35  
**relink actions on association**  
  **end**, 330  
**remainder**, 458  
**remote procedure call (RPC)**,  
  588, 682  
  remove, 362  
    on attributes, 269  
**remove field**, 705  
**remove table**, 705  
**remove value action**  
  on attributes, 269–270, 362  
  on properties, 362  
  on variables, 362  
  removeAt, 362  
    on attributes, 270  
  removeFirst, 362  
    on attributes, 270  
  removeLast, 362  
    on attributes, 270  
  removeOne, 362  
    on attributes, 269  
**reorder actions**, 329  
  on association ends, 329  
**replace action**, 330  
  on association end, 330  
**reply message**, 532  
**reports, development**, 723  
**requirements analysis**, 598  
**requirements capturing**, 598  
  addressing ambiguities, 611  
**requirements engineering**,  
  594, 598, 609. *See also*  
  conceptual modeling  
  activities, 598  
  artifacts, 600  
**requirements specification**,  
  598, 600  
**requirements specification**  
  **document**, 600–601  
  central part, 606  
  closing part, 607  
  domain description, 604  
  internal conventions, 604,  
    613  
  introductory part, 603  
  non-technical requirements,  
    607  
  purpose, 601  
  specifying classes, 612  
  structure, 602  
  style, 602  
  system's purpose, 605  
  system's scope and  
    boundaries, 605  
  technical requirements, 606  
**responsibilities**  
  of classes, 613  
  distribution, 744  
**resulting tuple types of**  
  **queries**, 391, 393  
**retrieve records**, 705  
  return, 532  
    argument binding, 435

- parameters, 427, 464  
**return statement**, 463  
**reusability**, 688  
**reuse of models**, 153  
**reverse engineering**, 32  
**RichText data type**, 244  
**risk management**, 658, 687  
**roles**, 282, 631
  - of associations, 62
  - of classes, 613
  - in collaboration, 520–**521**
  - in entity-relationship, **728**
  - in relational paradigm, 702**rollback**, 478, 479, 683  
**round-trip engineering**, 32  
**rows**, 696  
**RPC (remote procedure call)**, 588, 682  
**Rumbaugh, J.**, 27  
**run-to-completion**, 506  
**runtime environment**, 44  
**runtime semantics**, 46  
**rush-to-code syndrome**, 22
  - in OO development, 32
- ## S
- scaffolding**, 37  
**scalability**, 671  
**school timetable**, 340
  - multiplicity, 335
  - N-ary associations, 331**scope in OOIS UML**, 456  
**scope discontinuity**, 20, 36
  - in OO approaches, 33**scope resolution rules in UML namespaces**, 157  
**sd**, 524, 543  
**search components**, 566  
**searches for objects in GUI**, 580  
**security**, 668  
**SELECT**
  - navigation, 407
  - nesting in OQL, 404
  - in OCL, 381
  - in OQL, **398**
  - semantics in OQL, **400, 407****in SQL**, **712**  
**SELECT queries in OQL**, 129  
**select records**, 705  
**selection in relational algebra**, **702**  
**selection clause**
  - OQL, 400
  - OQL, **405****self**, 366
  - in constraints, 354, 357
  - in interactions, 526
  - in OCL constraints, 89, 90**self reserved word**, 370  
**semantic discontinuity**, 21, 36
  - in OO approaches, 33
  - UML, 29**semantics**
  - dependencies, 168
  - derived, 144
  - dynamic, 46
  - executable, 36
  - importing elements, 162
  - of OOIS UML, 38
  - runtime, 46
  - of UML, 29, 30**Sequence in OCL**, 375, 380  
**sequence diagram**, 519, 524
  - messaging system, 519**sequencing of creational specifications**, 230  
**servers. See application server; database server**  
**service**, 588, **682**
  - Web, **682****service-oriented architecture (SOA)**, 588, 682  
**sessions in OOIS UML**, 472  
**set**, 279, 362
  - on attributes, 271
  - in OCL, 375
  - operations in OCL, 378**set action**
  - on attribute, 271, 279, 362
  - on property, 362
  - on variable, 362**{seq} string**, 285  
**{sequence} string**, 285  
**shared variables**, 588  
**short type**, 448  
**side effects**, 195, 196  
**Simula**, 186  
**single-valued**, 171  
**singleton design pattern**, 301
  - static operations, 424**size**, 80
  - in OCL, 377
  - on attributes, 264**sizing**, 75  
**slot**, 183, 277, **278**
  - of property, 63**Slot command**, 554  
**slot components**, 566  
**Slot type**, 548  
**slots**, 53, 247
  - access through reflection, 277
  - attributes and, 64
  - feature access operator, 457
  - semantics, **249****SOA**, 588, **682**  
**SOAP**, 588, **682**  
**software**
  - architecture, 687
  - flexibility, 688
  - maintenance, 688
  - reuse, 688**sortedBy in OCL**, 382  
**Sound data type**, 244  
**source class model**, 213  
**source domains**, 213  
**source object structure**, 213  
**source/derived information trade-off**, 309, 669  
**specialization**, 616. *See also generalizations/specializations*
  - in entity-relationship, **730**
  - mapping to relational model, 733
  - in user interface, 35**specifications**, 141  
**spiral process model**, **686**  
**sporadic actions**, 673  
**sport competition**
  - multiplicity, 335
  - N-ary associations, 331

# SQL (Structured Query Language)

---

## **SQL (Structured Query Language), 705, 710**

\* (asterisk), 715  
aggregates, **717**  
aliases, 713  
AS, 713  
ASC, 718  
Avg, 717  
built-in functions, 716  
case sensitivity, 711  
Count, 717  
data modification statements, **720**  
**DELETE**, 704, **720**  
**DESC**, 718  
**DISTINCT**, **713**  
**FROM**, **713**  
**GROUP BY**, 718  
**INSERT**, 704, **720**  
**LIKE**, **716**  
literals, 716  
Max, **717**  
Min, **717**  
nested queries, 714  
operators, 716  
**ORDER BY**, 718  
queries, 710  
record sets, 710  
**SELECT**, **711**  
select statement, **711**  
statement types, 710  
statements, 710  
sub-expressions, 716  
Sum, 717  
**UNION**, **716**  
**UPDATE**, **720**  
**UPDATE**, 704  
vs. OQL, 128, 129  
**WHERE**, **715**  
**SSA, scope discontinuity**, 20, 21  
StartTrans, 480  
**state**, 485, **488**, 489, 496  
active, 489, **496**, 504  
active state configuration, 504  
completion, 501  
composite state, **496**

current state, 489, 493, 496  
default entry, 509  
entering, 509  
entry behavior, **503**, 509  
entry points, **511**  
event processing, 505  
exit behavior, **503**, 508  
exit points, **511**  
exiting, 508  
final, 486, **501**  
freezing, 493, 506  
history pseudostate, **498**  
initial, 486, 509  
initial value, 493  
invariant in interaction, 535  
least common ancestor, 508  
multiplicity, 493  
as namespace, 489  
notation, 504  
primitive state, **496**  
pseudostate, 488, **499**  
querying, 489, 496  
redefinition, 513  
refining, 496  
submachine, **513**  
type, 493  
visibility, 493  
**state diagram**, 486  
**state invariant**, **535**  
notation, 536  
semantics, 536  
**state machines**, 483  
completion, 501  
conditional branch, 501  
context, 487  
current state, 493  
dispatching events, 506  
entry points, **511**  
execution, 505  
exit points, **511**  
GUI, 515  
inheritance and redefinition, 489  
as namespaces, 489  
in OOIS UML, 487  
refining states, 496  
reuse, 515  
semantics, **493, 504**

in standard UML, 487  
submachine state, **513**

triggering, 515

vertex, 488

visibility, 493

visualization, 516

## **statements**

break, 461, **463**

continue, **463**

do-while, **461**

for, **462**

## **if**, **459**

**in OOIS UML**, 446, 459

label, 463

return, **463**

switch, **461**

while, **461**

**static, N-ary association end**, 338

## **static association ends**, 300

access, 302

actions, 302

inheritance, 302

initialization, 302

modeling rules, 301

navigability, 301

notation, 302

ownership, 301

semantics, 301

use, 302

## **static attributes**, 248, **250**

inheritance, 251

initialization, 251

references to, 251

semantics, 251

uses, 251

**static checking in constrained action groups**, 361

## **static operations**, 392, 488

vs. global procedures, 424

invocation, 429

notation, 426

target instances, 434

## **static operations**, **424**

## **static typing**, **449**

## **stick man notation**, 632

## **stored procedures**, **707**

vs. derived attributes, 258

- strict**, 539  
**string literals in OCL**, 386  
**strings as a name in UML**, 155  
**strong conformance**, 362  
**structural features**, 247  
  class vs. data type, 193  
  classifiers, 183  
  in OOIS UML, 193  
**structure**, 146  
  of classifier instances, 193  
  design, 744  
**structure diagram**, 150  
**structured system analysis**,  
  **727**  
  data flow, **734**  
**sub-collection**, 312  
**sub-contexts**, 101  
**sub-nodes**, 66  
**sub-settings**, 99  
**subclasses**, 69  
**subject, use cases**, 631, 635  
**submachine state**, **513**  
  entry, 515  
  exit, 515  
  notation, 515  
**subOrderedSet in OCL**, 379  
**subSequence in OCL**, 380  
**subsets keyword**, 310  
**subsetting association ends**,  
  **310**  
  modeling rules, 311  
  semantics, 312  
**substitution**, 176  
  in OO programming language,  
  452  
**substitution rule**, **70**, 742  
**subtraction**, 458  
**subtyping**, **741**  
  design of hierarchies, 744  
  in OOP languages, 747  
**Sum**  
  in OCL, 378  
  in OQL, 409  
  in SQL, 717  
**superclasses**, 69  
**suppliers of dependency  
  relationship**, 168  
**super keyword**, 209  
**surface language**, 59, 79, 81  
  in entity-relationship, 732  
  in relational paradigm, 704  
  SQL, 710  
**switch**, **461**  
**symbols**, 147  
**symmetricDifference in  
  OCL**, 378  
**synchronous message**, 532  
**synchronous operation calls**,  
  438, 532  
**syntax**  
  abstract, 44, 147  
  concrete, 44, 147  
**system**, 3  
  computer-based, 4  
  information system. See  
    information systems  
  prototyping, 658  
**system analysis, specifying  
  operation semantics**, 443  
**system analyst**, 598, 665  
**system architecture**, **676**  
  client/server, **677**  
  distributed, **681**  
  mainframe, **677**  
  N-tier, **680**  
  patterns, **676**  
  thick clients, 679  
  thin clients, 679  
  three-tier, **679**  
**systems**  
  distributed, 676  
  interactive, 667  
  real-time, 673  
  time-driven, 673  
  transformational, 667
- T**
- Tables**, **696**  
  columns, 696  
  formalism, 701  
  rows, 697  
**target class model**, 213  
**target domains**, **213**  
**target instances**, 434  
  null, 435, 459  
reference this, 457  
**target object structure**, **213**  
**technical ID**, 699  
**template method design**  
  pattern, 551  
  for queries, 397  
**templates**, 689  
  in OO programming language,  
  451  
terminal, 677  
**terminate pseudostate**, **501**  
**testing**, 594  
  using interactions, 530  
**Text data type**, 244  
**textual notation**, 55  
**theater booking**, 305  
**thick clients**, **679**  
**thin clients**, **679**  
this, 79  
  in OOIS UML, 457  
  in OQL queries, 412  
  in queries, 393  
**this object reference  
  specification**, 220  
**threads**, 472  
  in OOIS UML, 438  
  in OOIS UML, **472**  
**three amigos**, 27  
**three-tier architecture**, **679**  
  in OOIS UML, **586**  
**throw**, **468**  
**Throwable**, 467  
**Time data type**, 244  
**time-driven systems**, 673  
**TimeInterval data type**, 244  
**timeliness**, 668  
**timetable, N-ary associations**,  
  **331**  
**toolbars, browser commands**,  
  **124**  
**trace dependency**, 170  
**trace keyword**, 170  
**trace of events**, **527**  
  alternative, 536  
  coregion, 538  
  critical region, 538  
  general ordering, 530  
  interleaving, 529, 538

### trace of events (*continued*)

invalid, 534, 542  
loop, 540  
optional, 537  
parallel, 538  
reuse, 543  
strict sequencing, 539  
trans, 479  
Transaction, 479  
**Transactions**, **479, 683**  
of commands, 551  
commit, 479  
nesting, 479  
in OOIS UML, 479  
in other languages, 479  
in relational databases, 722  
**transformation**, **213**  
**transformational systems**, 667  
**transformer**, 9  
**transitions**, **486, 488**  
completion, 501  
to a composite state, 496  
compound, **501**, 507  
conflict, 507  
conflict resolution, 507  
effects, 491  
enabled, 507  
execution, 493, 507, 509,  
    513  
firing, 493  
from a composite state, 498  
group, **504**  
guards, 491  
high-level, **504**  
from a history pseudostate,  
    500  
to a history pseudostate, 500  
internal, **503**, 507  
least common ancestor, 508  
notation, 492  
priority, 507  
selection, 507  
**transitivity of type**  
    **conformance**, 178  
**translator**, 9  
**tree view**, 18  
**tree view components**, 566

**trees, by creational specification**, 230  
**triggers**, **486, 488, 707**  
    vs. derived attributes, 258  
try, 466  
    backward error recovery, 479  
    forward error recovery, 478  
**Tuple**  
    for query results, 394, 413  
    in OCL, 374  
    in operation calls, 433  
    results of operation calls,  
        436  
**tuple types**  
    of inline queries, 413  
    in OCL, **374**  
    in operation calls, 433  
    of pattern object structure,  
        416  
    of queries, 391, 393  
    result of operation call, 433  
    results of operation calls,  
        436  
**tuples**  
    base collection in OQL query,  
        **402**  
    in OCL, **374**  
    in operation calls, 433  
    in relational paradigm, **696**  
    of OQL queries, **399**  
    of queries, **391**  
    result of operation call, 433  
    result of OQL queries, 402,  
        405  
    results of inline queries, 413  
    results of operation calls,  
        436  
    result of OQL queries, 399  
    results of OQL queries, 407  
    results of pattern object  
        structure, 416, 418  
    results of queries, 393  
**TupleType**  
    in OCL, 375  
    for query results, 393  
    results of operation calls,  
        436

**two-way command mapping**,  
    **114**  
**type**, **738**  
    conformance, **178**  
**type casting operators**, 457  
**type conformance, violation**,  
    449  
**type-instance dichotomy**, 52,  
    60  
    horizontal, 628  
    power type, 629  
    vertical, 627  
**type-instance relationships**  
conceptual modeling, 623  
modeling with association,  
    627, 628  
modeling with attributes, 624  
modeling with class  
    hierarchy, 626  
using prototypes, 625  
**type-instances, attributes**, 54  
**typed elements**, **174, 447**  
equality, 189, 191  
substitution, 176  
variables, 174, 447  
**types**  
and abstraction, 738  
of association ends, 193  
of attributes, 193, **248**  
built-in, 738  
casting, 449  
casting operators, 457  
classifiers as, 183  
conversion, 457, 459  
encapsulation, 738  
of exception, 467  
expressions, 459  
static typing, 449  
user-defined, 738  
of variables, 447

## U

**UML (Unified Modeling Language)**, **11, 27**  
2.0, 27  
action semantics, 27  
advantages, 28

- applicability, 28  
 character set, 155  
 complexity, 28  
 customization, 28  
**diagram, 147**  
 behavioral, 150  
 types, 150  
 structural, 150  
 style guidelines, 148  
 dichotomies, 141  
 discontinuities, 30  
 drawbacks, 28  
 and entity-relationship, 15  
 extensibility, 28  
 extensions, 30  
 history, 27  
 and implementation  
     languages, 29, 31  
 as a language, 147  
 learning curve, 28, 30  
 notation, **147**  
 popularity, 28  
 profile for OOS. See OOS  
     UML  
 profiling, 30  
 semantic discontinuity, 29  
 semantics, 29, 30  
 standardization, 28  
 versions, 27  
 XMI, 147
- UML analysis model**, 594, **595**
- UML design model**, 594, **595**
- UML diagram**, **147**  
 behavioral, 150  
 structural, 150  
 style guidelines, 148  
 types, 150
- unary operators in OCL**, 372
- unbounded multiplicity**, 172  
 unfreeze, 276, 307  
 unfreeze **attribute**, 255  
**unfreezing attributes**, 275
- Unicode**, 448  
 and UML, 155
- Unified Method**, 27
- Unified Modeling Language.**  
**See UML**
- UNION**  
 in OCL, 378–380  
 in OQL, 410
- union association ends**, **312**  
 modeling rules, 313  
 notation, 313  
 semantics, 313
- union queries**, 716
- unions in OQL**, 410
- unique**, 173  
 variable, 175, 447
- unique constraint**, 188
- UniqueCollection**, 454
- unordered keyword**, 173
- unrestricted modifier**, 254
- upcasting**, 450  
 in OCL, 371  
 in OQL, 408
- UPDATE in SQL**, **720**
- update records**, 704
- upper bound**, 171
- UpperBoundViolationException**,  
 267, 270, 272–273, 322
- usability**, 669  
 of object paradigm, 34  
 of relational paradigm, 17
- usage dependency**, 169
- use**, 169
- use case diagram**, **636**
- use cases**, **633**, 636  
 actors, association with, 635  
 as functional requirement,  
     634
- ATM, 633
- business, 651  
 vs. business processes, 650
- classifiers, 635
- complexity, 637
- CRUD analysis, 654
- discovering, 651
- engineering, 651
- extend relationship, 640
- features, 636
- generalization/specialization,  
     642
- goals, 635
- implementation, 634
- include relationship, 637
- interaction diagrams, 645  
 management, 646  
 names, 635  
 notation, 636  
 ownership, 636  
 and planning iterations, 658  
 postconditions, 645  
 preconditions, 645  
 relationships, 637  
 scenarios, 634  
 specification, 644  
 specification template, 644  
 subject, 631, 635  
 triggers, 645  
 variations, 635
- user access rights**, 668  
 user interface  
 in object paradigm, 34, 37  
 in relational paradigm, 17, 22  
 usability, 669  
 user management, 238  
 acyclic associations, 314  
 composition, 297  
 derived associations, 309  
 redefined properties, 317  
 subsetting association ends,  
     310
- user-defined constraints**, 354
- user-defined types**, 241, 244,  
 738
- users**, 598

**V**

- val on attributes**, 265
- valid candidates in operation overloading**, 432
- Value command**, 554
- value-based classification**,  
 624
- value-based equality comparison**, 191
- value-based identity**, 53
- variables**  
 actions, 175  
 add value action, 362  
 assignment, 179, 362  
 clear action, 362

### **variables (*continued*)**

as a connector, 520  
dangling references, 478  
declarations, 175, 447, 457  
default value, 449  
feature access operator, 457  
implementation in OO  
    programming language, 451  
initialization, 448  
multiplicity, 175, 358, 447, 449  
null, 175  
in OCL, 387  
in OOIS UML, 175, 447, 456  
ordering, 175, 447  
of primitive type, 449  
read action, 362  
as references, 174, 446  
remove value action, 362  
scope, 457  
set action, 362  
substitution, 177  
types of, 174, 447  
uniqueness, 175, 447  
**variables, 174, 446**  
**view-centric approach**, 147

### **views, 706**

as data separation layer, 706  
updatable, 706  
vs. derived attributes, 258

### **virtual machine, 44**

### **visibility, 159**

in element import, 162

### **vocabulary of domain, 606, 609**

## **W**

### **waterfall process model, 685**

drawbacks, 685

### **weak conformance, 362**

### **weak entities, 729**

existence dependence, 730

identification dependence,  
729

### **Web browsers, 679**

### **Web GUI vs. desktop GUI, 569**

### **Web servers, 680**

### **Web service, 682**

#### **WHERE**

in OQL, 411

semantics in OQL, 400, 405

#### **while, 461**

### **whole-part relationship, 296**

### **workflow management**

derived unions, 312  
redefined properties, 318

### **working set, 61, 673**

### **write association end action, 326**

### **write attribute actions, 263, 268**

### **write attribute value actions**

as attribute value  
specifications, 222

### **write property action, 326**

#### **WritingReadOnly**

AssociationEndException,  
303, 306

#### **WritingReadOnlyAttribute**

Exception, 253, 255,  
268

### **WYSIWYG, 723**

## **X**

### **XMI, 147**

### **XML, 147, 680, 682**

#### **xor constraint, 354**

#### **xor operator in OCL, 372**



# Take your library wherever you go.

Now you can access more than 200 complete Wrox books online, wherever you happen to be! Every diagram, description, screen capture, and code sample is available with your subscription to the **Wrox Reference Library**. For answers when and where you need them, go to [wrox.books24x7.com](http://wrox.books24x7.com) and subscribe today!

## Find books on

- ASP.NET
- C#/C++
- Database
- General
- Java
- Mac
- Microsoft Office
- .NET
- Open Source
- PHP/MySQL
- SQL Server
- Visual Basic
- Web
- XML

# Model-Driven Development with Executable UML

Motivated by the viewpoint that software development efficiency can most certainly be improved, author Dragan Milicev proposes a technique that raises the level of abstraction and reduces accidental complexity in software development. This comprehensive reference focuses on an executable profile of UML and a method that allows you to save significant time and avoid human error by using modeling. Serving as an in-depth tutorial on model-driven development, this book places special emphasis on information systems and describes in detail just what is meant by this term. You'll discover how information systems can be understood better and developed more efficiently by using the object paradigm, model-driven development, and a profile of UML that is formal and executable. Helpful examples demonstrate the development process and show how to use UML for building information systems. An extensive overview describes the fundamental concepts of object orientation, and further discussions review the full coupling of object orientation with information systems development and how challenges with this development can be handled.

## What you will learn from this book

- The characteristics of information systems, and process models for their development
- Advantages and common problems with popular technologies for building information systems and how they affect development
- An introduction to the object paradigm as applied to building information systems
- Concepts and parts of UML that are likely needed in building information systems

## Who this book is for

This book is for software practitioners who analyze, specify, design, model, develop, or test information systems and are interested in improving their knowledge and productivity by exploiting model-driven rapid application development with an executable UML.

**Wrox guides** are crafted to make learning programming languages and technologies easier than you think. Written by programmers for programmers, they provide a structured, tutorial format that will guide you through all the techniques involved.

p2p.wrox.com  
The programmer's resource center

|                                      |                      |
|--------------------------------------|----------------------|
| Recommended Computer Book Categories | Software Development |
|                                      | General              |

\$59.99 USA  
\$71.99 CAN

**Wrox**  
An Imprint of  
 WILEY

[www.wrox.com](http://www.wrox.com)

