



Object-Oriented Analysis and Design for Information Systems

Modeling with UML, OCL, and IFML

MK
MORGAN KAUFMANN

Raul Sidnei Wazlawick

Object-Oriented Analysis and Design for Information Systems

Modeling with UML, OCL, and IFML

Raul Sidnei Wazlawick



AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann is an imprint of Elsevier

Table of Contents

Cover image

Title page

Copyright

Dedication

Acknowledgments

About the Author

Foreword

Preface

Chapter 1. Introduction

Key Topics in this Chapter

1.1 This book

1.2 Object-oriented systems development

1.3 Unified Modeling Language (UML)

1.4 Unified Process (UP)

1.5 The process so far

1.6 Questions

Chapter 2. Business Modeling

Key Topics in this Chapter

2.1 Introduction to business modeling

2.2 General view of the system

2.3 Business use cases

2.4 Business activity diagram

2.5 State-dependent aspects of a business

2.6 Remarks

2.7 The process so far

2.8 Questions

Chapter 3. High-Level Requirements

Key Topics in this Chapter

3.1 Introduction to high-level requirements

3.2 System actors

3.3 System use cases

3.4 How to find system use cases in the business model

3.5 Requirements

3.6 Preliminary conceptual model

3.7 The process so far

3.8 Questions

Chapter e4. Use Case Based Project Planning

4.1 Introduction to effort estimation and risk analysis in software projects

4.2 Use case point analysis

4.3 Planning an iterative project

4.4 The process so far

4.5 Questions

Chapter 5. Expanded Use Cases

Key Topics in this Chapter

5.1 Introduction to expanded use cases

5.2 Main flow

5.3 Alternate flows

5.4 Writing recommendations

5.5 Included use cases and fragments

5.6 Expansion of stereotyped use cases

5.7 Other sections of an expanded use case

5.8 System sequence diagrams

5.9 The process so far

5.10 Questions

Chapter 6. Conceptual Modeling: Fundamentals

Key Topics in this Chapter

6.1 Introduction to conceptual modeling

6.2 Attributes

6.3 Concepts

6.4 Associations

6.5 Collections

6.6 Organization of the conceptual model

6.7 Invariants

6.8 Iterative construction of the conceptual model

6.9 The process so far

6.10 Questions

Chapter 7. Conceptual Modeling: Patterns

Key Topics in this Chapter

7.1 Introduction to conceptual model patterns

7.2 High cohesion

7.3 Specification classes

7.4 Quantity

7.5 Measure

7.6 Strategy

7.7 Composite

7.8 Organizational hierarchy

7.9 Object joining

7.10 Account/Transaction

7.11 Range

7.12 Temporal patterns

7.13 Discussion

7.14 The process so far

7.15 Questions

Chapter 8. Functional Modeling with OCL Contracts

Key Topics in this Chapter

8.1 Introduction to functional modeling

8.2 Preconditions

8.3 Transient associations

8.4 Query return

8.5 Postconditions

8.6 Exceptions

8.7 Pattern contracts for CRUD

8.8 Pattern contracts for listing objects

8.9 Contracts related to use cases

8.10 The process so far

8.11 Questions

Chapter 9. Domain Tier Design

Key Topics in this Chapter

9.1 Introduction to domain tier design

9.2 Object responsibility distribution

9.3 Visibility

9.4 Dynamic modeling based on postconditions

9.5 System queries

9.6 Delegation and low coupling

9.7 Design class diagram

9.8 The process so far

Chapter 10. Code Generation

Key Topics in this Chapter:

- 10.1 Introduction to code generation
- 10.2 Classes and attributes
- 10.3 Unidirectional associations
- 10.4 Bidirectional associations
- 10.5 Delegated methods and system operations
- 10.6 Patterns for filtered queries
- 10.7 The process so far
- 10.8 Questions

Chapter 11. Testing

Key Topics in this Chapter

- 11.1 Introduction to testing
- 11.2 Functional testing
- 11.3 Stubs and drivers
- 11.4 Test-driven development
- 11.5 Unit testing
- 11.6 System operations testing
- 11.7 Use case testing (System, acceptance, and business cycle tests)
- 11.8 The process so far
- 11.9 Questions

Chapter 12. Interface Tier Design with IFML

Key Topics in this Chapter

- 12.1 Introduction to interface tier design
- 12.2 Interaction flow modeling language (IFML)
- 12.3 View components
- 12.4 Pages
- 12.5 Flows

12.6 Hypertext organization

12.7 Web Interface Patterns

12.8 Modeling operations in the interface

12.9 IFML models for CRUD operations

12.10 Use case interface modeling with IFML

12.11 The process so far

12.12 Questions

Chapter e13. Data Persistence

Key Topics in this Chapter

13.1 Introduction to data persistence

13.2 Object-relational mapping (ORM)

13.3 Saving and loading objects

13.4 The whole process

13.5 Questions

Postface

References

Index

Copyright

Acquiring Editor: Andrea Dierna
Editorial Project Manager: Kaitlin Herbert
Project Manager: Punithavathy Govindaradjane
Designer: Matthew Limbert

Morgan Kaufmann is an imprint of Elsevier
225 Wyman Street, Waltham, MA, 02451, USA

© 2013 Elsevier Editora Ltda.
Copyright © 2014 Elsevier Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: www.elsevier.com/permissions.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors,

contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data

Application submitted

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

ISBN: 978-0-12-418673-6

Printed in the United States of America

14 15 16 17 18 10 9 8 7 6 5 4 3 2 1

For information on all MK publications visit our website at www.mkp.com



Working together
to grow libraries in
developing countries

www.elsevier.com • www.bookaid.org

Dedication

This book is dedicated to my parents and ancestors; without them I would not exist.

Acknowledgments

I wish to thank some people that in some way helped me to make this book possible: Prof. Luiz Fernando Bier Melgarejo, for introducing me to the world of object-oriented systems back in 1987; my colleague Prof. Marcos Eduardo Casa, for all his work in developing pair programming back in the 1980s when object-oriented concepts were regarded as “something from another planet”; my colleague Prof. Antonio Carlos Mariani, for developing the Actors World, a tool that helped us teach object-oriented programming; the companies that allowed the use of the techniques explained in this book in a real production environment; my advisees Everton Luiz Vieira, Kuesley Fernandes do Nascimento, and Iuri Cardoso for helping me consolidate some of the new techniques; the Department of Informatics and Statistics (INE) of the Federal University of Santa Catarina (UFSC), for the opportunity to develop this work; and especially software engineer Gilmar Purim, for the fruitful discussions that helped to define the shape of the first version of this book in 2003.

I wish to thank also the more than one thousand students, who I refrain from nominating, victims of my course on Object-Oriented Analysis and Design for two decades – their doubts, misunderstandings, and difficulties motivated me to research and learn even more; my friend, Prof. Rogério Cid Bastos, for providing orientation and motivation; and finally my friends and brethren, for the moments of joy and fraternity.

About the Author

Raul Sidnei Wazlawick is an Associate Professor of Computer Science at UFSC, Florianópolis, Brazil. He has received a Bachelor's and Master's in Computer Science, and Doctor in Engineering (1993). Wazlawick was chair of the IFIP Working Group on Informatics and ICT in Higher-Education, trustee of the Brazilian Computer Society, general chair of the Brazilian Symposium on Software Engineering (2006), and co-chair of the IFIP World Conference on Computers in Education (2009). He has published three textbooks in Portuguese and about 100 papers. His field of experience includes object-oriented systems, with which he has worked since 1986. Wazlawick's main research interest is object-oriented software engineering. He is currently (2012–2015) heading a software development team that is developing software systems to support the Brazilian public health system.¹ He often makes presentations at universities, companies, and government institutions on topics related to software engineering. He also has worked as a software engineering consultant since the early 1990s.

¹<http://dab.saude.gov.br/portaldab/esus.php>.

Foreword

Roger S. Pressman, Ph.D.

Because things are often fuzzy at the beginning of a software project, because it is sometimes difficult to get stakeholders to agree, because required information and functions are often unclear, because customers have no compunction about requesting systems with frightening complexity and staggering size, and because it's so tempting to assume that we already know what is required, there remains a compelling need for good books that teach software engineers a pragmatic, proven approach to analysis and design. Raul Wazlawick provides us with such a book.

Over the past 50 years we have seen many methods proposed for analysis and design of computer software, but in my view, the most intuitive and effective adopts an object-oriented mindset. Remember, analysis and design occur relatively early in the software process at a time when things are fluid, where iteration is not only common, but mandatory, and where “objects” are often the most visible of all elements of the problem.

The beauty of the object-oriented paradigm is that it is wonderfully elastic. We can describe objects that are easily understood by business people at the conceptual level, and then through a process of iteration and elaboration, refine those objects into much lower levels of abstraction that technical people can use effectively. In this book, Professor Wazlawick provides us with guidance for accomplishing this.

Using UML as his notational form, he begins where all good analysts should start—the beginning! He provides us with the tools required to understand the problem through a process of elicitation, elaboration, and representation. He emphasizes use cases, and helps us understand that they serve as both an excellent mechanism for problem understanding and as a mechanism that can greatly assist project planning.

He insists that good analysts and good designers must iterate as they develop an increasingly better and more detailed understanding of the problem. To accomplish this, he provides the reader with object-oriented techniques and the UML tools to get that job done.

Through the book, a continuing case study is used to provide examples of all analysis and design techniques. Basic principles are introduced, expanded into a set of workable techniques, and then used to develop analysis and design representations and models using UML.

During my long professional career, I have met with hundreds of companies and thousands of software professionals worldwide. Each company and every person has a unique opinion on what makes software so challenging, and there are as many opinions

on how to meet those challenges in a particular situation. But few software professions would disagree with a very simple statement: "If you don't know where you're going, it is very, very difficult get there."

Effective methods for analysis and design provide software professionals with a destination and a road map for getting there. In *Object-Oriented Analysis and Design for Information Systems*, Raul Wazlawick provides the tools software professionals need to craft the road map, and as a consequence, this book helps every reader get where he needs to go.

Preface

This book presents, in a deep and didactic way, elements of object-oriented analysis and design for developing information systems.

The area of software development for information systems has evolved during the last two decades around the Unified Modeling Language (UML) and the Unified Process (UP), adopted as international standards by the Object Management Group (OMG). Even agile developing methods nowadays use object-oriented development as the leading paradigm.

In contrast to some other books in this area, which are organized around the presentation of UML diagrams and their many possible uses, this book concentrates on the daily analyst's and designer's activities, and it explains how the different diagrams may help them in developing better systems and developing systems better.

Although the object-oriented culture has more than 40 years of existence, there are still companies that find it hard to produce good quality software, especially due to the lack of maintainability and flexibility on the systems they produce. This book presents information and techniques to minimize that deficiency.

The techniques described here were applied with success by the author in many software development companies, as well as in research projects at the university level.

This book is intended for computing professionals (analysts, designers, and programmers), and computer science students, both in undergraduate or graduate studies with at least a basic knowledge of object-oriented development or modeling. Beginners and experienced people will find the guidance presented in this book useful in improving the quality of their projects. This book was written to help them achieve the best design possible.

In contrast to what occurs with the waterfall model ([Royce, 1970](#)), where analysis and design activities happen sequentially, iterative models like UP propose that those activities (also called *disciplines*) may happen with higher or lower emphasis at any phase.

This book gives more emphasis to the activities that are typically performed by analysts and designers, including business modeling, requirements engineering, analysis, and design. Other disciplines such as project management, implementation, and testing are also referenced. For more information on other disciplines, books on software engineering such as [Pressman \(2010\)](#) may be consulted, or books more specific to the Unified Process such as [Kruchten \(2003\)](#).

The goal of the UP *Inception* phase is to build a general view of the system and its context, and plan further development. The two main modeling tools for this phase are *conceptual modeling* ([Chapters 6 and 7](#)) and *high-level use cases* ([Chapter 3](#)). *Business*

modeling (Chapter 2) may be necessary with more or less emphasis, depending on the project that is going to be developed.

At the end of the Inception phase, high-level use cases may be used as a basis for *planning* the rest of the project (Chapter 4). Usually, that planning involves rating use cases as small, medium, or large (the t-shirt scale) with regard to their complexity, and deciding on their priority. When the complexity of each use case is known, it is possible to estimate the average time necessary to develop them (Karner, 1993), and thus, the iterations may be planned as a set of activities that can be realistically performed with the time available.

The Elaboration phase of UP is performed as a sequence of iterations. Each iteration has a set of use cases assigned for development.¹ Developing an individual use case usually starts with its expansion into flows (Chapter 5). Optionally, the use case flows may be represented as *system sequence diagrams*, in order to help discover system queries and commands that have to be implemented.

During the Elaboration phase, when the architecture is evolving, the *conceptual model* (Chapters 6 and 7) is refined and completed.

System queries and commands may be implemented with a programming language. But the designer may choose rather to specify *contracts* for system queries and commands (Chapter 8), which besides being a higher-level definition for the expected behavior of each procedure, may be very useful in helping to define *test cases* (Chapter 11). These contracts correspond to the functional model of the system, because they define the functionality of each query and command in terms of the input they receive and the results they produce.

At this point, again, the system may be implemented, or its code may be automatically generated if tools are available. However, if elegant code is desired, it may be helpful to produce a new set of diagrams (sequence or communication diagrams) for each of the system operations (Chapter 9). Those diagrams indicate how the different objects will exchange messages in order to distribute responsibilities and attain the desired postconditions specified by the contracts. With this procedure, design patterns are easier to apply to the code. These diagrams also help to specify which methods (besides getters and setters) must be implemented in each class, and how they must be implemented.

During Elaboration, the design of the interface can be completed (Chapter 12). This book presents the IFML notation² as an option for modeling that aspect of a system. IFML is a brand-new standard modeling language for information flow that is being adopted as one of the OMG standards from March 2013.

Although it is more emphatic during the construction phase, the elaboration phase may also require the generation of a database structure (Chapter 13) and running code (Chapter 10). Data persistence may be generated automatically if the appropriate tools are available. However, it is worth at least knowing how the persistence mechanism works, even if it is automated. Code generation is also presented as a set of rules that can be automated.

In order to save space for a deep presentation of the topics, most of the exercises can be accessed on the Internet.

¹An iteration may also have a *mitigation plan* assigned for a given risk or a *change request* originated by the client, user, or even the development team.

²<http://www.ifml.org/>.

CHAPTER 1

Introduction

The main goal of this book is to present a set of good practices that allows software developers to comprehend and use object-oriented principles in an effective and efficient way. This chapter presents the motivation for this book and a quick introduction to UML (Unified Modeling Language) and UP (Unified Process). The goal of this book is not to be another reference manual to such technologies, but to help people who are working with them to accomplish their jobs better.

Keywords

UML; Unified Process; object-oriented analysis; object-oriented design; object-oriented development; design patterns

Key Topics in this Chapter

- Object-Oriented Systems Development
- Unified Modeling Language (UML)
- Unified Process (UP)

1.1 This book

There is a vast literature that aims at syntactically presenting UML diagrams, such as for instance, [Miles and Hamilton \(2006\)](#). There are also a significant number of books about development processes and management activities, such as [Satzinger, Jackson and Burck \(2011\)](#). However, there are relatively few books that go really deep into the presentation of best practices that allow the effective application of object-oriented techniques in software development in the real world. This means that despite the vast literature in this area, many questions faced by developers are still left unanswered.

This book presents more details on many object-oriented concepts and activities, especially those presented by [Larman \(2004\)](#), which is based on the Unified Process, commonly known as UP ([Jacobson, Booch, & Rumbaugh, 1999](#); [Scott, 2001](#)).

The reason this book is based on Larman's concepts is because he presents a concise and efficient approach to systems analysis and design using an object-oriented paradigm. In his approach, each artifact (document or diagram) has a very clear and practical reason to exist and the connections between artifacts are very precise.

In our approach, agile principles adopted by *Extreme Programming* (XP) ([Beck & Andres, 2004](#)) are largely accepted. However, we propose that programming must be undertaken first using a modeling language before directly programming code. The use of UML diagrams as high-level programming tools instead of just documentation tools is

recommended. With that approach, the artifacts used should contribute directly to the understanding of the problem (analysis) or to the design of a solution. In order to improve the process of designing a solution, we consider model transformation and automatic code generation fundamental.

Furthermore, in contrast to Larman's work, this book presents some original concepts and other details on topics as follows:

- Objective criteria to identify use cases and decide when to subdivide them or not.
- A technique to expand use cases that reduces the disparity among descriptions created by different analysts (the concept of mandatory and complementary steps allows one to decide exactly how many steps a use case must have).
- System sequence diagrams built with actor, interface, and control (instead of actor and system only) to help realize a clear difference between system events and system operations.
- An original approach for writing system commands and queries contracts with the use of the Object Constraint Language, OCL ([Object Management Group, 2010](#)) that allows for automatic generation of running code, and not only postcondition checking, as current tools do.
- An adaptation of [Fowler's \(2003\)](#) analysis patterns to UML, and restructuring some of those patterns to simplify their identification and application in practice.
- A systematic technique to generate dynamic object models from OCL contracts, which follows good design patterns, and reduces significantly the amount of code necessary for running an application, avoiding, for example, redundant verifications.
- An interface tier design presented with the use of IFML (Interaction Flow Modeling Language), a brand new OMG standard for interface modeling.

The aforementioned features are useful for producing high-quality software that is well organized, based on a multitiered architecture, and able to change or accommodate new requirements.

1.2 Object-oriented systems development

What is object-oriented systems development? By observing the way object-oriented analysis and design is taught and practiced in some places, it can be concluded that many professionals simply adopt an object-oriented programming language, or use fragments of an object-oriented based development process. However, very often, they are not as effective as one would expect.

It is not sufficient to organize the system architecture in tiers and modules if the code implemented inside of it is disorganized. Some programmers organize the system adequately in classes and packages, but they still write *spaghetti code* inside the methods of these classes and packages. In addition, other developers still use top-down functional decomposition inside methods, which is not appropriate when using object-oriented programming (top-down decomposition is adequate if structured programming is used instead).

In order to build code that is really object-oriented, developers should learn the

techniques of *delegation* and *responsibility assignment*, which can lead to reusable code and low coupling. Those techniques are explained in this book.

It is useless to invest heavily in object-oriented CASE (computer-aided software engineering) tools without learning the way to *think* in terms of object-oriented programming. The use of diagrams will not necessarily improve the quality of the software, although it may help.

1.3 Unified Modeling Language (UML)

Some developers believe that UML is a methodology, maybe because of the “M” in the acronym. However, that is not true: UML means *Unified Modeling Language* and it is therefore a *language* that can be used to describe things.

Knowing a language does not necessarily imply the ability to produce useful artifacts. For example, English is a language, but someone who knows how to speak English does not necessarily know how to write good poetry or how to make good speeches. Besides the language syntax, there are knowledge and techniques of best practices that greatly help poets and speakers to place the elements of the language in an order and structure that is adequate to produce the expected results.

The UML language has been under development since James Rumbaugh and Grace Booch joined forces at Rational Software and started to unify their already well-known diagrammatic notations and processes. Later, Ivar Jacobson joined the group and added his use cases and other notations to the unified language that was under development.

UML is constantly being revised and currently has the following three families of diagrams:

- *Structure diagrams*: Includes package, class, objects, composite structure, component, profile, and deployment diagrams. They are used to define what must be implemented in the system in terms of components. They are useful to specify the part of the system architecture that is time independent.
- *Behavior diagrams*: Includes use case, activity, and state machine diagrams. They emphasize what must happen in the system or business process. They are used to describe the functionality of the system.
- *Interaction diagrams*: Includes communication, sequence, timing, and interaction overview diagrams. These are a subset of behavior diagrams and describe the control flow between different components of the system.

Not every diagram must be used during the development of a system. Only those that represent useful information for the project are recommended. This book emphasizes the use of the activity, machine state, use case, sequence, communication, and class diagrams for modeling information systems. However, other diagrams can be useful depending on the features of the system being modeled. For more information on the different UML diagrams, the book by [Miles and Hamilton \(2006\)](#) may be consulted.

1.4 Unified Process (UP)

The techniques presented in this book are compatible with the Unified Process, which is heavily based (although not necessarily) on UML.

UP was also proposed by the *three amigos*, Grady Booch, James Rumbaugh, and Iva Jacobson (Jacobson, Booch, and Rumbaugh, 1999), as the result of their extensive experience.

This process is based in the following principles:

- *Use case driven*: The development is planned and organized over a list of use cases.
- *Architecture centered*: The development process leads to the construction of a system architecture that allows the implementation of the requirements. That architecture is based on the identification of a structure that is iteratively built from a conceptual model.
- *Iterative and incremental*: Development is divided into iterations or development cycles. At each iteration, new features are added to the system architecture, or corrected/refined, leaving it more complete and closer to the final desired system.
- *Risk oriented*: The elements of greater risk for a project are addressed early. For instance, critical use cases are identified, detailed, and implemented before the others.

UP includes in its disciplines the main activities related to software development. Those activities have different levels or emphasis during the four major UP phases Inception, Elaboration, Construction, and Transition (Figure 1.1). Although sequential in time, those phases must not be confused with the waterfall phases (Royce, 1970). In the waterfall model, requirements specification must be completed before design, design must be completed before construction, and so on. In UP, requirements specification, design, construction, and other activities are performed in all phases with different emphasis.

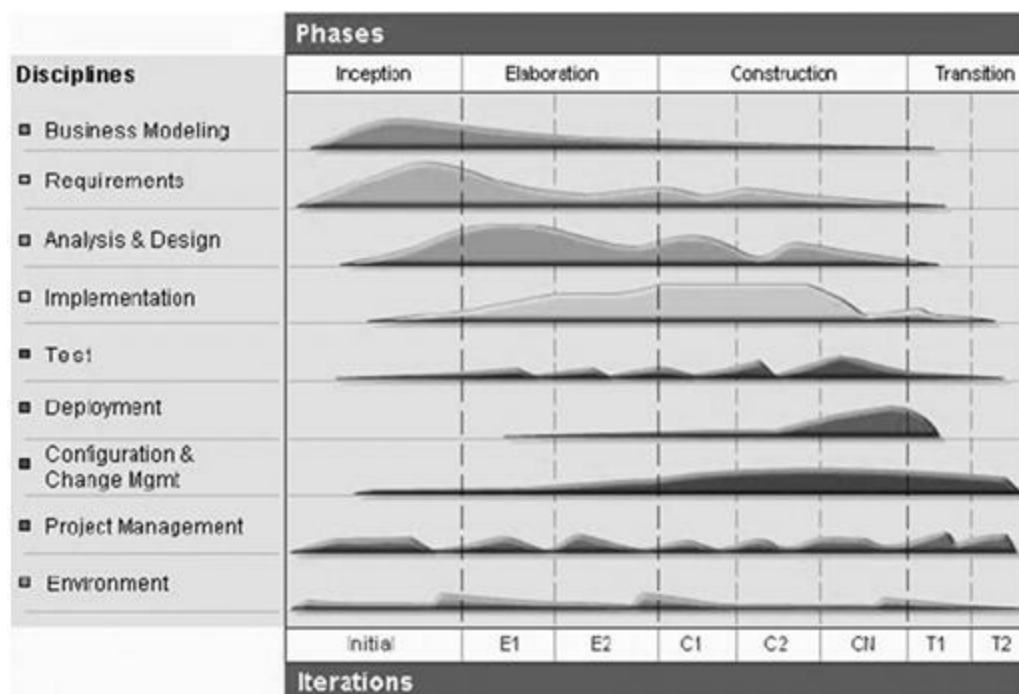


FIGURE 1.1 The emphasis of the different disciplines during the four phases of the Rational Unified Process

Inception is the first phase of UP, in which the main requirements are discovered and the extension of the system is understood. The output of this phase usually consists of a preliminary conceptual model (Section 3.6); a requirements document, usually in the form of a list of high-level use cases (Section 3.3) and supplementary specifications (Section 3.5.8); and a development schedule based on the use case list (Section 4.3). Additionally, a list of high-importance risks and their mitigation plans may be created, as well as other plans attending the special needs of the project. But those aspects are out of the scope of this book, which will concentrate on modeling techniques, not management. Only iteration planning based on high-level use cases is presented in detail.

The *Elaboration* phase includes a more detailed requirements analysis, which is performed by expanding the use cases, that is, writing the sequence of steps that characterizes each of their possible flows. The conceptual model is refined after each use case is expanded. Depending on the priority of the use cases, it is expected that the number of changes applied to the software architecture decreases as the project proceeds during Elaboration.

During the *Construction* phase most of the code production and test activities are performed. It is expected that the Elaboration phase produces an architecture sufficiently stable so that its refactoring will be minimized during this phase.

The *Transition* phase consists typically of the final tests and the delivery of the system to its users including possibly its installation and data migration. During this phase, the system will be deployed, possibly replacing an existing system (manual or automatic).

The Elaboration and Construction phases are performed in iterations. An iteration may have as an objective developing one or more use cases, implementing change requests, or mitigating selected risks. During an iteration, use cases are expanded and the information learned from them is incrementally incorporated in the product. It is expected that the Elaboration phase deals with the major risks of the system, as well as with the more complex or risky use cases that affect the system architecture significantly. On the other hand, the Construction phase concentrates on producing code for the whole application and implementing change requests.

UP is usually understood as a prescriptive process. But it may also be performed as an agile method, with few artifacts. Two popular agile implementations of UP are AUP⁴ (Ambler & Jeffries, 2002) and OpenUP⁵ (Kroll & MacIsaac, 2006). An *agile process* is one that prioritizes:⁶

- People and iterations over tools and processes.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

In order to obtain such agility, all documentation must be directed to software production. Each activity performed by the developer must have a very clear goal and a precise use, aiming always toward the production of code that meets the requirements in the best way possible and in the shortest reasonable time. Software is designed with two

goals in mind: understanding client needs, and producing a viable solution to those needs. In order to help people adequately communicate their needs and solutions, different artifacts, such as diagrams, may be created; diagrams are more useful when they allow code to be automatically generated from them.

1.5 The process so far

At the end of each chapter, a table shows which analysis and design activities have been covered by the book, highlighting those that were covered by the current chapter. This initial chapter is a general view of the Unified Process. Only the Inception and Elaboration phases are addressed in this book, and only the disciplines of business modeling, requirements, analysis and design, implementation, test, and some aspects of project management are addressed.

	Inception	Elaboration	Construction	Transition
Business Modeling	Areas covered by this book			
Requirements				
Analysis and Design				
Implementation				
Test				
Project Management				
Deployment				
Configuration and Change Management				
Environment				

1.6 Questions

1. What are the four fundamental principles of the Unified Process?
2. What is the goal of each of the four phases of the Unified Process?
3. How do the phases of UP differ from waterfall's phases?

¹<http://www.uml.org/>.

²RUP (Rational Unified Process) is the oldest and most known implementation of the Unified Process.

³Source: <http://www.ibm.com/>

⁴<http://www.ambysoft.com/unifiedprocess/agileUP.html>

⁵<http://epf.eclipse.org/wikis/openup/index.htm>

⁶<http://agilemanifesto.org/>

CHAPTER 2

Business Modeling

Business modeling is the first time-consuming activity that is performed during the development of most projects in the software industry. Although business use case diagrams and other UML diagrams are currently used to help teams understand business processes, sometimes the connections between those elements are not clear. This chapter shows how business use cases may be used to produce a high-level view of a business, and how they can be detailed by using business activity diagrams and state machine diagrams that describe key business processes and key business objects. It also shows how business use case diagrams may be used to define the scope of a system.

Keywords

Business modeling; business use case diagram; activity diagram; state machine diagram; project scope

Key Topics in this Chapter

- General View of the System
- Business Use Case Diagram
- Activity Diagram
- State Machine Diagram

2.1 Introduction to business modeling

The Inception phase of the Unified Process (UP) consists of a period of time when the analysts are looking to gather information about the business to be automated or restructured. It is assumed that the knowledge that the analysts have about the business is minimal, and that the interaction with the stakeholders will be intense. The goal in this phase is to discover if it is worth it to do the analysis without developing too deep into the project.

Usually, before starting to discover and analyze system requirements, it is necessary to understand the goals of the target organization. This view can be obtained by the UP discipline called *business modeling*.

Business modeling is, therefore, an activity that supports the discovery of system requirements by helping the team perceive the wider business context where the future system will operate. Different text or diagram artifacts can support it, as this chapter presents.

The artifacts in this phase usually do not need to be too detailed. Meetings with stakeholders could produce a record (minutes) that probably can register several ideas about the goals of the company and the automation opportunities. By using this information, the team may build a general view of the business.

Starting with the records obtained, the team can organize an initial document for the project, which can be called *general view of the system* or *executive summary*. That document usually includes the *scope declaration*.

Other relevant information is usually included in the general view as well, such as *stakeholders*, *power assignment*, *risk analysis*, *preliminary budget and schedule*, and the *project premises*.

The scope declaration of the project may be related to the business analysis activities. After discovering useful information about the business, the team is better able to determine what must and what must not be included in the project. The general view of the system may be elaborated with the help of some UML diagrams. Business use case diagrams (Section 2.3), and business activity diagrams (Section 2.4) may be used, and occasionally state machine diagrams (Section 2.5) can be used as well. Alternatively, the *Business Process Model and Notation*¹ BPMN², may be used instead of the UML activity diagrams. These diagrams help in determining if the scope of the project is complete and consistent with the goals of the organization.

Business modeling consists of studying and understanding the organization and its processes, because usually the system to be developed will not be an isolated product, but an organic part of the organization context. The goals of this discipline are to:

- Understand the *structure and dynamics* of the target organization in which the software will be used.
- Understand the *current problems* of the target organization and identify *potential improvements* that can be obtained with the software.
- Assure that clients, users, and the development team share a *consistent understanding* of the target organization.
- Derive the *requirements* that will lead to the desired improvements.

One of the critical points for the success of a software development project is its funding. In order to keep interest in the development of a software product a client must always stay convinced that its investment represents a gain, and not a waste of time and money. Comprehension of the business model is fundamental so that the development team can keep the client interested in the project.

Another important aspect of business modeling is to bring the business team and the software engineering team closer, so that the actual problems of the organization and its needs are understood, analyzed, and solved using information technology.

Business modeling is very important when significant behavioral changes are introduced to a group of people. In this case, business context and the consequences of installing a new system must be acknowledged and studied. It may be less important when such behavioral changes are not expected.

Thus, business modeling may have different levels of importance depending on the project needs. Kruchten (2003) identifies six scenarios of growing complexity in terms of the need for business modeling:

- *Organizational chart*: It may be necessary only to build an organizational chart in order to know the sectors and responsibilities related to the area in which the project is inserted. In this case, business modeling, in general, is very simple, and happens only

during Inception. The organizational chart will help analysts to locate the people that are responsible for the different aspects related to the future system, and what degree of authority is assigned to each one when deciding the requirements.

- *Domain modeling*: If the goal of the project is to build applications to manage and present information, then business modeling may be performed during domain analysis, that is, when a static model of the information (conceptual model) is produced. In this case, business modeling is performed during Inception and Elaboration.
- *One company, many systems*: It may be the case that the team is developing a whole family of systems for a company. In this case, business modeling is not only about one system, but spreads along many projects, and could even be a separated project of its own. It will help to discover individual system's requirements as well as to determine a common architecture for the whole family.
- *Generic business model*: If the goal is to build one or more systems that will serve a pool of organizations, then business modeling is useful in order to align the different organizations to a common business view, or, if that is not possible, to obtain a business view in which the specifics of the organizations are visible.
- *New business*: If an organization decides to start a new business, and a whole set of systems must be developed to give support to it, then a significant business modeling effort must be undertaken. In this case, the goal of business modeling is not only to find requirements, but to verify the effective viability of the new business. Thus, business modeling, in that situation, could be an independent project.
- *Renewal*: If an organization decides to completely renew its way of business, then business modeling will be a separate project, performed in several steps: vision of the new business, reverse engineering of the existing business, direct engineering of the new business, and installation of the new business.

Other factors that increase or decrease the need for business modeling in a project are:

- Well-defined and stable requirements at the beginning of the project (for example, creating a substitute for an existing system) reduce the need to perform intense business modeling.
- If the new system is going to change the way people do their work, then special attention must be paid to business modeling, especially when the business model is going to be changed or rebuilt after the automation of some business use cases.
- If the project has important risks, especially business risks, then business modeling must be more detailed and intense.
- If the user community is large and heterogeneous, then business modeling probably will be more difficult.
- If it is necessary to integrate the new system with hardware and legacy systems, then the comprehension of those elements must be included in business modeling, generating more work.
- More uncertainty about requirements means more attention to business modeling.

No matter the level of detail and effort to be put into business modeling, every project must begin with preliminary documentation that allows for an understanding of its

scope. A project with poorly defined scope will certainly give rise to discomfort or even fights between the client and development team, regarding the inclusion (or lack of inclusion) of certain features.

2.2 General view of the system

The activities related to the definition of the scope of a project should usually take a relatively small fraction of the time of the whole project, although variations may exist depending on the kind of the project: projects with simple and well-defined requirements may demand no more than a couple of days of business modeling while complex and large projects may demand weeks or months. At the time of scope definition, all information about the company business must be explored by interviewing users, clients, and domain specialists, as well as by the examination of documents, reports, existing systems, and related bibliography.

For most projects, the first question the analyst should answer is the following: What is the vision of the company for the project? In other words, what does the company want with the project? Why is it being proposed, and why is the company going to spend money with it?

At that moment, another question, often forgotten, may be raised: Buying or developing? Sometimes, the product the client wants is available for purchase.

These questions must be answered in a relatively short time. That is why it is suggested that the Inception phase proceed relatively quickly. Why? Because, at this time, the client and the development team usually do not have an idea of the real extension of the project and it is, from the point of view of both, an investment on the future, and, therefore, a risk.

The general view of the system or executive summary is a free format document, where the analyst may report the relevant items she discovered about the system after the initial interviews with the stakeholders. The document usually includes the scope declaration for the project.

We do not intend to propose rules for writing that document. But it is suggested that it should not be too long. A few pages of text and some diagrams seem to be sufficient to describe in a summarized way the scope of most systems. More than that could mean that too much detail was included in the summary. These details could be alternatively addressed in other documents such as the list of risks, requirements, or use cases.

The scope declaration in the general view must present, in general lines, the product that must be developed: what must be included, and what *could* be included but should not. This information may be obtained initially by interviewing the stakeholders, and may be refined later with the use of the diagrams presented in this chapter.

If possible, the main deliverables of the project should be also defined in the general view as well as the time frame when the client is going to receive some kind of delivery from the development team. Normally, this list of deliverables consists of implemented versions of the software, but the list can also include other items, such as design, manuals, installation media, training, etc. It can be very difficult to establish deadlines

for deliveries before doing the requirements analysis of the system, because estimation effort techniques such as *function points* (Albrecht and Gaffney Jr., 1983), *use case points* (Karner, 1993), or *COCOMO II* (Boehm, 2000) can only be applied after a good deal of the requirements is known. Thus, the information about deadlines at this early stage is more a guess than a formal strong commitment for development.

The general view document may also contain some acceptance criteria for the product, that is, quantifiable items that will be used to decide whether the project was a success or not. The product acceptance criteria must at least include metrics for deadlines, budget, and quality. If subjective criteria such as “customer satisfied,” “system easy to use,” or “state-of-the-art technology” are used, they must be quantified, that is, it must be defined how to measure “customer satisfaction,” “ease of use,” “state of the art,” and so on. Examples of quantifiable acceptance criteria are “the system must support up to 50,000 simultaneous accesses without degrading performance,” “the system must eliminate the need to use paper for performing processes x and y ,” and “the system must allow selling up to 100 books a minute on the Internet.”

More information may be added to the general view document if needed (for example, main risks, technologies to be used, etc.). The general view of the system is the base agreement document for the client and the developer. It will be taken as a basis for planning the rest of the project.

It is important to mention that when the general view is being built, the requirements analysis has not yet been completed, and therefore the information mentioned in the general view consists of early commitments. It is expected that the requirements analysis will detail the scope in depth, but not augment its extension. For example, in requirements analysis the process of selling books may be detailed with discounts, sales, delivery, and so on (considering that “selling books” is mentioned in the scope declaration), but if the payment roll of the company was not mentioned in the scope declaration, then the inclusion of that item in the project would require a renegotiation of the scope declaration.

Pressman (2008) proposes a number of techniques to communicate with stakeholders in order to discover business goals, for example:

- *Traditional focus groups*, where a trained analyst meets with a group of end-users or people representing their roles.
- *Electronic focus groups*, where the meetings are conducted by the use of electronic media.
- *Iterative surveys*, where a number of surveys are conducted among end-users, presenting them some questions and asking for clarification.
- *Exploratory surveys*, where an exploratory research is conducted among end-users that use a similar existing product.
- *Scenario building*, where a group of end users is led to produce a description of a number of scenarios for using the system.

The last option seems to be very popular among agile method practitioners, where scenarios are sometimes identified as *user stories*.

Figure 2.1 presents a general view with *very* preliminary scope declaration for a

fictional virtual bookstore Livir,³ which will be used to illustrate the modeling techniques within this book. This text would have been obtained after a quick preliminary meeting with the client. Using this as a basis, a deeper study may be conducted with business use cases and other diagrams, so that the information contained in it is refined, as shown below in this chapter. The document is divided into two parts:

- A simple vision statement of what the project will offer to customers and the business that's creating it.
- The constraints for version 1.0.

General View for the Livir Project

Version 1.0

Vision:

A new business will be initiated: a virtual bookstore. The system to support it must manage the acquisition and selling processes of the company. Access for customers and management people must be accomplished through a Web site.

When a book is ordered, it is delivered immediately if available in stock, or else, the book is ordered to a publisher, and a compatible deadline is informed to the customer. The system must calculate taxes and delivery fee as well as applying discounts to the sale when applicable.

The system must allow a manager to generate reports on bestselling books, and on most profitable customers, as well as suggest books for buying based on past customer's interests.

Constraints:

- a) Customers would only pay by credit card.
 - b) The bookstore will deal only with new books, not used ones.
 - c) Access to the system will be available through a web site only.
-

FIGURE 2.1 General view for the Livir project.

To allow for the presentation of this project in a book, a series of simplifications were considered. The description and modeling of a real project would be much more extensive. Furthermore, the models presented would sometimes only be partial at a given time, being refined or changed later.

It can be seen that the general view is a nonstructured view of the project. It includes business information for managers, and operational information for developers. Sometimes, even details about the technology to be used can be described if necessary. What the analyst must consider is that the general view describes the main concerns of

the clients and those concerns will be better structured later during analysis and design.

A good analyst must act like a psychoanalyst, that is, she must avoid speaking in the place of the client. The analyst must help the client to speak; when the client starts to report some of her needs, the analyst must hear, instead of presenting fast prebuilt (and often mistaken) solutions. The analyst must motivate the client to detail her requirements and must question those requirements so that the client can decide which are urgent and which are not.

2.3 Business use cases

Jacobson (1994) presents in detail techniques for business modeling with business use cases. In general, in business models, there are entities (people or companies) external to the target organization, called *business actors*, and internal entities, called *business workers*. Business use cases are the processes performed by those actors and workers that allow them to reach some of the company business goals.

The business use case model considers the entire company as a system, and the actors may be people, companies, or other organizations that create business or maintain relationships with the business (Kroll and Kruchten, 2003).

The first thing the team must think about when doing business modeling is that what is being modeled is *not* a software system. A *company* is being modeled. Business information is sought from higher-level management people and specialists, because they know and sometimes create the company goals.

The quantity of business use cases that will be identified may be relatively small. A business process is a long-range process that is performed by the company; think, for example, of selling products, conducting marketing, providing services, solving customer problems, etc. Each process of this type must contribute to a business goal. This is not the moment for detailing such processes. For example, selling products may include many subprocesses such as registering customers, offering products, sending products, applying discounts, etc. But those processes (that later possibly will be identified as system use cases if they are going to be automated) are involved by the highest-level process, which is *Sell books*.

Although UML still does not have a standard symbol to differentiate business use cases from system use cases (Chapter 3), it is usual to represent business use cases with the stereotype `<<business>>`, or by cutting the use case ellipse with an edge as shown in Figure 2.2.



FIGURE 2.2 Business use case.

In this example, *Sell books* is a business use case because it involves a relationship

between the company and an external entity (a customer), producing a perceptible and consistent result: one or more books sold. This use case possibly will be performed by the customer with the collaboration of some workers inside the company; it may or may not use software systems.

Use case names must be chosen carefully, because at this stage they will summarize critical information about the system, and wrong choices may prevent the team and stakeholders from sharing comprehension about the real intention of business actors and workers. Blain⁴ presents some best practices collected from many sources:

- *Good use case names reflect user goals.* Names such as *Access system* or *Open main window* must be avoided because they reflect technology and not a business goal. Also names without verbs such as *Invoice* must be avoided because it is not clear what is going to happen to the invoice. Better names might be *Generate invoice*, *Cancel invoice*, *Pay invoice*, etc.
- *Good use case names are as short as possible.* Although some people may propose that a use case name should not have more than two or three words (or any other number), it is unfeasible to define such a limit because there is a risk of losing clarity on the business goal if the name is too short. As Blain points out, which word could be removed from the name *Collect late payment* without obscuring its meaning? However, that name is better than *Collect late payment from customers that are past due*.
- *Good use case names use meaningful verbs.* Use case names not only should have *strong* verbs, but they must be meaningful. A meaningless verb is one that does not make clear what the use case accomplishes; for example what does a use case such as “*Process order*” accomplish? What result does it produce? A name such as “*Separate ordered items for dispatch*” would be more meaningful in that case.
- *Good use case names use active voice.* As in general writing, active voice is preferable to passive voice. *Pay for order* is better than *The order is paid*.
- *Good use case names use present tense.* The present tense indicates what the user is trying to do. Past and future tense may sound confusing and unnecessary.
- *Good use case names do not identify the actor.* As use cases are linked to actors, actors are not part of their names. Thus, a use case name must not identify the actor as in *Manager creates sales report*. Just name the use case *Create sales report* and link it to a manager or other actors if necessary.
- *Good use case names are consistent.* The same naming rules and naming conventions should be applied along a project or group of projects. Avoid, for example, naming one use case *Generate sales report* and another *Produce reservations report*. Chose one verb for that meaning and use it consistently.

Additionally, [Probasco \(2001\)](#) proposes that the list of use case names should resemble a *to do* list. Thus, instead of writing *Cash withdrawal*, *Funds transfer*, and *ATM service*, it would be better to write *Withdraw cash*, *Transfer funds*, and *Service the ATM*, respectively.

Probasco also insists that the meaning of a use case name must be precise. For example, is *Auction* a good use case name? Is it a verb or a noun? If you consider that it is a noun and that a verb must be added, is *Execute auction* a good name? The verb *Execute* would probably be a programmer’s choice, but it probably has no meaning for an

auctioneer (Is the *Auction* going to be shot to death?). In this case, an idea would be to talk to a domain specialist in order to obtain a better name. On the other hand, even if “auction” is indeed a verb it is not sufficiently clear. Is it the auction of a single item? Is it the auction of a complete set of items? Is it simply an auction participant placing a bid? In that case, names such as *Auction items*, *Auction single item*, or *Place bid* would be clearer.

2.3.1 Business actors and business workers

During business modeling there are two kinds of actors that must be addressed (Figure 2.3):

- *Business actors*: People, organizations, or even systems that perform some activities belonging to the process, but which are not part of the target company. That is, they are not under company control. Business actors could be, for instance, customers, publishers, external auditors, partner companies, or even other automatic systems that the target company could interact with.
- *Business workers*:⁵ People, organizations, or even systems that perform some activities belonging to the process and that are part of the target company. They could be the company’s employees, its departments, or even existing software systems belonging to the company. Graphically, they are distinguished from business actors by the use of the stereotype `<<worker>>`.

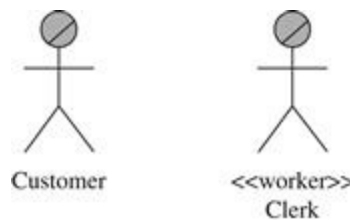


FIGURE 2.3 Business actor (left) and business worker (right).

This differentiation is important because business actors usually cannot be automated, that is, they will not be replaced by computational systems. However, business workers roles can possibly be replaced by automatic systems (English, 2007).

Business actors and business workers are linked to the business use cases they participate in by connecting lines without arrows,⁶ as shown in Figure 2.4.

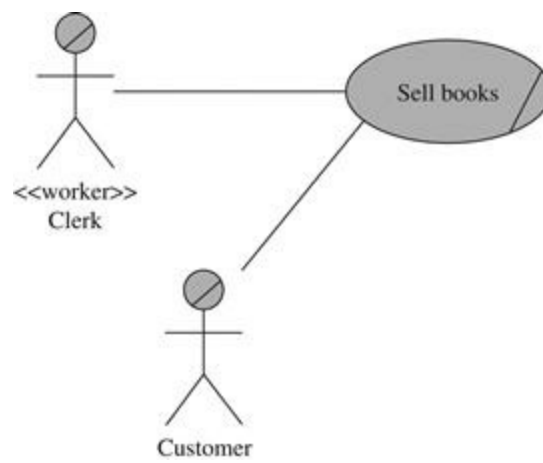


FIGURE 2.4 Links between business actors, business workers, and business use cases.

2.3.2 Automation opportunities

A bookstore that intends to sell through the Internet, like Livir, should produce a business model that identifies its main business use cases, external actors (customers, publishers, etc.), and business workers (managers, clerks, etc.). If we suppose that the bookstore company already exists and has no automation at all or very incipient automation, then the business workers will be, basically, real people. These workers may be candidates for automation if there is interest in increasing the level of automation of the company. For example, the role of the clerk that currently helps customers find books on the shelves may be replaced in the virtual bookstore by an automatic guidance system. However, the customer role cannot be replaced by an automatic system.

Every business worker role will usually not be replaced. It will depend on the scope of the project to be defined. An advantage in making the business use case diagram, then, is to help in visualizing and making decisions on automation. With the diagram, it is possible to visualize what must stay inside and outside the automation scope, depending on the project's goals.

Normally, during an automation process within a company, it is useful to replace at least the workers' roles that correspond to mere proxies, that is, the ones that simply perform actions in the place of an external actor. In the case of the bookstore, the sales clerk role is necessary only to help the customer at the physical store. But, when the sales are moved to virtual space, that role is not necessary anymore, because the customer may easily find her way on the website.

The association of the business use cases with the company's major business goals helps to identify the scope of the system, that is, the activities that will be effectively automated in the project that is being initiated. In the example of [Figure 2.5](#), all use cases are candidates for automation at differing levels of priority. However, the company could be interested only in implementing sales and acquisition at this point, leaving marketing and customer services left to a second project.

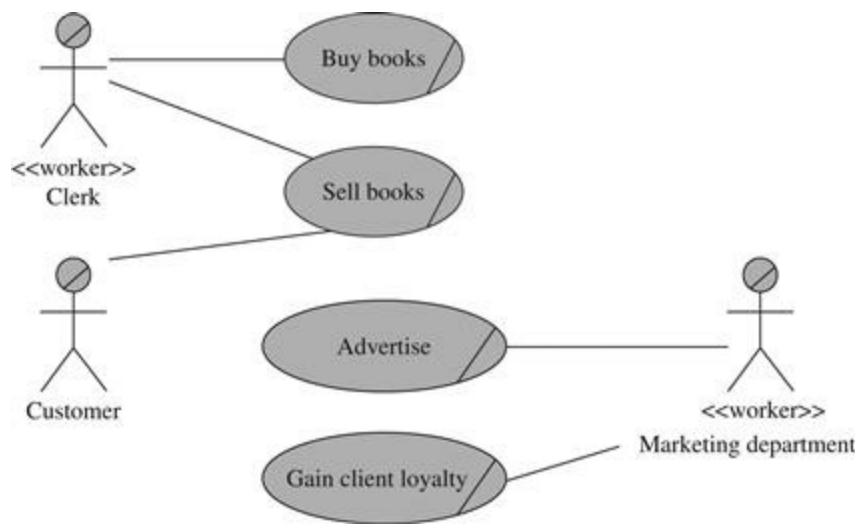


FIGURE 2.5 Business use cases: candidates for automation.

The system boundary can be used to identify the set of business use cases and business workers that will be automated, as shown in [Figure 2.6](#).

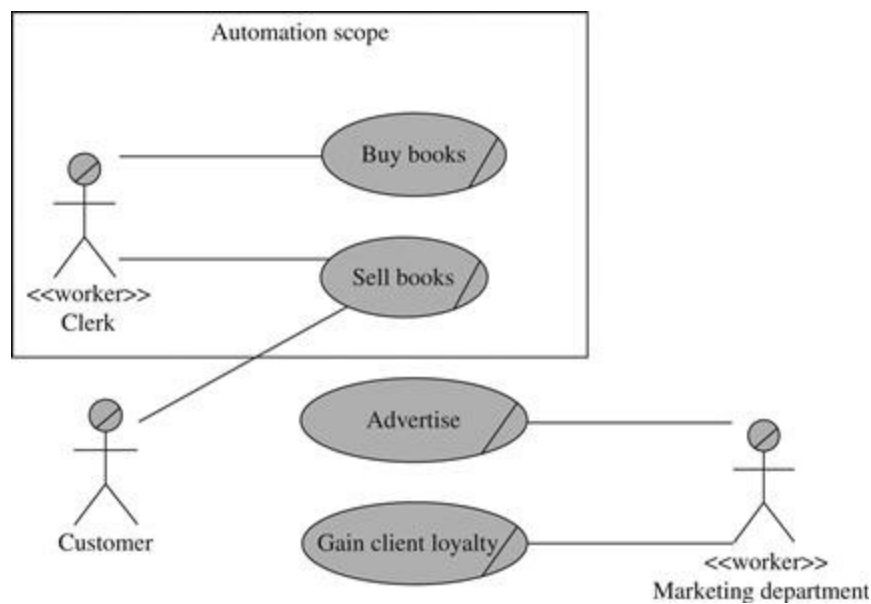


FIGURE 2.6 System boundary being used to indicate the scope of the project.

The business use case diagram shows that the automation project will include the processes of buying and selling books and the role of the clerk. It also shows that the processes of advertisement and gaining client loyalty are outside the scope of the project.

However, a deeper analysis may show that the worker that buys books for the bookstore cannot be easily automated. It might be the case that all books could be automatically bought by an automatic process that could analyze potential sales and stock, without the participation of any human being. But, if that is not the case, the best thing to do is to consider that the clerk really has two different roles: help the customer (that role will be automated) and buy books for the bookstore (that role will not be automated). [Figure 2.7](#) shows an evolution of the diagram where part of the responsibility of the clerk is reassigned to a worker named "Acquisition manager." There

is no one-to-one equivalence between a role and a person: a person may perform many roles and a role may be performed by many different people.

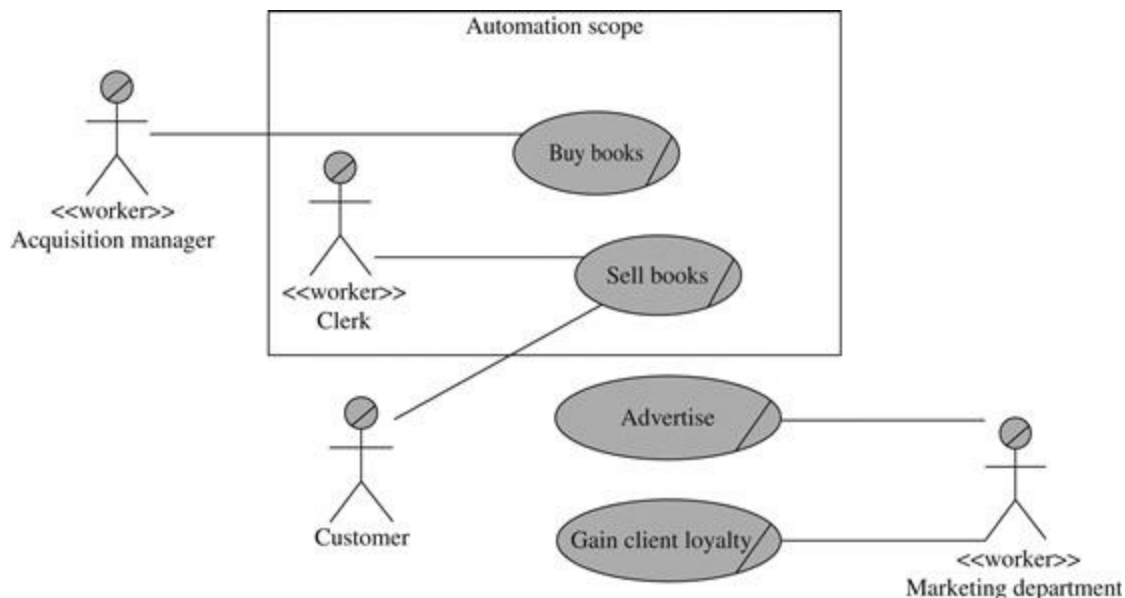


FIGURE 2.7 An actor is split because only part of its role will be automated.

If the project, on the other hand, consists of a number of subprojects or stages, in which different parts of the company would be automated, that could also be indicated by the use of different system boundaries in the diagram: one for each subproject or stage.

The next step that leads to the requirements analysis is the detailed study of the business use cases that will be automated. The level of precision and detail depends on the objectives of the project, as already discussed early in this chapter.

2.4 Business activity diagram

The UML *activity diagram* is a very popular option for studying the details of a business use case. With this diagram, the different activities performed by business actors and workers in order to reach the general goal of the use case can be specified.

For a better understanding of the way the company works through its use cases, it is possible to create an activity diagram for each business use case that is going to be automated. The activity diagrams can be used to represent processes at the organizational level.

2.4.1 Basic elements

Much of the UML activity diagram is similar to a flowchart. There are activities sequentially linked. Decision points and loops may be used and even parallelism may be expressed in such diagrams.

The activity diagram may be divided into *swim lanes*, each one representing an actor or

worker that participates at a set of *activities*. As in the use case diagram, an actor or worker may be a person, a department, a computational system, or even a complete organization.

The process represented in the diagram usually has an *initial node*, represented by a solid circle ●, and an *activity final node*, represented by a solid circle inscribed in a circumference ⊙. The initial node may not be considered mandatory in some cases when the beginning of the process can be inferred by observing an activity that does not have an antecessor. However, the use of that mark improves the readability of the diagram because it is easier to see where the process really begins.

The activities are represented by *action nodes* ◻. When an activity is represented inside a swim lane it means that the corresponding actor is the one responsible for the activity.

Flows or *dependencies* between activities are represented by arrows →. Flows usually link two activities, indicating precedence between them.

A *path* is a sequence of activities linked by flows: ◻ → ◻ → ◻ → ◻ → ◻.

In the Livir example, the activity diagram may be used as a way to visualize the details of business use cases such as *Sell books*. The model presented in [Figure 2.8](#) shows a first approach to the process.

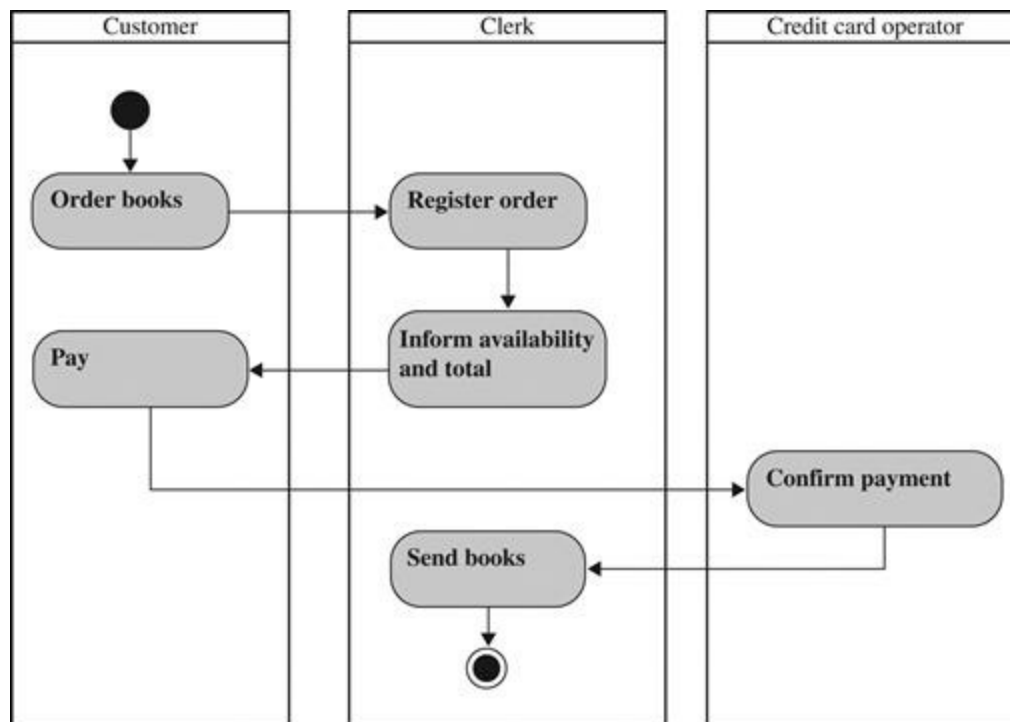


FIGURE 2.8 First approach to an activity diagram for modeling the book sales business process.

The diagram that is starting to be designed in [Figure 2.8](#) is not intended to be a model for the system to be built and should not be thought of like that. Its utility is to help the team understand which activities and actors are involved with the main business processes of the company, so that, using that information, they can perform a more effective requirements analysis later. The logical path of the diagram must be

understandable and validated by business people.

Later, the analyst will examine in detail each of the activities that must be performed. If a given activity belongs to the scope of the system that is going to be implemented, then it must be a target for a more detailed analysis.

2.4.2 Control flow nodes

Two control flow nodes are common in the activity diagram: decision nodes and parallelism nodes.

- The decision nodes (branch and merge nodes) are represented by diamonds \diamond . The flows coming out of the decision node must have guard conditions (a logic expression between brackets). Two or more flows may leave a decision node, but it is important that the guard conditions are mutually exclusive, that is, only one of them may be true at a time.
- The parallelism nodes (fork and join nodes) are represented by bars —|— . The flows coming out of a fork node are performed in parallel.

The diagram in [Figure 2.8](#) is still a very crude representation of the real process of selling books. Just to illustrate a possible evolution of that diagram, let's examine the situation when some of the ordered books are not available in stock. It would be necessary to order them from one of the publishers and add them to the customer order after arrival. [Figure 2.9](#) shows this situation by indicating that if some books are not available, then they have to be ordered from publishers, and the order is sent only after they arrive.

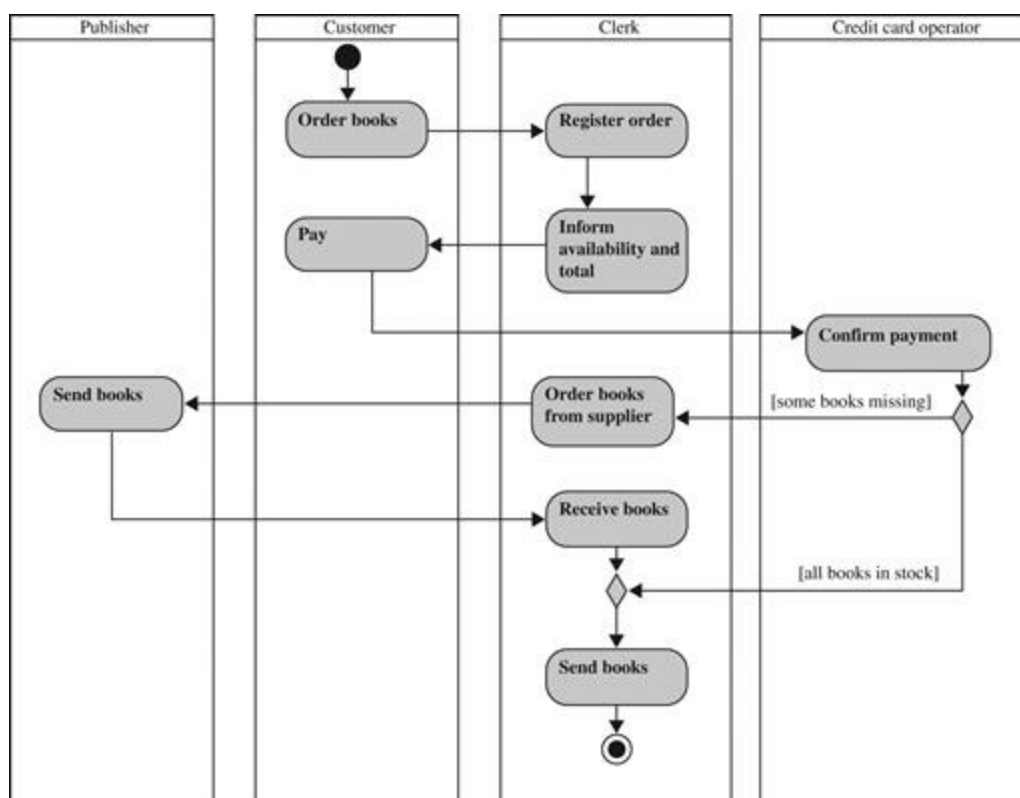


FIGURE 2.9 Example of decision control flow.

Just below the *Confirm payment* activity there is a branch node represented by the diamond. As noted before, the decision node allows only one exit flow to be followed. The guard conditions (*[some books missing]* and *[all books in stock]*) are mutually exclusive. From the branch node, the flow follows one of the alternate paths until reaching the merge node that is depicted just above the *Send books* activity.

Later, the analyst could discover that the model is still not satisfactory. For instance, even if some books are not in stock, the order could be sent in two or more deliveries. Thus, two paths could be performed in parallel: sending the books that are available in stock and, if necessary, ordering the other books, and sending them later, as shown in [Figure 2.10](#).

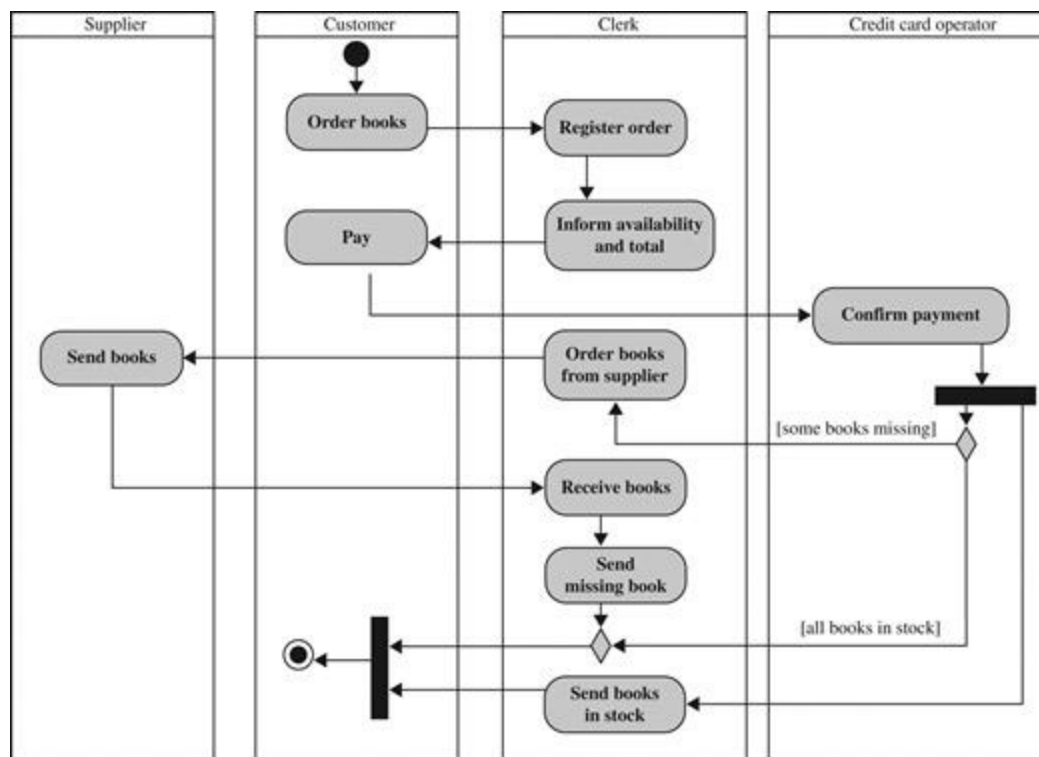


FIGURE 2.10 Example of parallel flows.

The bar below the *Confirm payment* activity in [Figure 2.10](#) represents a fork node because it starts two parallel paths, and the other bar represents a join node because it synchronizes the parallel paths into a single path.

The single path after a join node can only be followed if all paths that come to the join node have been followed. It can be seen that in that model, if all the books are in stock, only one of the parallel paths will have activities to be performed, because the other path immediately goes to the merge and join nodes.

The fork, join, decision, and merge nodes, as well as the initial and activity final nodes, may be placed inside swim lanes. However, this does not affect their semantics. Thus, the choice of a swim lane to place such nodes is only due to visual suitability.

Another node that can be useful sometimes is the *flow final* node, , which terminates a path (parallel or not) that is being performed. The difference between it and the activity

final node is that in the case of the activity final node, all flows of the activity diagram are terminated when a single flow reaches it, while in the case of the flow final node only one flow (among other parallel flows) is terminated, but the activity continues.

In the use case of [Figure 2.7](#), only one actor (customer) and one worker (clerk) participate in the use case *Sell books*. However, after detailing the activities related to that use case, as done in [Figure 2.10](#), it was discovered that two more actors are necessary: the *Publisher* and the *Credit card operator*. Additionally, the *Publisher* is an actor that has to be linked to the use case *Buy books*. Thus, depending on the level of detail desired at this point of the project, the business use case diagram may be updated to reflect those discoveries, as seen in [Figure 2.11](#).

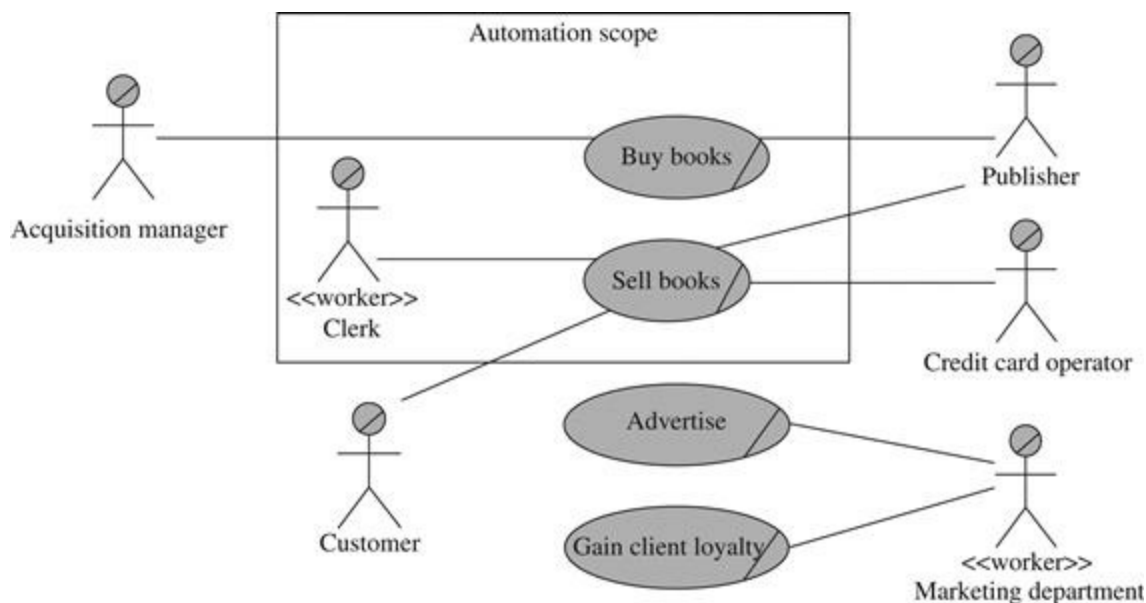


FIGURE 2.11 Business use case diagram updated with information discovered with the activity diagram.

Other options for detailing a business use case are the UML sequence diagram and communication diagram. However, those diagrams are used to represent messages being sent to elements. It is not always quite natural to find names to label those messages in the case of a business process, and meaningless names such as “*seek data*” and “*check result*” are chosen eventually. Thus, the activity diagram is a more natural choice to describe what happens in the real world, inside an organization of people. However, as will be seen in [Section 5.8](#), sequence diagrams are very useful to detail system use cases, because system use cases usually consist of a sequence of information flows being exchanged between actors and a system.

2.5 State-dependent aspects of a business

Besides business use cases, which may be detailed by activity diagrams, it may be important to understand other aspects of a business that do not always appear clearly in such diagrams, such as some key business objects that change state during their life cycle.

Thus, another UML diagram that can help the team to understand the organization's business is the *state machine diagram*. Like the activity diagram, this is a behavioral diagram, but, instead of modeling activities in processes, it represents a set of states in which a system, actor, or entity could be at a given instant.

The state machine diagram specifies *events* in its flows. An event triggers the flow that takes the entity from one state to another. That is, the flows are labeled with events that, if they occur, make the entity change from one state to another.

Those occurrences may be, however, restricted by *guard conditions*. For example, the *login* event may only send the system to a *logged* state if the password is correct.

Graphically, the events are represented in the flows by simple expressions, and guard conditions are represented between brackets, as in the case of activity diagrams.

The transitions in a state machine diagram can also include *effects* that are performed in response to a state transition. For example, the transition activated by the event *login* and guarded by the condition [*password correct*] can activate the action */grant access*, which is an effect of the transition (not its cause). Actions associated to transitions are represented by expressions initiated by a slash (/).

As an example on how that diagram can be useful during business analysis, we can imagine that a team is interested in understanding the life cycle of a book in the bookstore. Examining the different states of a key business object such as a book for this project may reduce the risk of underestimating the complexity of the system requirements.

Thus, the team may discover that the book is initially registered in the catalog of a publisher. The bookstore can then order a set of copies of the book. When the delivery arrives, the book is added to stock and made available for sale. Once sold, the book is removed from stock and taken to the shipment department. In some situations, books may return to the bookstore, for example, due to a wrong address, or for being damaged during shipment. In the first case, the bookstore must contact the customer and, depending on the contact, it can cancel the sale or resend the book. The book life cycle finishes only when the transport company confirms the delivery. All those changes of state (transitions) are represented in [Figure 2.12](#).

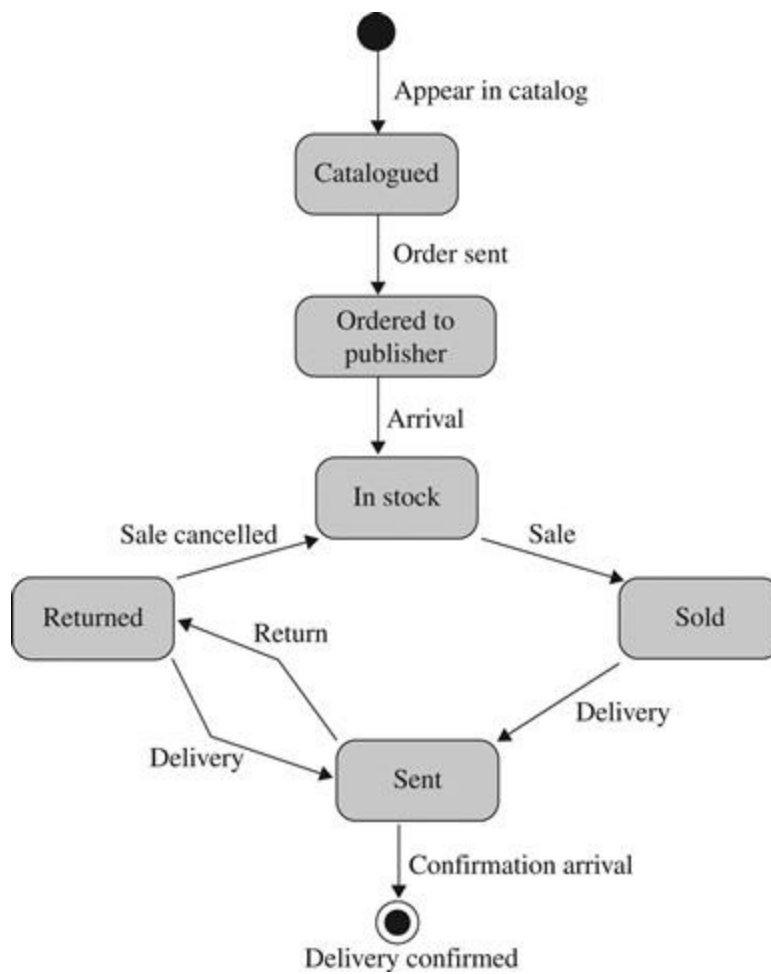


FIGURE 2.12 First state machine model for the life cycle of a book.

However, the model presented in [Figure 2.12](#) is still incomplete regarding the possible transitions of a book. For example, a delivered book can return damaged and, in that case, cannot be resent or sold again. This can be represented by adding a guard condition to the transition that goes from the *Returned* state to the *Sent* state, and adding a new final state in which the book is discarded, as shown in [Figure 2.13](#).

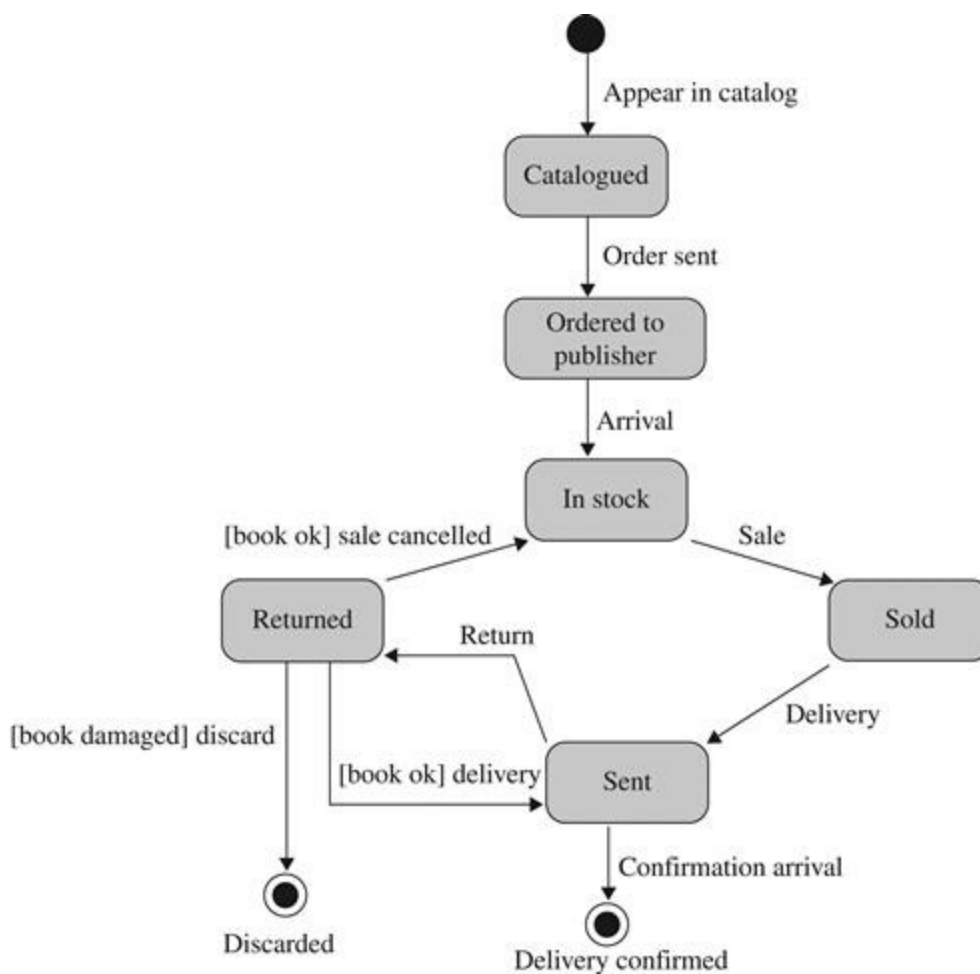


FIGURE 2.13 State machine diagram with guard conditions.

These guard conditions are still very informal. But this is acceptable during business modeling, because the goal is to understand the organization of the business, not specify a computational system. Nevertheless, precise guard conditions may be obtained if a formal language such as OCL (Object Constraint Language) is used. Later in this book expressions using OCL are introduced.

In some situations it is possible that an event causes a transition from more than one state. In the example of [Figure 2.13](#), it can be imagined that a book may not only be damaged when it is in state *Returned*, but also when it is in other states like *In store* and *Sold*. Then, a transition to *Discarded* should exist from those three states. However, to abbreviate this set of transitions a *superstate* may be used instead.

In [Figure 2.14](#), any transition from the superstate *In store* represents a set of transitions from each of its substates. In the case of a transition *to* the superstate, it is necessary to establish which of the substates would be the initial one. In order to do that it is necessary to include an initial state inside the superstate, or, instead, make the transition go straight to the substate.

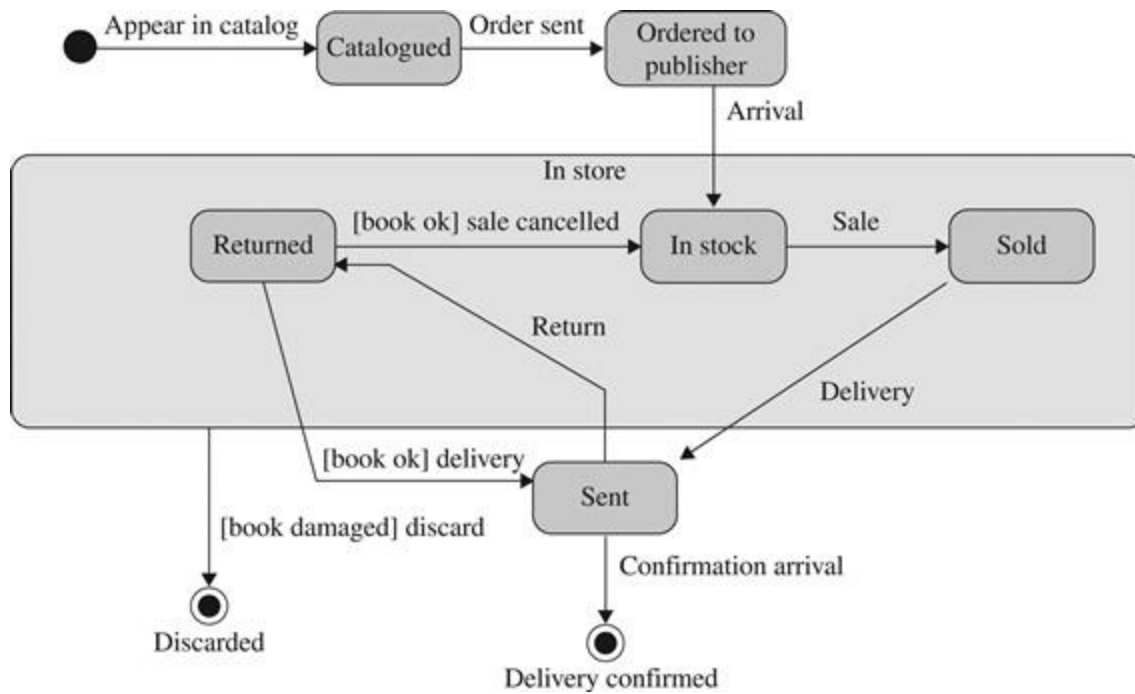


FIGURE 2.14 State machine diagram with a superstate.

An object may be simultaneously at more than one single state. For example, a book may be a special offer or not from the moment it is ordered from the publisher until the moment it reaches a final state as discharged or sold. Thus, a book may be modeled with two concurrent states: the first state determines if it is a special offer or not, and the second state determines its status regarding sales. In the diagram in [Figure 2.15](#) the concurrent states are represented inside *orthogonal regions* in a superstate.

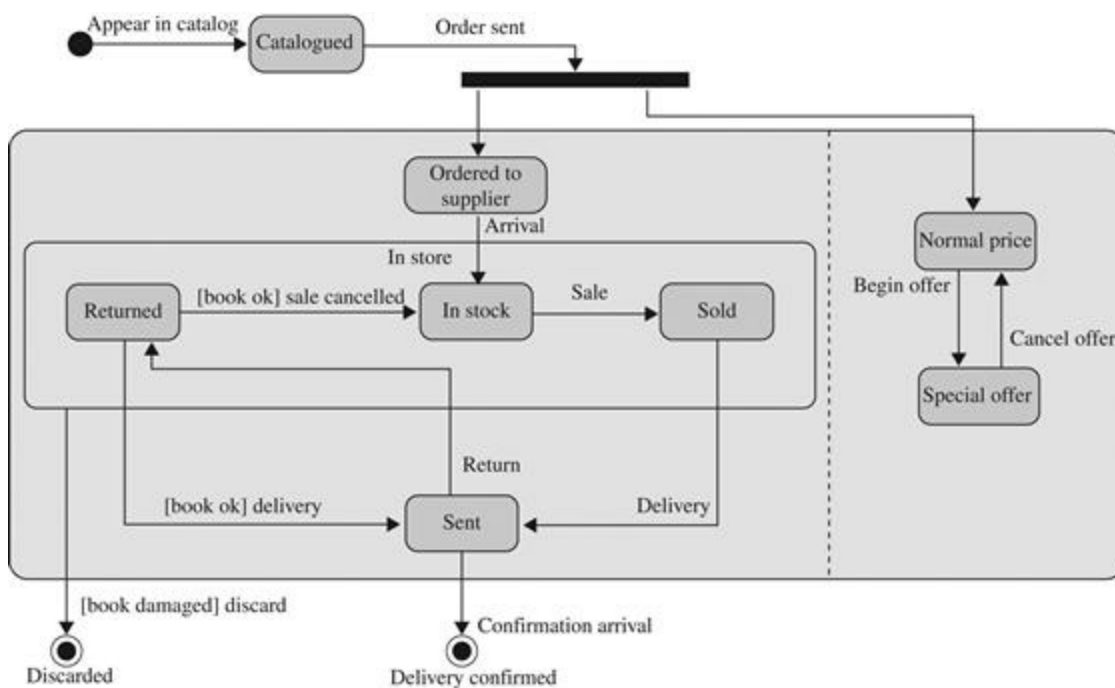


FIGURE 2.15 State machine diagram with regions.

When a transition leaves the superstate, then all concurrent substates are also left. For example, when an *arrival confirmation* or *discard* event happens in the diagram in [Figure](#)

2.15, both concurrent states cease to be active and the book is considered to be at a final state.

The analysis on the states of a book could continue. For example, it would be possible to determine that a book in state *Sold*, *Returned*, or *Sent* could not change price from *Normal price* to *Special offer* and vice versa. That could be modeled by adding a guard condition to the transitions from states *Normal price* and *Special offer*: [*not in state Sold, Sent, or Returned*].

Utilizing a state machine diagram to represent a business object is useful for improving the discovery of system requirements related to some key business objects, as shown in [Section 3.4](#).

2.6 Remarks

It is important to remember that the goal of business modeling is usually to find a general view of the business, and not a detailed specification of its processes. Thus, in most cases, the team's goal is to concentrate on discovering information about the business, and not on formally specifying how the business works as if it was a machine.

A question still may be asked: Which business elements deserve the production of a state machine or activity diagram? Unless stated to the contrary, it is not advisable to prepare a diagram for every business element, because, if this is done, Inception would take too long to complete, and its objectives would be hindered. What is usually needed at that point is a model for some key elements, so that their behavior can be better understood and their requirements derived later.

A clue to identify those key elements is to ask what the business objects are. In the case of a bookstore, they are books, in the case of a hotel, lodging, in the case of a court, criminal processes, and so on. Thus, those are the key elements that must be modeled in greater detail to help in understanding the business.

A second question could be the following: Which diagram should we use – activity or state machine? The answer depends on the nature of the element to be modeled. A state does not correspond necessarily to an activity. A TV, for instance, may be in state *off*, when it is doing nothing. An activity diagram is useful when modeling people, organizations, or systems doing things. On the other hand, the state machine diagram is useful when a single entity passes through different states in which it is not necessarily doing something. Furthermore, the activity diagram usually details a business use case (that is, a process like selling, buying, checking, etc.), while the state machine diagram is usually associated with a business object (like books, people, orders, etc.).

Business modeling is not just about building diagrams, however. The purpose of diagrams is to help understand the context and the initial general requirements of a project and a system. Business modeling is one of the key activities that help a team to identify and prepare for project risks. Other risk mitigation activities may also be performed during Inception, such as proof-of-concept, workshops, prototyping, early tests, etc. Any strategy that helps in understanding the big picture and identifying the major risks and complexities of a project are valid.

2.7 The process so far

	Inception	Elaboration
Business Modeling	Build a general view of the system: <ul style="list-style-type: none">• Build a business use case diagram and determine the automation scope for the project.• Build preliminary activity diagrams for business use cases.• Build preliminary state machine diagrams for key business objects.	
Requirements Analysis and Design		
Implementation		
Test		
Project Management		

2.8 Questions

1. What is the purpose of business modeling?
2. Business modeling can occur with different levels of intensity for different kinds of projects. Explain and give examples!
3. What is the difference between a business worker and a business actor?
4. Which aspects of a business may be detailed by an activity diagram? Which may be detailed by a state machine diagram?

¹Formerly known as Business Process Modeling Notation.

²<http://www.bpmn.org/>

³Why a virtual bookstore? Because it is a typical commercial information system, and because most readers should already have experience with that kind of system, and also because it is sufficiently simple, although rich, to illustrate the modeling techniques of this book. Such systems are already available for acquisition and customization, and it might not make sense to develop a new one from scratch, but the purpose here is just didactic.

⁴<http://tynerblain.com/blog/2007/01/22/how-to-write-good-use-case-names/>

⁵If we are modeling a company as a whole usually business workers do not appear, as they are internal to the company's system. But if connections between different subsystems of the company are shown, then business workers must appear.

⁶Some approaches suggest the use of arrows to indicate whether the actor only receives or sends information to the system. But that practice, besides not being supported by UML 2, brings practically no useful information at this point of the business analysis, because at the moment it is only important to know which use cases really exist, and which actors are involved with them. The details on the interaction will be given later with the activity diagrams that will describe how a business use case is performed.

CHAPTER 3

High-Level Requirements

This chapter presents system use case diagrams and supplementary specifications. It explains how the system use cases are connected to the business use cases, and which business actors become system actors. The chapter explains how high-level use cases represent the highest-level requirements of a system (functional and nonfunctional), and how the supplementary specifications may be obtained based on the quality model of the recently consolidated ISO 25010 *SQuaRE Model*. The chapter also explains how to choose the granularity of a use case in order to avoid diagrams so detailed that the number of elements hinders understanding, and how to avoid diagrams that miss important use cases so that the analysis could be deluded by the apparent simplicity of an actually more complex system. At the end of the chapter, we present the preliminary conceptual model, which is a class diagram used to represent the concepts that are relevant to the system. This model is detailed in [Chapters 6](#) and [7](#), but here it is briefly introduced just to stress its importance in order to find the correct and complete set of requirements that a system must implement.

Keywords

Use cases; actors; requirements; ISO 25010; SQuaRE; supplementary specification
preliminary conceptual model

Key Topics in this Chapter

- System Actors
- System Use Cases
- How to Find System Use Cases in the Business Model
- Requirements
- Preliminary Conceptual Model

3.1 Introduction to high-level requirements

Once the business has been reasonably understood and modeled, the requirements analysis may begin. Although there are several approaches to represent requirements, this book presents a technique based on use cases.

The *system use case* is an individual process that is identified from the business activities ([Section 2.4](#)). *High-level* (or *brief*) system use cases represent the more general *functional requirements* of a system. The *annotations* on these use cases represent the *nonfunctional requirements*, that is, the constraints and qualities related to those functions. The *supplementary requirements* are nonfunctional requirements that apply to the system as a whole – not only to individual functions. They may be represented in the *supplementary specification* document ([Section 3.5.8](#)).

System use cases are useful for many activities related to systems development, such as:

- *Definition and validation of the system architecture*: In general, classes, associations, and attributes that form part of the system architecture are obtained from the use case¹ texts (Section 3.6).
- *Creation of test cases*: Use cases can be seen as a basis for system and acceptance tests (Section 11.7), where the functionality is tested from the point of view of the user.
- *Planning iterations*: Each use case receives a priority (Section 4.3), and the effort to develop it is estimated (Section 4.2), so that the team can decide on which use cases to develop at each iteration.
- *Basis for user documentation*: Use cases are descriptions of the system's normal operation flows, as well as alternate flows that represent how to deal with eventual exceptional conditions (Chapter 5). These descriptions are an excellent basis for starting the user manual, because all possible functionalities are described there in a structured and complete way.

Each use case represents a coherent set of functional system requirements. Usually, more than one function is related to a single use case, especially if it is a complex one. Some functions, on the other hand, may be associated to more than one use case. In some situations it may also happen that a function corresponds to a single use case and vice versa. This usually occurs with very simple use cases such as reports and entity management.

There are at least three approaches for requirements and use cases:

- Create a list of functional requirements, and then identify use cases associated with them.
- Create a set of use cases, and then extract the functional requirements from them.
- Consider that use cases *are* requirements. The brief version of the use cases corresponds to high-level requirements and the expanded use cases correspond to the complete set of requirements.

In this book, the third approach is taken, because it is simpler, straighter, and avoids generating different documents with the same goal. Requirements elicitation and analysis corresponds, then, to the discovery of system use cases and their properties.

3.2 System actors

A *system actor* is an entity of the real world that interacts with the system through the use case. Actors may be roles performed by people such as customers, publishers, sellers, operators, etc. Actors can also be external systems, that is, systems that are outside the scope of the project being developed.

Human actors or external systems interact with the target system by sending and receiving information through an interface. In the case of human actors, the information is usually exchanged through data entry devices such as keyboards, mice, or other special devices. Those actors receive information from the system through interfaces such as screens, printers, or other special devices.

Communication with actors that are external systems usually happens through a computer network. In this case, the communication interface consists of the network and

its protocols.

The idea of external systems as actors must not be confused with internal ones that are components of the system under development. For instance, a database management system (DBMS), used to implement data persistence for the system under development is not an actor but a component of the inner architecture of the system. The following rules may help to appropriately identify external systems that could be actors:

- System actors are *complete information systems*, and not only libraries of classes or components. These systems store their own information, which can be exchanged with the system being developed. That information may change independently from the system being implemented.
- System actors are *out of the scope of development*, that is, the team will not necessarily have access to the internal design of these systems, or the ability to change it. The team must communicate with an system actor using the system actor's own definitions, because they cannot be modified.

Some approaches consider that there are two kinds of actors: *primary* and *secondary*. Primary actors are the ones whose goal the use case is trying to satisfy, while secondary actors are actors that just provide some service to the process under discussion. Examples of secondary actors are printers, web services, or people that must provide information or confirmation, as for example, a clerk that confirms that the client has paid in cash. In this book we do not follow that distinction: we distinguish use case actors into *human* or *system* actors only.

3.3 System use cases

System use cases differ from business use cases ([Section 2.3](#)) in some aspects. For example, business actors may spend days or even weeks performing a business use case, while system use cases are often executed in a short period of time, usually minutes, with one or a few actors interacting with a system and obtaining a consistent and complete result for at least one of their goals. System use cases also must be performed without interruptions while business use cases are not restricted in that respect.

Another fundamental difference between a business use case and a system use case is that the business use case is usually performed by many human actors, while the system use case is normally performed by a few (sometimes just one) human actors. The fact is that if a system use case is going to be performed by more than one human actor, then they should be interacting at the same time with the system, and that is not a common situation. Usually, each human actor accesses the system at a time of her convenience, accessing necessary data and performing the necessary actions. This leads to a sequence of system use cases and not to a single use case, as explained later. However, sometimes more than one human actor may be available online; for example, at a supermarket when a customer is buying goods, a supervisor may be summoned to perform actions that the cashier is not allowed to do (e.g., cancel a sale). The point is that the supervisor must be available online and the other actors must wait for her to show up before proceeding.

On the other hand, external computational systems may be considered *online* actors,

because they are available continuously. For example, it can be assumed that a credit card operator is available online continuously. At the moment a customer decides to make a payment with her credit card, the system actor *Credit card operator* will be available.

A high-level system use case is represented only by a name inside an ellipse. It is usually associated to one or more actors as seen in [Figure 3.1](#).

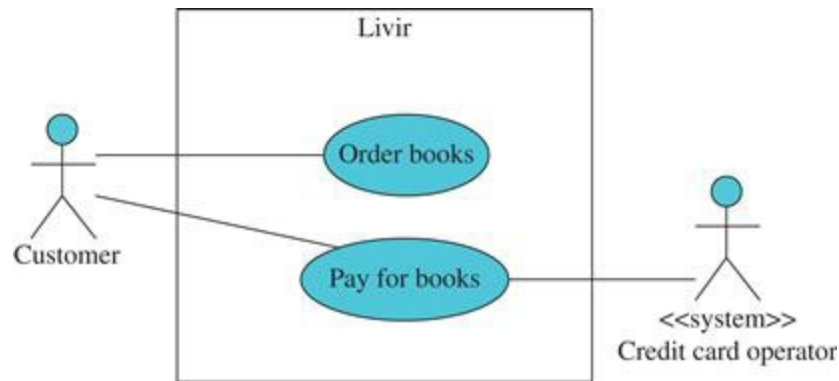


FIGURE 3.1 System use case diagram.

In the diagram in [Figure 3.1](#), the ellipses represent system use cases. By default, an actor is a role performed by a human. Other kinds of actors may be represented by the use of stereotypes, as shown in [Figure 3.1](#), where the stereotype `<<system>>` indicates that *Credit card operator* is an external system, not a human.

The use case diagram is a very popular UML diagram, but it is also frequently misunderstood. It is usual to see these diagrams with dozens of use cases and also some of their fragments attached. However, during Inception, the important thing is to know what the major processes of the system are, and not to detail them. Thus, the presence of fragments in the diagram, and the use of the *include* and *extend* relationships between use cases (which sometimes reveal part of their internal structure) is not advisable. Concerns about fragments and use cases including or extending others may be left for the moment when those high-level processes are detailed by system sequence diagrams ([Section 5.8](#)).

Usually there is not enough information to discover all of the fragments at this time. Why should only some of the use case fragments be shown and others not shown? Simply show none at all in the diagram during Inception! This prevents the diagram from having a large number of ellipses, which can hinder its comprehension. Thus, there is a need for strong criteria to decide which use cases should be maintained in the diagram to avoid, on one hand, a large number of excessively detailed processes, and on the other hand, too few processes, which could lack important features of the system.

The rule is to consider as a use case only those processes that can be performed in isolation. Partial processes that must *necessarily* be performed during other processes must not be represented in the system use case diagram.

However, even following this guide the number of use cases in a real-world system may still be too high, so that dealing with them may become difficult. To reduce the number without losing information and precision a second rule may be used. It consists

of grouping use cases that are somewhat related, especially if that may be done more than once, like a pattern. For example, there could be four use cases such as *Create book*, *Retrieve book*, *Update book*, and *Delete book*, or just one use case called *Manage book* that includes the four single processes. This is a pattern, because it may be repeated for other concepts: *Manage publisher* may be used instead of *Create publisher*, *Retrieve publisher*, *Update publisher*, and *Delete publisher*. This pattern is known as *CRUD*, which is an acronym for *Create*, *Retrieve*, *Update*, and *Delete*.

The next subsections detail those rules and present other criteria for achieving the best use case granularity.

3.3.1 Single session

A good system use case must be performed in a single session.² This means that it should begin and finish without interruption. For example, the registration of an order is made during a single session of the system, involving the identification of the customer, book selection, price visualization, payment, address selection, etc. Each of these aspects is a functional requirement of the system.

Use cases must be performed as single processes. Processes that can only happen in the context of other processes are just fragments, not use cases.

In the case of the Livir system, calculating taxes is something that will happen only during the process of ordering books. Let's suppose that the requirements of the system do not determine that it must be considered an independent process (although it could be, if requirements were different). In this case, *Calculating taxes* is not a use case, and because of that fact, it should not be included in the diagram. [Figure 3.2](#) illustrates the situation that must be avoided – using fragments in the diagram.

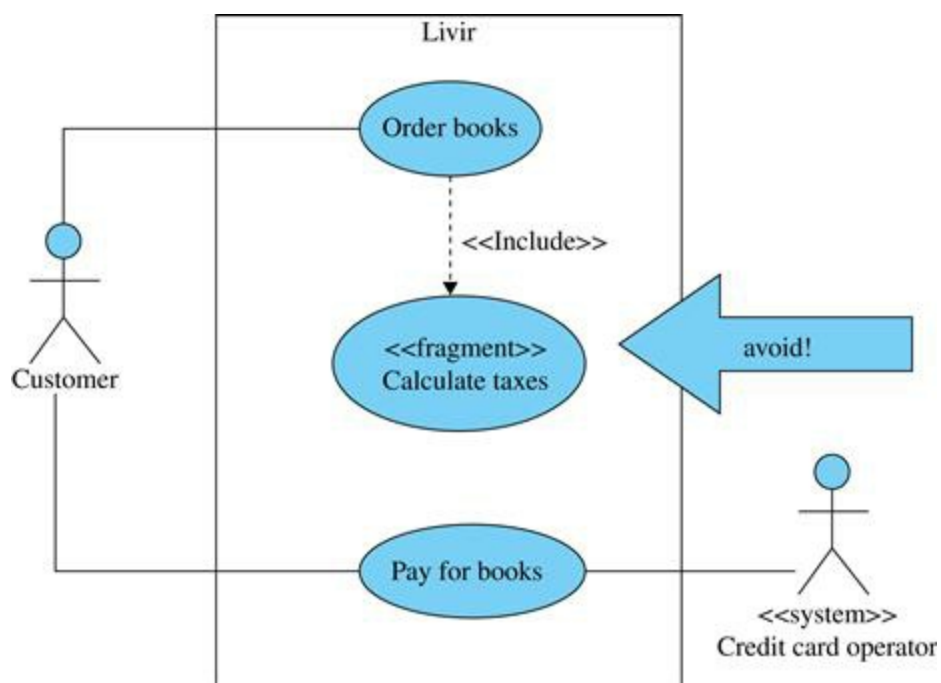


FIGURE 3.2 Example of a use case fragment.

On the other hand, *Order books* can be considered a use case because it is a process with a well-defined start and finish, it occurs in a contiguous time interval (without interruptions), and it produces a sound result (the order is registered).

In the example represented in [Figure 3.1](#), there are two possible scenarios that could be investigated:

- The system will only confirm the order if the payment is made online (it is not possible to save the shopping cart). In this case, the process of ordering books and paying are just fragments of a single use case that could be called *Buy books*. If this is the real requirement, then the diagram must be changed and the two use cases replaced by a single use case named *Buy books*.
- The system will register the order, but not necessarily proceed to checkout, that is, the shopping cart may be stored until the customer decides to finish the order (which can be done immediately or at some other time). This is the case represented in [Figure 3.1](#). In this case, *Order books* is one use case and *Pay for books* is another use case. They can be performed immediately one after the other, or with an interval of some days, and because of that, they should be considered independent use cases.

On the other hand, books are not automatically delivered when the order is paid: someone is responsible for verifying, from time to time, which orders have been issued; that person collects books in stock, and sends the package. The delivery process may happen just after the order is paid, but it can also happen on the next day (or even six to eight weeks later). This fact characterizes *Delivery* as a different use case. The *Delivery* use case has buying books as a precondition, but buying is not part of delivery: delivery just happens after it. Thus, buying and delivering, although sequential and even part of the same business use case, must be considered as different system use cases, as shown in [Figure 3.3](#) where the *Send books* use case is introduced.

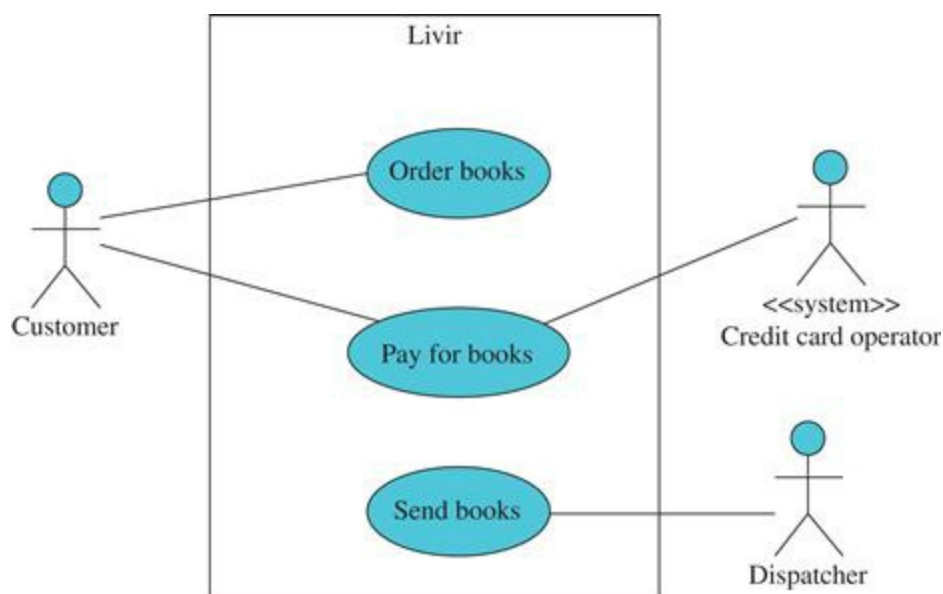


FIGURE 3.3 Use cases related to the business process of selling books.

3.3.2 Interactive

A use case must also be interactive, meaning that an actor must exist to interact with the system. The internal processes of the system are not use cases no matter how complex they are. On the other side, a simple query on some information may be a use case if there is an actor that starts it.

The aforementioned internal processes are either part of a use case (as, for instance, *Calculate taxes* is part of *Buy books*), or supplementary requirements (as, for instance, *Store data at a relational database*). In the case of supplementary requirements, they are usually implemented by internal mechanisms that do not depend on user interaction. During the Inception phase they are recorded as supplementary specifications (Section 3.4.8) or nonfunctional requirements (annotations on use cases – Section 3.4.5), so that the risk and effort that they imply is not underestimated.

3.3.3 Consistent result

A use case must produce a consistent result, be it a complete entry or transformation on a piece of information, or simply a query where relevant information is passed to a user. A use case cannot leave the information at an inconsistent state at its end. For instance, the registering of an order cannot be concluded without the identification of the customer and the books she is ordering, or else the information about the order would be incomplete regarding the business rules; the bookstore could not resume the order or charge it without knowing who the customer is and which books are ordered.

To decide if a use case has a consistent result, one could think like this: only a complete process is a system use case, in the sense that a user could go to the computer, turn it on, perform the process, and at its end, turn off the computer, because the process is complete, and some business goal was obtained (some relevant information was received or updated by the user).

This excludes from the use case definition fragments such as *Calculate taxes*, in the case of the bookstore example, because those taxes are calculated only inside the ordering process, and not as an isolated process. That also excludes operations such as *Login*, because performing a login and then turning off the computer cannot be seen as a complete process. It may be just part of one or more use cases (Figure 3.4).

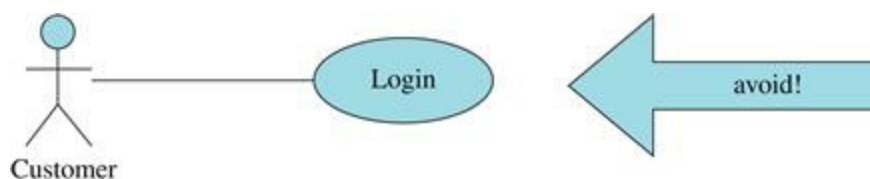


FIGURE 3.4 Login must not be considered a complete use case.

To avoid representing *Login* as a use case or even as part of one it may be simply assumed that every system operation could only be performed by a user that is regularly

identified and authorized to perform it. This mechanism is part of the technological design of the system and does not need to be considered in the high-level use case model.

On the other hand, it is possible that complete use cases occur inside other use cases. For example, the process of registering a customer can be considered a complete use case. But that process can also occur during book ordering, especially if it is the first time that the customer is using the system, or when she needs to update data (Figure 3.5).

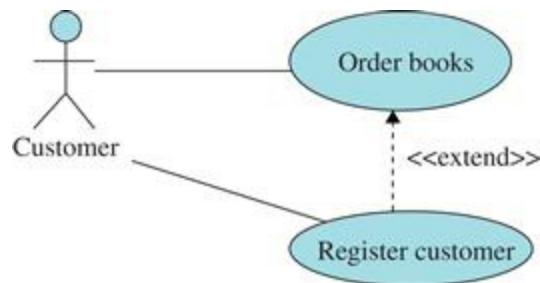


FIGURE 3.5 Independent use cases that may be related.

The dependency association between use cases is stereotyped with `<<extend>>` to indicate that the process of ordering books may occasionally be extended by the process of registering the customer. Both processes may occur independently from each other, but the customer registration can also occur during book ordering.

To avoid abusing `<<extend>>` in the use case diagram, it is important to keep in mind that only complete use cases must be maintained in the diagram. Fragments like those shown in Figures 3.2 and 3.4 must be avoided, even if they are extension points of a use case. These fragments will be addressed appropriately later, when the high-level use cases are detailed (Chapter 5). In practice, at this point, any association between use cases such as *extend* or *include* should be avoided in the diagram because they usually add no useful information for the Inception phase.

3.3.4 Essential

Two styles for writing use cases may be identified:

- *Essential use cases*, which do not mention interface technology.
- *Concrete (or real) use cases*, which are specifically written for a given interface technology.

During requirements elicitation and analysis, system use cases are considered requirements, not design. It is a common mistake to include among these use case actions that are purely related to an interface technology (such as *Open main window*, *Print report*, and *Login*). People prepared to handle that facet of design will decide on those actions later, after the requirements are discovered.

Ambler (2000) points out that essential models are more flexible, leaving more options open and more readily accommodating changes in technology. He also says that essential models are more robust than concrete representations because they are more likely to

remain valid in the face of changing implementation technology.

Thus, essential use cases must be considered as the correct option during requirements elicitation, although technology details or options may be annotated to allow risk management and effort calculation. Later in this book ([Section 5.4.1](#)), the difference between essential and concrete use cases will be explained in more detail.

3.3.5 Brief

During Inception use cases usually are brief, meaning that they are described just by their name or, in some cases, by one or two sentences. However, this is not the only way a use case may be described. Later they will be extended and contain more details about the requirements ([Chapter 5](#)). [Cockburn \(2001\)](#) identifies three types of use cases with regard to the level of detail:

- *Brief*: A one-paragraph synopsis of the use case.
- *Casual*: Written in simple, paragraph, prose style. It is likely to be missing project information associated with the use case, and is likely to be less rigorous in its description than a fully dressed use case.
- *Fully Dressed*: Expanded to include a main flow and alternate flows, as well as other sections such as postconditions, preconditions, stakeholders, and technological variations.

In this book we expect that use cases considered during the Inception phase will be brief. This means that usually their name is sufficient to explain their meaning. However, additional explanation is allowed for the sake of requirements comprehension. Also, if some key use cases have to be expanded in order to identify risks regarding their inherent complexity, this is also acceptable. Usually, fully dressed or expanded use cases will be useful only after Inception, when requirements must be detailed adequately.

3.3.6 System boundary

One of the decisions a team must make when designing system use cases is where to place the system boundary. Graphically, the boundary is only a rectangle that is placed on the diagram. Inside it are the use cases and outside are the actors. In business use case diagrams, the system boundary represents the limits of the organization (company, department, etc.). Here, it represents the limits of a computational system.

A decision was made earlier on which workers to include in the automation border. One example is the clerk that helps the customer buy books. In that case, when modeling system use cases the *Clerk* disappears, and the use case is performed by the customer itself.

However, if it was not a virtual bookstore, but a regular one where customers buy at a store, should the clerk be kept in the use case diagram? For a use case to be as essential as possible, it is recommended that only actors that are really interested in the use case be kept in the diagram. The clerk of the example is just a proxy of the customer; the clerk has no personal goals in the process of buying books. It acts simply as a system interface.

Whether the bookstore is virtual or not, the essential use case should be the same, because the same information would be exchanged between the customer and the system; the clerk would only transmit information as she receives it. In this case, the only actor would be the customer and the clerk should not even appear in the diagram. In this way, the analysis will produce use cases that are technology independent.

Someone could ask the following: What if the clerk must indicate that she assisted in the sale so that a percentage would be paid to her? In that case, what we have is a use case that is different from the one mentioned in the last paragraph. Here, both the customer and the clerk *should* be actors: the customer is interested in the books and the clerk is interested in the percentage. Also, if the clerk may perform actions that are not allowed for the customer (e.g., overriding a product price) she must be considered an actor and may not be removed from the diagram.

3.4 How to find system use cases in the business model

In order to discover system use cases and actors, the team may examine the business use case diagram with the defined automation boundary (scope of automation).

First, the actors that are really interested in the process to be automated should be identified. These are the business actors that interact with the business use cases inside the automation scope, and the business workers that interact with such use cases but that are not themselves inside the automation scope. In the case of [Figure 2.11](#), the following system actors could be identified:

- *Customer, Publisher* and *Credit card operator*, because they are business actors that interact with use cases that will be automated.
- *Acquisition manager*, because it is a business worker that will not be automated, but that interacts with a use case that will be.

On the other hand, the following will not be considered system actors:

- *Clerk*, because it is a business worker that will be automated.
- *Marketing Department*, because it is a business worker that does not interact with any use case inside the automation scope.

The business workers whose functions will be partially or totally supported by the system may be important sources of information, because knowing how they perform their duties currently may be crucial to understand how the system must work. Moreover, knowing how the duties are really performed may be valuable because sometimes what happens in practice is not what is written in the procedure books.

Another source of requirements is the business actors that will become system actors. However, they will not always be available for requirements elicitation, and, in that case, they should be replaced by domain specialists. For example, it may not be possible to interview the customer of a bookstore that does not exist yet. Thus, someone that knows the business and can explain how it is supposed to work will be a key person for obtaining the right requirements. If the publishers and credit card operators are also not available, it still would be possible to examine existing documentation on the communication protocols and services offered, so that the interfacing requirements with

them can be known. In the case of the customer, however, such a possibility would not be feasible.

Once system actors have been identified from the business use case diagrams, it will be possible to observe the activity and state diagrams that were produced during business analysis in order to verify which activities performed by the system actors can be considered system use cases.

By analyzing the activities of [Figure 2.10](#), we can see that the activities performed by the *Customer* are:

- *Order books.*
- *Pay for books.*

The only activity of the *Publisher* is *Send books*, and the only activity of the *Credit card operator* is *Confirm payment*.

Not every activity of the original *Clerk* becomes a system use case. As seen in [Figure 2.7](#), the clerk was split into two actors: *Acquisition manager*, which will not be automated, and *Clerk*, which will be automated. The activities performed by the *Acquisition manager* role may become system use cases. However, the activities performed by the business worker *Clerk*, which is being automated, become internal actions of the system. For example, the activities *Register order* and *Inform availability and total* in [Figure 2.10](#) do not become system use cases; they are just part of the use case *Order books* initiated by the customer. However, *Order books from supplier* is a system use case whose actor is the *Acquisition manager*.

Regarding processes that will be identified as use cases, one must still consider two concerns:

- Do the activities connected by a flow necessarily occur immediately or can they occur in different moments? In the example, are *Order books* and *Pay for books* activities that must occur in a single session of use of the system, or can they be performed in different moments? The answer will depend on the way the business is organized, that is, it will depend on the business rules. If the company decides that the order is received only after the payment is made, then those two activities form a single use case. However, if the company decides that it is possible to register an order (save a shopping cart) and pay for it some other day, then there are two different use cases.
- Are the activities selected complete and sound? For example, is *Send books* a good description of the activity performed by the publisher? By the similarity to the process performed by the customer, it can be inferred that the publisher will first receive an order, and then it will send the books it has in stock. However, the payment in the business-to-business case is usually not made immediately – an invoice is generated and paid by the bookstore with a predefined deadline (for example, four weeks after purchasing).

At this point, the diagram of [Figure 2.10](#) should be revised. The publisher does not simply *Send books*. It is going to receive the order, send an invoice for payment, and finally send the books that are available. The books must be received and registered by some worker at the bookstore. That worker will be a new system actor and could be called *Deposit worker*.

A recurrent question is if actors correspond to security profiles. This is not necessarily the case. The goal behind the identification and modeling of actors is more related to the process of finding and organizing requirements than to the process of granting access to the system. A good software design will treat access permission in a dynamic way, allowing user profiles to be created, and permissions to be associated to the different profiles of individual users. As that aspect of the system is not domain dependent it is not adequate to model it within the domain. In other words, use case actors should not be considered as security profiles. Those profiles will be created dynamically with the use of a generic domain-independent mechanism.

The state machine diagram presented in [Figure 2.15](#) is also particularly helpful in terms of system use cases that must be analyzed. Each change of state of a book (main business object), must be done by someone, and that “someone” is an actor performing a use case. Thus, from the transitions in that diagram a new set of system use cases was discovered as shown in [Figure 3.6](#).

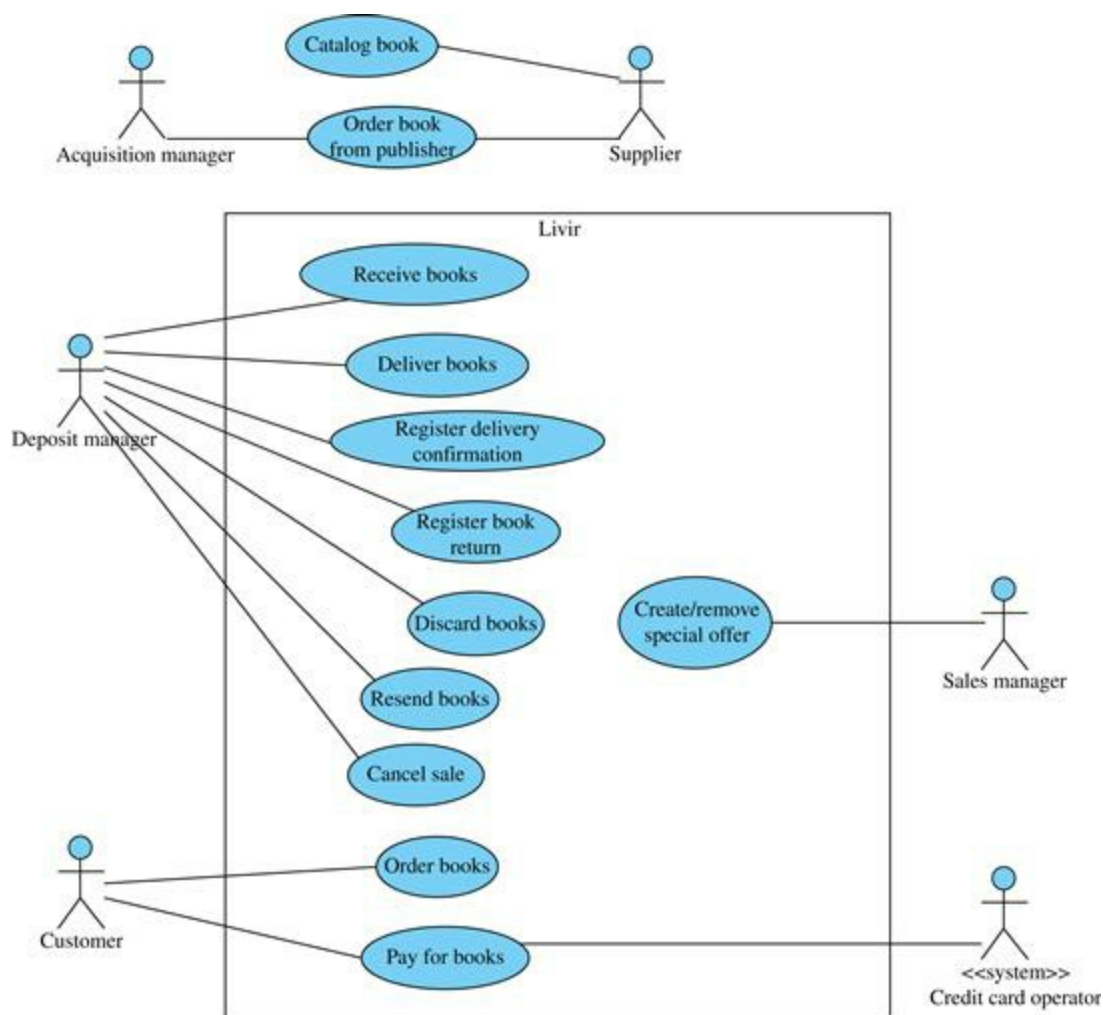


FIGURE 3.6 System use case model completed with information discovered on the state machine diagram of [Figure 2.15](#).

In [Figure 3.6](#), the use cases *Catalog book* and *Order book from publisher* are maintained outside the scope of the Livir system. This is due to the fact that it was discovered at a point that such use cases would be performed within the publisher system, not within

the system under development. As they are not included in the bookstore system, they may be maintained in the diagram, but outside the system boundary, because they are out of scope and will not be implemented.

After making sure that it is understood that they will not be implemented, they may be removed from the diagram, in order to simplify it and avoid later confusion with people who may not understand that what is outside the boundary will not be implemented. In that case, their names may be listed in an *out of scope list*, for recording purposes.

The use cases associated with the new actor *Deposit manager* are necessary to cover most of the transitions indicated in the state machine diagram of [Figure 2.15](#). It is interesting to notice that some transitions are performed outside the scope of the company. For example, when a customer decides to return a book, this is not done through the bookstore system; she simply sends the book back, usually by mail. The information regarding the return only reaches the bookstore when it receives the book. Thus, the use case referred to here is *Register book return* and not *Return book*, because the registry is made by the deposit manager, and not by the customer. This is similar to the registry of the arrival of books, represented by the use case *Receive books*, because when the publisher sends the books it cannot be registered in the bookstore system directly (unless both systems are integrated). Only the arrival can be registered by the bookstore. This is a fundamental difference between a business use case and a system use case: in a business use case the process as it happens in the real world may be described as part of the use case, but when the focus is on the system, only processes that involve the system may be represented.

3.5 Requirements

Identifying system use cases is an activity that is part of the *Requirements* discipline of the Unified Process. Requirements elicitation and analysis is a significant part of the Inception phase. The team can and must use every available source to identify requirements (specialists, users, documents, interfaces, literature, etc.), and, for each source, a set of functions that the system must perform may be identified.

3.5.1 Requirements elicitation

Requirements elicitation corresponds to the search for information about the functions that the system must perform, and for the constraints under which the system must operate. In the approach to requirements presented in this book, they will be recorded in the system use cases.

Another way to register requirements without use cases would be by using a document that consists of a list of functional requirements, possibly accompanied by a list of constraints ([Sommerville, 2006](#)). The advantage of using use cases instead of a list of functions is that a good use case has a well-defined granularity, which allows for the generation of a high-level requirements list that is much more comprehensible than a list of individual functions. Usually the number of high-level use cases is much lower than

the number of individual functions. The individual functions will appear in the use case expansion, when the high-level use cases have their structure detailed.

In the case of the Livir example, requirements elicitation will allow the team to discover that the system must control buying and selling books, receiving payments, allow damaged books to be registered, generate sales reports, verify if books are available in stock, etc. These operations and many others will constitute the functionality of the system, and this is why these requirements are called functional requirements. These functions will be incorporated into one or more use cases.

On the other hand, during requirements elicitation the analyst may face business rules or constraints on how functions must be performed by the system. For example, a business rule could state that the bookstore only sends books after the payment is confirmed. That kind of rule is a nonfunctional requirement that can be recorded as an annotation or remark on the use case itself, in order to be recalled and verified when the use case has to be detailed (Figure 3.7). Alternatively it could be recorded separately as a numbered list or spreadsheet, with a reference to the use cases through a unique number.

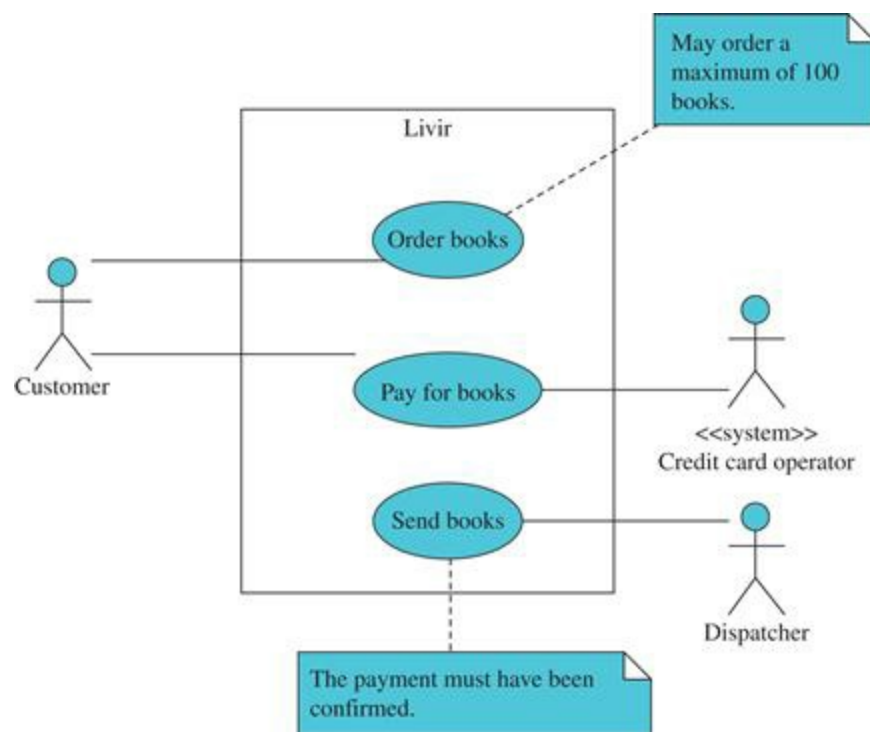


FIGURE 3.7 Nonfunctional requirements annotated in use cases.

As can be seen in Figure 3.7, the use case diagram with notes will quickly become too complex to be useful. This is why it is recommended that such annotations should be recorded inside the use case specifications or in a separate document. Most CASE (Computer-Aided Software Engineering) tools allow that a diagram element such as a use case have a specification window as shown in Figure 3.8. One exception to that rule would be the case when the annotation is absolutely required to help understanding the diagram. However, this is seldom the case.

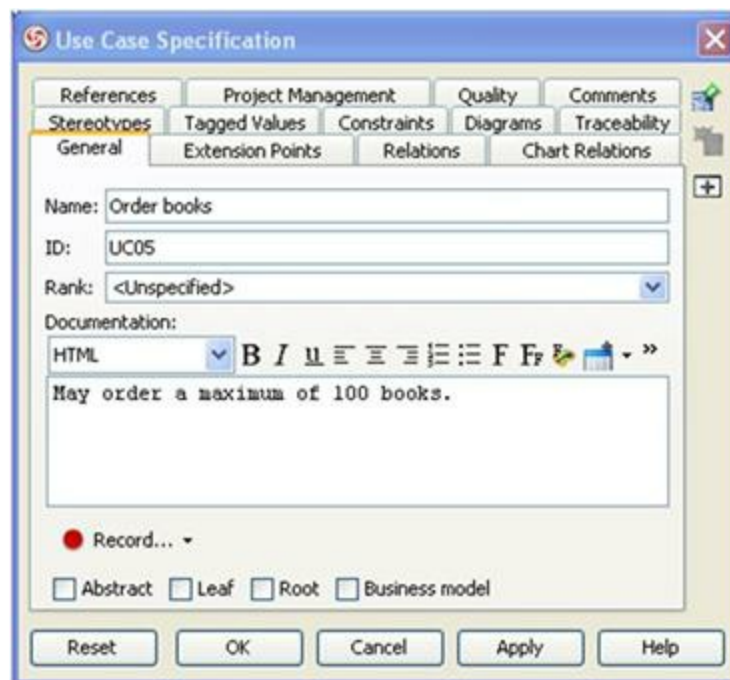


FIGURE 3.8 Example of specification window for a use case. Image produced with the CASE tool Visual Paradigm™ Community Edition.

3.5.2 Eliciting requirements is not design!

A system that is going to be analyzed is like a forest. To explore an unknown forest one cannot start by examining each plant and animal. There is a saying that states that some people cannot see the forest for the trees. The forest is the system and the trees are the requirements. Only at the end of the process could a team say that they acquired knowledge about the smallest parts. First, a view of the whole must be taken, and only after that the details may be studied.

Thus, the Inception phase must provide a view of the whole – so that what is more important can be seen first – and then the whole can be divided into parts so that the details can be analyzed and finally a solution designed. The organization of iterations in the Elaboration and Construction phases correspond to dividing the forest into sectors, in order to see each sector at a time, and in that way, being able to deal with the inherent complexity. Thus, one of the final goals for the Inception phase is to organize work division in terms of use cases, which will be explored during the upcoming iterations.

During Inception, requirements elicitation must be quick and generic. The right way to do this is to look at the extension of the requirements, not their details. The analyst must understand the extension of what the system must do, without detailing how it will do that. Only during the Elaboration iterations will the requirement analysis be deepened.

The time used for requirements elicitation time must be about discovery, not invention. During this period, the team of analysts, with the clients, users, and other stakeholders will try to list most of the capacities and restrictions with no concern about detailing them. Details about requirements will be conveniently accommodated during the next iterations.

It must also be clear that requirements are something that a client asks for, and not

something that the team designs. Some analysts mix requirements gathering – that is, the memory of the demands of the client – with the beginning of the system’s design. One example of this kind of confusion is a set of requirements involving relational database design. Unless there are legacy systems that must be compatible to the new system, how can it be justified that a set of relational tables is a client demand? That could eventually be possible with some sophisticated clients with a degree in Computer Science, but it is not the general rule. Database tables are part of the *solution* domain, not the *problem* domain. The analyst must seek the requirements that correspond to the client needs and goals in terms of information. Later, she may decide if that information would be stored in a relational database or in another structure.

3.5.3 Requirements challenges

The requirements document, which can be formed by the use case diagrams with annotations and by the supplementary specifications, must register the capacities of the system and the conditions under which it must operate. The challenges regarding requirements are, at a minimum, the following (Pressman, 2010):

- How to *discover* requirements.
- How to *communicate* requirements to the other phases and teams of the project.
- How to *recall* requirements during development in order to verify if they have all been implemented.
- How to *manage* requirements change.

It would be useless to develop a nice use case diagram, and later not be able to know if the requirements incorporated there were included in the design. The existence of automatic mechanisms to conduct this verification is crucial. Therefore, it is important to maintain traceability relations between use cases and other parts of the design. The following chapters show how those relations among design artifacts are obtained.

It is necessary to keep in mind that requirements necessarily change during the development of a project. Thus, the change must be managed, not avoided, after Inception.

Sometimes, requirements change after the system is deployed. Context conditions, regulations, company policies, or work methods may change anytime. Although the analyst cannot foresee these changes, mechanisms to accommodate them may be created in order to ease the change process when it is necessary. There are specific design patterns to deal with these requirement instabilities (as, for instance, the *Strategy* pattern, presented in Section 7.6). The changes are totally unpredictable. If the system is not structured to accommodate the changes in requirements it would probably be hard to implement them.

This kind of situation makes *requirements-driven analysis and design* processes (Alford, 1991) inadequate for most systems. Using requirements as a base to support system architecture is like building a house over moving sand; when requirements change, the structure suffers. However, the Unified Process (UP) uses a much more stable basis for the architecture, which is based on classes and components that encapsulate information

and behavior.

These classes implement functionalities that combined allow the implementation of the requirements. If requirements change, the combinations change, but not the basic structure. This kind of architecture follows the *open-close principle* (Meyer, 1988), in the sense that it is always closed for modification (it works), but open for extension (it may accommodate new functionalities).

It is important to trace the origin of each requirement (e.g., a business actor, a business worker, the client itself, or even a domain specialist), because it is necessary to validate requirements with those sources, verifying if they are well written, complete and sound.

Sometimes it may also occur that different people or departments present different specifications for the same requirement. In that case, it is necessary to produce an agreement among them, or identify who has the highest authority to determine the acceptable form for the requirement.

3.5.4 Evident and hidden functional requirements

Functional requirements can be optionally identified as *evident* or *hidden* (Gause and Weinberg, 1989):

- *Evident* functional requirements are functions that are performed with the user's knowledge. These requirements usually correspond to information exchange between the user and the system, such as queries and data entry, which flows through the system interface.
- *Hidden* functional requirements are functions performed by the system without explicit knowledge of the user. Usually these functions are math operations and data updating performed by the system without explicit user knowledge, but as a consequence of other functions performed by the user.

Hidden requirements are performed internally by the system. Thus, although they do not appear explicitly as use cases, they must be adequately associated to them in order to be recalled at the time of design and implementation. Thus, they could also be added as annotations to a use case.

An example of an evident requirement is the production of a report on bestselling books, which is something a manager might require. An example of a hidden requirement is applying a discount policy to a sale. In that case, the user does not explicitly request that the system perform the operation. As it is an activity that the system performs automatically, it is a hidden requirement.

3.5.5 Nonfunctional requirements

Nonfunctional requirements are constraints or qualities that may be linked to specific functions of a system (for example, "an order cannot contain more than one hundred books," "the order transaction must be preserved if communications are broken," etc.), and therefore they can be treated as annotations on the use cases that incorporate the respective functions. Sometimes, however, nonfunctional requirements may be general,

that is, not necessarily attached to a function (for example, “the system must be implemented in Java”), and in that case, they will appear in the supplementary specifications document. Constraints and qualities that are specifically attached to a function are called *nonfunctional requirements*, and general constraints and qualities are called *supplementary requirements*.

There are two kinds of nonfunctional requirements:

- *Logical issues*: Business rules attached to a function. For example, during sales registration, a series of constraints could be considered, such as not closing the sale until the credit card operator confirms the payment, or not closing the sale if the last delivery to the same address has been returned due to invalid address.
- *Technological issues*: Constraints and qualities related to the technology used to perform the function, such as, for example, the user interface, the kind of communication protocol, security constraints, fault tolerance, and so on.

For example, establishing that the user interface to perform an order must follow a design pattern based on a sequential flow of screens is a technological constraint (interface constraint) about the way that function must be performed. Another example of a nonfunctional requirement is “the payment confirmation must not take more than 5 seconds.” This is a technological constraint related to system performance and it would affect the way a designer would think about the communication mechanism with the credit card operator. In this case, the system design would have to consider seriously the broadband connections to the operators.

A requirement does not establish how the constraint will be implemented, it just demands it. The design and implementation of the system must meet the requirement in one way or another, or otherwise the analyst should negotiate with the client for some requirement flexibility.

3.5.6 Permanence and transience of nonfunctional requirements

One of the most fundamental features of a requirement is whether a given nonfunctional or supplementary requirement is *permanent* or *transient*.

Nonfunctional and supplementary requirements can be considered permanent (they are not expected to change) or transient (they are expected to change) depending on a decision made by the client. Permanence or transience is not an intrinsic feature of a requirement: it is decided depending on the convenience. The same given requirements can be considered permanent or transient depending on what is desired regarding the time and cost of software development and maintenance.

If one invests in flexible design, so that most constraints are transient, less effort will be spent during maintenance in order to accommodate changes. However, the cost of that flexible design during development may become prohibitive for some projects. It will always be a good idea to ponder which constraints should really be treated as transient requirements.

For example, a supplementary requirement could establish that the Livir system must

deal with a single currency: the dollar. If that requirement is considered permanent, then the system will be designed for a single currency (“dollar” could even be a data type used to define variables and attributes). However, if the requirement is considered transient, then even if there is no other currency being used today, the whole system must be prepared to accommodate future currency, or even more than one currency at a time.

The consequences of deciding that a requirement is *permanent* are the following:

- It is cheaper and quicker to develop the system.
- It is more expensive and difficult to change the system if, by any chance, the requirements change in the future.

On the other hand, deciding that the requirement is *transient* has the following consequences:

- It is more expensive and complex to develop the system (it should accommodate functionalities for changing the currency, for example).
- It is easier and quicker to maintain the system (if the currency changes, the system is already prepared to accommodate that with a simple reconfiguration).

Thus, it is not the nature of the nonfunctional requirement that will decide whether it is permanent or transient. It is the client, with the help of the team, that must make the decision. The ideal situation would be to list the requirements of greater importance (those that are really expected to change in the near future with a bigger impact on the system) and consider them transient, while leaving the others as permanent.

3.5.7 Mandatory and desired requirements

Requirements may also be considered *mandatory* or *desired*,³ that is, those that must be obtained by any means and those that could be obtained if no bigger issue hinders the development process.

In the case of functional requirements, that classification indicates development priorities. If there is flexibility in the contract such that only the most important use cases are implemented if there is no time to implement them all, then the team must know which ones are mandatory.

However, if the team is able to make a good estimate of the effort needed to develop the system, and if they have a good history of accurate estimation, there will be less motivation for such distinction regarding functional requirements, because all requirements are expected to be implemented in time.

On the other hand, nonfunctional and supplementary requirements are much more unpredictable than functional ones with regard to effort estimation. Thus, in some cases, it may be necessary to consider those requirements with some flexibility.

In this case, some restrictions are defined such that they must be obtained by any means and others may be considered simply desirable, and some time is allocated to pursue them.

For example, in the case of the Livir system, the requirement for using a web interface may be considered a mandatory supplementary requirement. In that case, other solutions are not acceptable. However, additional access through a cell phone could be

considered as desirable requirement, because such access is not absolutely necessary for the effective success of the system.

Nowadays, with the formalization of software development contracts, there is less flexibility regarding desirable requirements. In most cases, the developer must state which requirements will be implemented, how much time it will take, and how much it is going to cost. Stepping outside that line may require the payment of fines or even canceling the project.

3.5.8 Supplementary requirements

Supplementary requirements are all types of constraints and qualities related to the system as a whole and not only to individual functions. For example, a supplementary requirement may establish that the system must be compatible with a given legacy database, or be implemented with a given programming language, or even follow a given *look and feel*.

Care must be taken when supplementary requirements are defined. A requirement such as “the system must be easy to use” is not sufficiently clear. It would be better to say something like “novice users must be able to complete tasks without errors on their first attempt.” That gives a more precise idea of what must be designed to accomplish the requirement.

Nonfunctional and supplementary requirements may also be identified with different groups such as, for example, interface, implementation, performance, fault tolerance, etc. The goal of making such distinctions is to allow for better organization.

Although most UP practitioners would choose to use the *FURPS+* classification system (Grady, 1992) for organizing supplementary requirements, the newest source to decide on supplementary requirements classification is ISO/IEC 25010,⁴ as shown in Table 3.1.

Table 3.1

Supplementary Requirements Classification Based on ISO/IEC 25010:2011 and Requirements Generating Questions

25010 Feature	25010 Subfeature	Requirements Generating Questions
Functional suitability	Functional completeness	Are these all necessary functions that must be implemented? May some of them be left outside the system scope? May some functions be implemented by a further project?
	Functional appropriateness	To what degree must the user activities be made easier by the system?
	Functional correctness (accuracy)	Are there accuracy specifications, that is, a desirable precision for data? Are there limits for tolerating imprecision?
Reliability	Maturity	Must special care be taken to prevent the system from presenting defects during use? Must critical parts be defined by formal specifications? What is the intensity and type of tests that must be performed to assure the system is free from bugs?
	Availability	What is the degree of availability required for the system? How many simultaneous accesses must be supported? How many hours per day? How many days per year?
	Fault tolerance	How should the system react in the case of externally provoked anomalies, such as communication interruptions?
	Recoverability	Must the system recover automatically in the case of a disaster? Under which circumstances must lost data and aborted processes be recovered?
Usability	Appropriateness recognizability	In what way should the software present itself to a potential user so that she may recognize its applicability? How should the software be packaged?
	Learnability	How are the concepts inherent to the software being presented to the user so that she can become competent in its use?
	Operability	To what degree must the product be easy to use and control? What kind of help must the system provide? What forms of documentation and manuals should be available? How are they going to be produced? What kind of information should they present?
	User error protection	What kind of protection against user error is necessary?
	User interface aesthetics Accessibility	Which design patterns will be used for the interface to provide visual pleasure and satisfactory interaction? To what degree must the product be designed to attend to people with special needs?
Performance efficiency	Time behavior	Which time restrictions exist related to the software processes and functions?
	Resource utilization	Are there data storage space restrictions? Energy limitations? Communication network limitations?
	Capacity	Regarding processing capacity, what are the nominal values expected and what are the critical values? For example, the software may be designed to support up to 2,000 simultaneous

(Continued)

25010 Feature	25010 Subfeature	Requirements Generating Questions
Security	Confidentiality	accesses, but assured to keep working when up to 10,000 simultaneous accesses occur. To what degree must the information and functions of the system be available only to those authorized to access them?
	Integrity	To what degree must data and functions be protected against unauthorized modification by people and systems?
	Nonrepudiation	To what must the system guarantee that the records kept by it are effectively true so that their authors cannot deny them?
	Accountability	To what degree are user actions recorded by the system? What information is kept?
	Authenticity	To what degree must it be guaranteed that a logged-in user is really who she is supposed to be?
Compatibility	Coexistence	With which products must the software potentially coexist? Which tools and programming languages must be used? Is it necessary to run the system with legacy systems?
	Interoperability	Which other products must communicate with the system? To which systems must it send data? From which systems must it receive data?
Maintainability	Modularity	What kind of architecture will be used? Layers? Partitions? Components? Web services?
	Reusability	Will the system be produced from parts of other systems? Must the system be designed so that its parts could be reused in future projects?
	Analyzability	Will special techniques or tools be used to ease debugging of the code?
	Modifiability	Will special techniques or tools be used to guarantee that changes introduced in the system will not produce new defects?
	Testability	Will special techniques and tools be used to ease regression tests? Will automated test tools be used? Will TDD (test-driven development) be used? How are test assets produced during development going to be kept?
Portability	Adaptability	To what degree must the software adapt itself to other contexts besides the ones it was originally designed for? Must it be recompiled or just reconfigured? What can be configured in the system? Examples of configurable items are printers, currency, company policies, interface fonts and colors, language, etc. Transient requirements would probably be related to configurable items.
	Installability	What installation resources will be provided? Should installation be automatic? Is data migration automatic?
	Replaceability	What resources must the system provide when it replaces other systems with the same goals? What resources must the system provide when it is going to be replaced by other systems with the same goals? Should it generate data and configuration in formats widely understood such as XML?

(Continued)

25010 Feature	25010 Subfeature	Requirements Generating Questions
Effectiveness	Effectiveness	Which business goals on the real use environment must the system help to obtain? To what degree must the system be responsible for obtaining those goals in a complete and correct way?
Efficiency	Efficiency	What kind of return of investment (ROI) must the system produce to the client?
Satisfaction	Utility	In which way must the software be designed to help the user perceive the consequences of its use?
	Pleasure	To what degree must the system provide pleasure to its users?
	Comfort	To what degree must the system preserve or improve physical and mental user comfort?
	Trust	In which way must the system be designed so that the stakeholders trust it will do the work?
Freedom from risk	Economic risk mitigation	What kind of financial risk (including property and moral damage) must the system reduce?
	Health and safety risk mitigation	What kind of physical risk to people must the system reduce?
	Environmental risk mitigation	What kind of environmental and property risks must the system reduce?
Context coverage	Context completeness	To what degree must the system be used effectively, efficiently, without risks, and with user satisfaction in its use contexts? What are those use contexts? Are there legal requirements related to the use of the software?
	Flexibility	To what degree must the system be used effectively, efficiently, without risks, and with user satisfaction in use contexts other than the ones it was originally designed for? What are those use contexts?

Although this list is extensive, the team must keep in mind that it is only a classification to improve the ability to identify which requirements are really important. There is no need to seek nonexistent requirements, for example establishing complicated packaging requirements for a client that does not care about the way the software will be packaged.

It is also advisable not to lose time discussing if a given requirement belongs to this or that type. More important than deciding on its type is to know that it exists: long discussions on requirements classification do not add significant knowledge to the project.

Table 3.2 presents an example of supplementary requirements that could be assigned to the Livir example.

Table 3.2
Supplementary Requirements for the Livir System

25010 Feature	25010 Subfeature	Supplementary Requirement
Functional suitability	Functional completeness Functional appropriateness Functional correctness (accuracy)	
Reliability	Maturity Availability Fault tolerance Recoverability	The system must be available continuously (24 hours a day, 7 days a week). Transactions partially concluded by the user in the case of temporary faults must be preserved for recovery for at least 24 hours. After that deadline, they may suffer rollback. In the case of energy, communication, or data failure, the system must become operational again without operator intervention.
Usability	Appropriateness recognizability Learnability Operability User error protection User interface aesthetics Accessibility	All access functions that generate value for each kind of user must be easily recognizable in the initial page of the system. Primordial functions must be highlighted. Names of functions (e.g., buttons and menus) must conform to usual Windows names in order to improve users' ability to learn about their use. The use of the system must be clear to novice users: there must not be the need for supplementary help or explanation. The application must prevent the user from entering inconsistent information. The system must be configurable to many human languages.
Performance efficiency	Time behavior Resource utilization Capacity	During peak hours it is important that the system's performance does not degrade. Peaks of 10,000 simultaneous accesses are expected.
Security	Confidentiality Integrity Nonrepudiation Accountability Authenticity	Data about sales and customers are only accessible by the customer herself, the sales manager, and workers authorized by the sales manager. A customer may change her own data and open orders only. All other data may be changed only by authorized personnel. Different levels of authority may be defined. The system must store in a tamperproof record the following data about each order: its contents, the date and time it was issued and received, and the identity of the customer.

(Continued)

25010 Feature	25010 Subfeature	Supplementary Requirement
		Users' passwords must not in any case be visible by other users or system managers.
Compatibility	Coexistence Interoperability	The system must be able to communicate automatically with credit card operators to allow confirmation of payment.
Maintainability	Modularity Reusability Analyzability Modifiability Testability	
Portability	Adaptability	The server must be installed on a single platform. The client application must be accessible at least by Internet Explorer, Google Chrome, and Mozilla Firefox. The client application must be sensible to the configuration of the user computer in order to identify the language, currency, and other definitions.
	Installability	The customer may access the system via the Web without the need to install any additional component except those that are usually available for browsers.
	Replaceability	
Effectiveness	Effectiveness	
Efficiency	Efficiency	
Satisfaction	Utility Pleasure Comfort Trust	
Freedom from risk	Economic risk mitigation Health and safety risk mitigation Environmental risk mitigation	
Context coverage	Context completeness Flexibility	

Not every field was filled because, as said before, requirements must not be invented, they must be asked for by the client. Thus, usually there are not requirements in all categories. The questions that generate requirements mentioned in [Table 3.1](#) are a good basis for finding eventual needs. But it is not mandatory to have an answer to all of those questions.

3.6 Preliminary conceptual model

Although [Chapters 6](#) and [7](#) present conceptual modeling techniques in detail, it is necessary to mention here that there are interdependency relations between system use cases and the so-called *preliminary conceptual model* ([Larman, 2004](#)). The preliminary conceptual model is built during Inception, and consists of a class diagram that

represents the main information units of the system. It is still not necessary to represent attributes. Although associations must appear in that model, it is not necessary to detail their features.

By analyzing the system use case diagram, many important concepts may be discovered. These concepts are represented as classes in the preliminary conceptual model; they represent the structure of the information that will be managed by the system. At the same time, an analyst, by observing the conceptual model, could notice if the use case diagram is sufficiently complete. This verification usually occurs when new entities are identified in a business process, and it is necessary that they be somehow recorded by the system.

Figure 3.9 presents a possible preliminary conceptual model for the use case diagram presented in Figure 3.6. The process for discovering classes consists of thinking about the use cases and imagining which (high-level) information is exchanged between the actors and the system to allow the process to work. The associations between classes represent dependencies or relationships between the pieces of information represented by them.

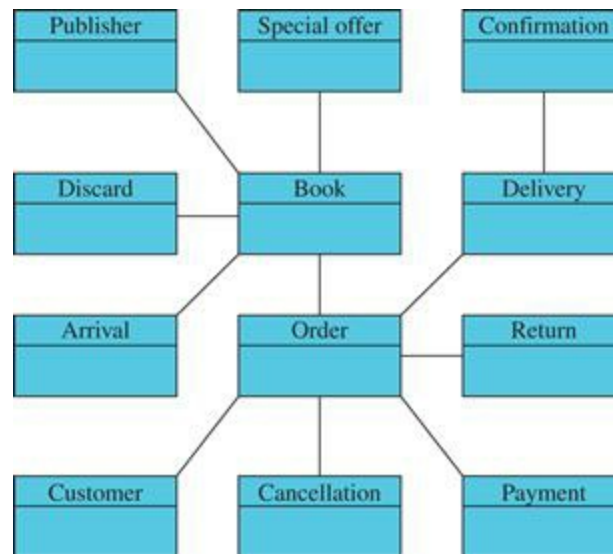


FIGURE 3.9 Preliminary conceptual model for the use cases of Figure 3.6.

Looking at Figure 3.6 with the purpose of finding the concepts shown in Figure 3.9, it can be seen that:

- *Order books* is probably the most important use case in the system. It gives rise to two concepts, *Book* and *Order*, which are associated.
- *Pay for books* adds a new concept, *Payment*, which is associated with *Order*. At that point the team could decide to change the name of the use case to *Pay order*.
- The *Receive books* use case gives rise to the concept *Arrival*, which is associated to *Book*. Maybe at that point *EntryOrder* or *BuyOrder* could be introduced as a new concept.
- *Deliver books* produces a new concept, *Delivery*, which is associated to *Order*. Now *Deliver books* may/should be renamed to *Deliver order*.
- *Register delivery confirmation* produces the concept *Confirmation*, which is associated with *Delivery*.

- *Register book return* produces *Return*, which is associated to *Order*. Maybe at that point the name of the use case should be changed to *Register order return*.
- *Discard books* produces *Discard*, which is associated to *Book*.
- *Resend books* does not create a new concept at first glance because it may be considered the repetition of the use case *Deliver books*. However, that still must be analyzed further.
- *Cancel sale* produces the *Cancellation* concept, which is associated to an *Order*. Maybe the use case name should be changed to *Cancel order* at this point.
- *Create/remove special offer* creates the *SpecialOffer* concept, which is associated to *Book*.

The identification of that preliminary conceptual model is especially useful to ease the visualization of the structure of the information that is going to be managed by the system; this helps in unifying the vocabulary among the team members and other stakeholders. The decisions about changing use case names in order to clean the vocabulary are already implemented in [Figure 3.10](#).

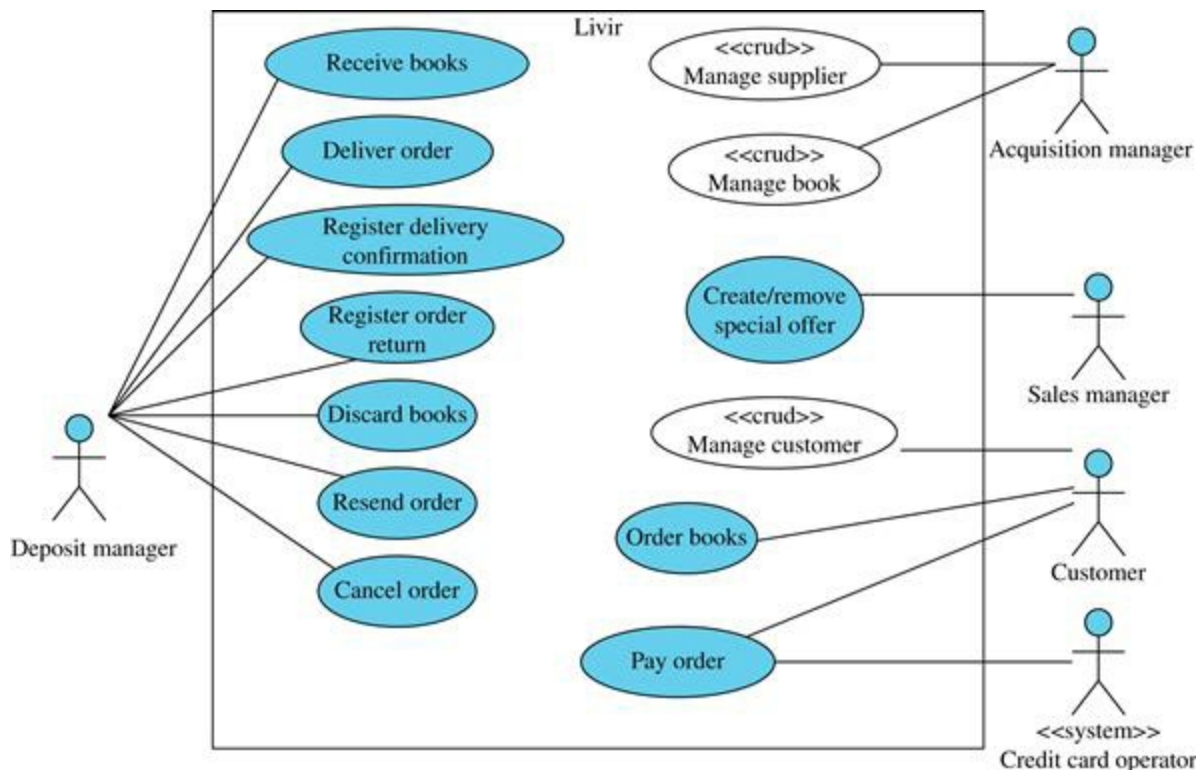


FIGURE 3.10 Use case model with names revised and CRUDs added.

But there is also the following important practical utility for the preliminary conceptual model. Among the concepts shown, most are information elements created or changed in the context of the use cases already identified. However, some of them are neither created nor changed by those use cases, and that means that some use cases could still be missing. This is the case especially for the classes *Book*, *Publisher*, and *Customer*. Those concepts may be considered as a CRUD, because they allow the four classical operations: *Create*, *Retrieve*, *Update*, and *Delete*. If they are to be added to the diagram, instead of representing them individually it would be better to represent the four operations by using a single CRUD use case, which is stereotyped with `<<crud>>`, as

seen in [Figure 3.10](#).

A new actor was defined for managing the publishers and books: the *Acquisition manager*. On the other hand, it was decided that the customer would be responsible for its own record.

Why is there no CRUD for orders, canceling, returning, payment, etc.? Because those are already managed by the use cases on the diagram, and it is not necessary to create other use cases specifically for them. For example, an order is created by the use case *Order books*; it is changed by use cases such as *Cancel order* and *Pay order*; it is visualized (retrieved) in many use cases; and, finally, it is assumed that it cannot be removed from the system.

Another question that could be raised with the stakeholders is: What are the reports that the system must produce? Although they are the simplest use cases, as explained later, reports are an important source of information for identifying which information is necessary to fill the stakeholders' needs.

There is a difference between reports and the retrieve query of CRUD: retrieving is simply recovering stored data about an object; a report, however, usually involves a number of objects, sometimes from different classes, and it necessarily includes some kind of filtering or combination of data (sum, product, average, greater value, lower value, etc.). For example, a query for information about a book, given its ISBN, should not be considered as a report use case, because that query is already included in the CRUD use case.

For the running example, some reports that are interesting to certain actors could be identified, such as:

- *Deposit manager*: Report on upcoming orders by period of time, report on deliveries by period of time, report on the total number of books available for sale, report on returns received by period of time, and report on books discarded by period of time.
- *Customer*: Report on orders status, and report on past sales.
- *Sales manager*: Report on book sales by period of time.

The client could demand other reports, and the above are only examples. Those that are used exclusively inside one of the existing use cases must be excluded from the reports list. For example, the deposit manager could be interested in the list of orders that are arriving. If that list is going to be consulted only at the moment an order really arrives and must be registered, then the query must be considered as part of the use case *Receive books*, and must not be included in the diagram. Only reports that are not necessarily part of other use cases must be included in the diagram; otherwise the list could quickly grow far beyond the manageable, and would be redundant and incomplete (because it is the use case expansion with sequence diagrams that will really indicate which steps, including queries, are necessary for each use case). [Figure 3.11](#) shows the use case diagram updated with the reports (stereotyped with `<<report>>`) as indicated above.

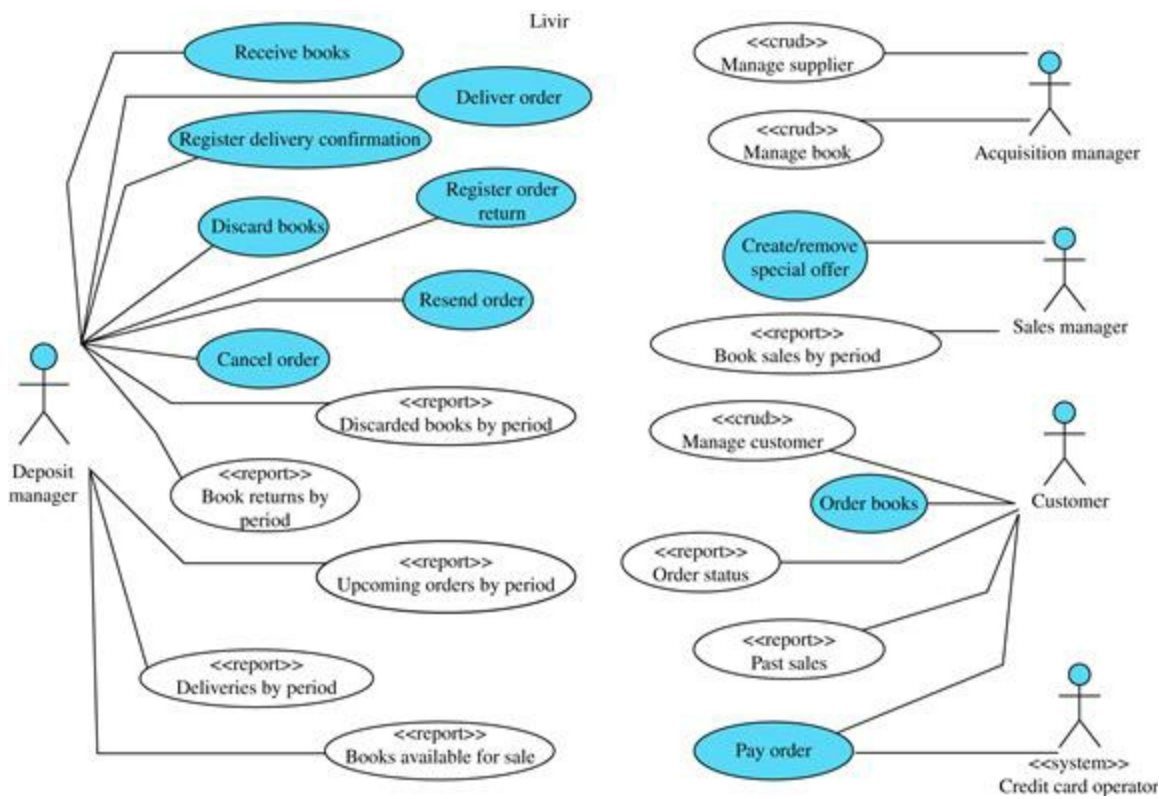


FIGURE 3.11 Use case diagram with reports.⁵

The quantity of `<<report>>` use cases will depend on the structure of the information they present. When parameterization is possible it must be used. It is not necessary, for example, to have a *Sales report by week* and *Sales report by month*. Unless they present different data structures, they are a single report. The reference to a week, month, or any other period of time is just a parameter.

However, it is not advisable to group reports with diverse natures, for example *Sales report by book* and *Deliveries report by period of time*: the inputs and outputs are different. Thus, those two reports must be considered two different use cases.

As seen in [Figure 3.11](#), the number of use cases may become high and the diagram may quickly become hard to organize. This is one of the reasons to avoid including fragments in these diagrams. Each single use case will be detailed by other means. Also, the CRUD and report use cases may not be included in the main use case diagram if the system is at least of moderate size. They must be listed somewhere else (a spreadsheet or a separated use case diagram, for instance), in order not to pollute the main use case diagram. Another solution that may be used sometimes is to represent them with a different color (as done in [Figures 3.10](#) and [3.11](#)), so that they do not hinder the visualization of the key use cases in the diagram.

3.7 The process so far

	Inception	Elaboration
Business Modeling	Build a general view of the system: <ul style="list-style-type: none"> • Build a business use case diagram and determine the automation scope for the project. • Build preliminary activity diagrams for business use cases. • Build preliminary state machine diagrams for key business objects. 	
Requirements	Prepare the system use case diagram (functional requirements): <ul style="list-style-type: none"> • Identify the system actors from the business use case model. • Identify the system use cases from the business use case model, and the activity and state machine diagrams from business modeling. Identify the nonfunctional requirements as use case annotations: <ul style="list-style-type: none"> • Identify the main business rules associated to use cases. • Identify the main quality issues associated to use cases. Identify supplementary requirements.	
Analysis and Design	Prepare the preliminary conceptual model by observing system use cases and the concepts needed by them.	
Implementation		
Test		
Project Management		

3.8 Questions

1. Explain the differences between a business use case and a system use case.
2. What is the utility of a system use case through the software development process?
3. Which business actors and business workers are converted into system actors?
4. What are functional, nonfunctional, and supplementary requirements and what features might they have?
5. Why must a preliminary conceptual model be made during the Inception phase?

¹From now on, system use cases will be referred to simply as *use cases* for simplicity, except when they are compared to business use cases.

²That idea follows Larman's (2004) definition of an *EBP (Elementary Business Process) use case*. The EBP definition comes from the business process engineering field: "a task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state."

³Another option is to use the MoSCoW scale: Must, Should, Could, and Would.

⁴http://www.iso.org/iso/catalogue_detail?csnumber=35733

⁵Use cases stereotyped with <<report>> have their names abbreviated to avoid repeating "Generate report for ..." repeated many times.

Use Case Based Project Planning

This chapter shows how to plan and develop a project using iterations, risk analysis, and effort estimation based on high-level use cases. It explains how the effort to develop the project is estimated based on the perceived complexity of each use case and other features of the system and development team. Finally, the chapter explains how to organize a series of incremental iterations based on a prioritized list of use cases and risks, which is the backbone for the development plan with the Unified Process.

Keywords

Use case point analysis; risk analysis; effort estimation; iteration; project planning

4.1 Introduction to effort estimation and risk analysis in software projects

The motivation for effort estimation in software projects comes from the fact that historically most software development projects:

- Take more time to complete than expected.
- Cost much more than expected.
- Generate a product that does not have the quality expected.
- Generate a product that does not have the scope expected.

Thus, the point is that software development teams and clients must learn to expect the right time, cost, quality, and scope regarding their project and their team. Effort estimation techniques are used to accomplish these goals.

4.1.1 Ad hoc techniques

A number of techniques for effort estimation have been proposed and used in the last several decades. One that is very popular is the *Smith* technique, which may be summarized as “Smith, tell us how much time you need to develop that system!” A variation of the Smith technique is the *constrained Smith* technique, which may be summarized as “Smith, you have three months to develop that system!” Although popular (and known by other names, such as Susan, Peter, Mary, etc.), this approach is not very accurate.

Another ad-hoc technique that is used is the *six months* technique. It consists of estimating “six months” for any software project at the beginning of its development and then adjusting it up or down as the scope and requirements are discovered. Variations of that technique are virtually infinite, including the *One Year* technique and the *Eighteen Months* technique, among others.

These kinds of techniques are known as *nonparametric estimation* because they are not based on measures on the project to be developed. Somerville (2006) identifies some other nonparametric techniques:

- *Expert judgment*. The expert judgment technique proposes that one or more experts on the project domain and software development should meet and use their experience to produce an estimate. The Smith technique is a potentially bad scenario for the expert judgment technique: the technique may be very inaccurate, depending on the expertise of the estimators. However, if the experts really have experience in similar projects it is a feasible, quick, and relatively cheap technique.
- *Estimation by analogy*. That technique is in fact the pragmatic basis for the expert judgment technique. It is assumed that the effort for developing a new system will be similar to the effort for developing other similar systems. The technique is not feasible if similar projects are not available for comparison. It can make good use of one or more experts in order to determine what qualifies as a *similar* system and how the time spent to develop it may be used as a basis to estimate the effort for the current project. Thus, this technique usually may evolve to expert judgment.
- *Parkinson's Law*. That technique is not usually adopted openly, but it is recurrent in software projects and in other areas: *the project costs whatever resources are available*. The advantage is that the project will not cost more than expected, but frequently the scope of the system will not be completely covered.
- *Pricing to win*. The cost of the project is equal to the price that it is believed will win the contract. The disadvantage of this method is that the system the client gets may not be what she expected, because as costs must usually be reduced, this may impact quality and scope. However, when there is no detailed information about the system and the team lacks experience on more advanced estimation techniques this technique may be chosen: the project cost is agreed based on common understandings and the development is constrained by the cost.

The agile community usually adopts an estimation strategy that is based on story points. A *user story* is a scenario of use of a system, and it is somewhat similar to a use case. However, use cases are more structured and formal; they are produced by a team of analysts. User stories, on the other hand, must be written by the users themselves. They must represent a scenario where the users see themselves using the system that is going to be produced. The idea behind this is that the users present the needs that are most important for them first; the details would be remembered later.

One *story point* is considered to be an *ideal working day* (6 to 8 hours focused and dedicated to a project). The estimation effort is then calculated for each user story in terms of story points. The question to be answered by the team is, "How many ideal days x people would take to develop a given user story?" If the answer is x people would take y days, then the number of story points is $x*y$. Story points assignment is subjective and may be considered a nonparametric technique. The team usually evaluates the effort based on their experience on similar past projects. The assignment is made based on effort, complexity, and risk. For example, the following questions could be asked:

- *Complexity*: Does this user story have many possible scenarios?

- *Effort*: Is the change going to be made in many different components?
- *Risk*: Does anyone in the team know the technology necessary to develop this story?

Another way to estimate user stories is to place all user stories on cards on the table. Then, the simplest ones to develop are separated and one story point or less is assigned to them. After that, the simplest of the remaining stories are separated and two story points assigned to them. This procedure is repeated until no more user stories remain on the table. It is suggested that each iteration increase the number of story points by following a series similar to the Fibonacci series such as 0, ½, 1, 2, 3, 5, 8, 13, 20, 40, 100, etc.¹ The reasoning behind this is that for an average human it is hard to estimate how much a horse weighs; however, most people will agree that a horse weighs more than a dog and less than an elephant.

If a user story is determined to take less than one ideal working day, it must be attached to another story, and stories that are above a given limit (for example, 10 story points) must be split into simpler stories.

4.1.2 Parametric techniques

Another class of estimation techniques includes the parametric ones which aim to present an effort estimation based on the estimated size of the system, the technical difficulty to develop it, and the capacity of the team, among other features. The size of the system may be measured in terms of:

- *Lines of code*. Techniques such as COCOMO II – COConstructive COst MOdel II (Boehm, 2000) start estimating how many *source lines of code* (SLOC) the system will have. In fact, as most systems have thousands of lines of code, the usual measure is *kilo source lines of code* (KSLOC).
- *Function points*. Techniques based on function point analysis (FPA) (Albrecht, 1979) and its variations are not concerned about lines of code. They estimate that the effort to develop a system depends on the *apparent functionality* that is going to be implemented. Thus, usually these techniques count the number of functional requirements and evaluate their complexity by asking how many classes or data records they need to use in order to be implemented and how many arguments a user sends or receives from the system in order to perform the function.
- *Use case points*. That is a variation of the function point analysis technique that is based on use cases instead of functional requirements for estimation. This technique is explained in detail in [Section 4.2](#).

Although COCOMO II is the most detailed technique it has a disadvantage compared to function points and use case points: the number of KSLOC must be estimated by specialists before applying the method, and this is a significant source of uncertainty.² Function points and use case points are based on requirements, which may actually be known at the beginning of a project (of course requirements may change during development, but when they change estimation may be immediately updated in a systematic way).

In addition to the size of the system measured in lines of code or apparent

functionality, the parametric techniques also make use of *technical adjustment* factors. It is assumed that implementing a function of the same size may be easier or harder depending on the kind of system that is being developed. For example, an information system like Livir is more straightforward to implement than a robot surgery system or a core system embedded into an airplane. The same amount of source lines of code may take more or less time to develop depending on the technical complexity of the system.

The most complete technique regarding technical adjustment factors is COCOMO I which proposed the use of up to 16 effort multipliers divided into four groups:

- *Product attributes*. These include the reliability required for the software, the relative size of the test database, the inner complexity of the product, development aiming at reusability, and the quantity of documentation expected and needed.
- *Hardware attributes*. These include aspects related to time and space efficiency and the volatility of the development platform.
- *Personnel attributes*. These include the capacity of analysts and programmers, personnel continuity, experience on similar applications, and experience on the platform, programming language, and other software tools.
- *Project attributes*. These include the use of computer-aided software engineering (CASE) tools, distribution of the development team, and the need to accelerate the development schedule.

Use case points and function points have similar effort multipliers but with a different organization and sometimes a different interpretation. In both methods, the effort multipliers are a numeric index that is multiplied by the estimated size of the system to produce effort estimation. However, COCOMO II has a unique feature: a set of *scale factors*. There are five such factors:

- *Precedentedness*. Is the product similar to others that were already developed by the same team?
- *Development flexibility*. Does the product have to be developed straight to the requirements or are the requirements only general goals?
- *Architecture and risk resolution*. Is there good support to resolve risks and architectural issues?
- *Team cohesion*. Have the team worked together before? Are they nicely integrated?
- *Process maturity*. What is the maturity of the organization? CMMI³ (SEI-CMU, 2010) and SPICE⁴ (Emam, Melo & Drouin, 1997) may be used as a measure of maturity.

COCOMO II also proposes a simple questionnaire that helps to estimate the maturity of the organization if other assessments are not available.

The difference here is that those scale factors are not multiplied by the system size: they are used in an *exponential*. Average values (near 1) for these factors means a nominal scale where each line of code produced costs the same regardless of the size of the software. If the scale factors are bad, then their numerical value is greater than 1 and this means that a line of code in a bigger system costs more than a line of code in a smaller system. If the scale factors are good, then the team gains in scale, that is, a line of code in a bigger system costs less than a line of code in a smaller system.

4.1.3 Risk analysis

A *risk* is something that may happen and cause trouble.⁵ Risks are very frequent and numerous in software projects. There are different categories of risk and they have different impacts: there are some that are just a nuisance and others that may cause the termination of the project. Usually the process of dealing with risks involves:

- *Identification*. The team must discover and be aware of the conditions that may trigger problems in a project. Checklists such as the one by Carr et al. (1993) may help the identification process.
- *Analysis*. Risks are analyzed so that some of their attributes are discovered. Usually at least two attributes must be identified: the *probability* of the occurrence of the condition that triggers the risk, and the *impact* that the risk has on the project. The *exposure* or *importance* of a risk is a combination of probability and impact. Usually risks with high probability of occurrence and high impact on the project are the most important.
- *Planning*. Once the exposure of the risks is assessed, plans must be made for medium- and high-exposure risks and the project must include activities to minimize the probability and/or impact of the most important risks. These plans are the *mitigation plans*. Moreover, *recovery* or *contingency plans* should also be prepared for the most important risks and they will be executed if, despite the mitigation activities, the risk still becomes a problem.
- *Control*. The risks of a project are even more unstable than its requirements: they change frequently and unexpectedly. New risks are discovered during development and their importance may change. Thus, the project manager must be aware of that and pay close attention to the status of the risks during the whole project.

Different sources of risks may be identified. Carr et al. (1993) presents the following structure:

- *Product engineering*:
 - *Requirements*: Are they stable, complete, clear, valid, and feasible? Is the project something that has never been made before? Is the project bigger than any other the organization has undertaken before?
 - *Design*: Is it difficult to obtain? Are the interfaces with other systems clear and easy? Are there performance, testability, and hardware constraints? Will software developed outside the company be used?
 - *Code and test*: Is the implementation feasible? Is the time reserved for testing enough?
 - *Integration and test*: Is the integration and test environment adequate?
 - *Engineering aspects*: Will the software be easy to maintain? Are the reliability, safety, and security requirements easy to obtain and verify? Are there special human factors to be taken into account? Is the documentation adequate for the project?
- *Development environment*:
 - *Development process*: Is there a clear and proven development process in use? Are the plans formally controlled? Is the process adequate for this kind of system? Do the members of the team know and use the process? Are there clear mechanisms

for managing changes to the product?

- *Development system*: Is there adequate hardware and other tools for the development of the project? Are the tools easy to use? Does the team know how to use the tools? Are the tools reliable?
- *Management process*: Are the plans adequate? Do they include contingency plans? Are roles and authority inside the project well defined? Do the managers have experience? How does the team communicate with other stakeholders?
- *Management method*: Are there metrics to help management? Are the project personnel trained? Is quality management adequate? Is the configuration of code and other artifacts controlled?
- *Working environment*: Is the team concerned about quality? Is the team cooperative? Is communication among team members good? Is the morale of the team high?
- *External constraints*:
 - *Resources*: Is the schedule reliable and stable? Is the team experienced and capable? Is the budget adequate? Are the facilities adequate?
 - *Contract*: What kind of contract exists? Are there contractual constraints? Does the project depend on external partners?
 - *Communication interfaces*: Are there problems with the client such as weak communication or incomplete knowledge about the domain?
 - *Subcontracted personnel*: Are there problems with partner companies such as weak interfaces, poor communication, or lack of collaboration? Does the project depend on them?
 - *Main contractor*: If you are a subcontracted company, are there any difficult issues regarding the main contractor?
 - *Management*: Is the higher management absent or micromanaging?
 - *Sellers*: Do the product sellers you are using solve issues quickly?
 - *Policies*: Are policies creating trouble for the project?

This list is just a reference; many other unpredictable risks may appear in most projects.

After being identified, risks must be analyzed. Many risk attributes may be identified. Among them, maybe the most important are:

- *Probability*: The chance that a risk really becomes a problem. If there is a historical series of data, risk probability could be measured numerically. However, most risks are assessed using the *t-shirt scale*, where “high” means that it is almost sure that the risk will become a problem, “medium” means that it is expected that the risk will become a problem, and “low” means that the chance for the risk to become a problem is minimal.
- *Impact*: The price to pay (monetary or not) when the risk becomes a problem. Again, numerical measures may be used if sufficient information is available. For most projects, risk impact could be assessed as “high” (may terminate the project), “medium” (may raise the project’s costs significantly), and “low” (may cause some trouble, but losses are easily recoverable).
- *Importance or exposure*: The combination of probability and impact, as expressed in

Table e4.1**How to Calculate the Exposure of a Risk**

		Probability		
		High	Medium	Low
Impact	High	High exposure	High exposure	Medium exposure
	Medium	High exposure	Medium exposure	Low exposure
	Low	Medium exposure	Low exposure	Low exposure

High- and medium-exposure risks should have mitigation plans for reducing the probability and/or impact. In the case of high-exposure risks, it is advisable to execute the plans as soon as possible, while medium-exposure risks may be addressed later. Thus, risk mitigation activities will be included in the project effort at this time. Low-exposure risks may just be monitored in order to detect if their exposure changes during the project.

Among other possible definitions, it may be said that a risk is composed of three parts:

- *Cause*: An uncertain condition that may create trouble.
- *Problem*: A situation that could occur given the uncertainty of the cause.
- *Effect*: An impact over one or more of the project's goals.

Thus, there are three kinds of plans that may be associated to risks:

- *Mitigation plan for reducing risk probability*: It must act on the *cause* of the risk, reducing the probability of its occurrence.
- *Mitigation plan for reducing risk impact*: It must act on the *effects* of the risk, reducing the cost of recoverability if the risk really becomes a problem.
- *Contingency plan or disaster recovery plan*: It states what to do after the risk actually becomes a problem, that is, it acts on the *problem* created by the risk.

Mitigation plans are preventive: they should be executed before the risk becomes a problem. Contingency plans, on the other hand, are performed only if the risk really becomes a problem.

Table e4.2 presents an example with some risks related to the Livir project and a simple statement of mitigation and contingency actions. More detailed plans could be elaborated, though, if necessary.

Table e4.2**Example of Risks and Plans for the Livir Project**

Code	Risk			Risk Attributes			Probability Reduction Plan	Impact Reduction Plan	Contingency Plan
	Cause	Problem	Effect	Pr.	Imp.	Exp.			
K1	Unstable requirements.	Key requirements may change during development.	Time lost developing parts that will not be used.	H	H	H	Take special care during requirements elicitation. Study similar products. Develop prototypes for requirements validation.	Emphasize modular development. Implement an efficient version control system.	Use a change management system to adequately control requirements changes.
K2	The team is inexperienced.	Bad design decisions may be made.	Code hard to maintain and full of defects.	H	M	H	Hire an experienced software architect to coordinate and validate all design decisions.	Use automatic testing extensively to find defects.	Refactor code to improve design.
K3	The team is not familiar with the communication protocol with credit card operators.	Wrong or inefficient communication interface with credit card operators.	Errors in credit card operation.	H	L	M	Develop a throwaway prototype for studying the communication protocol of the credit card operator system.	Implement double-check procedures for credit card operations based on invariants and postconditions.	Double-check security on credit card operations. Interrupt operations with credit cards until the problems are fixed.

Risks must be monitored while the project is being developed. As requirements usually do, risk may change over time. Even their names and properties may change. That is why risks are identified by a code: to keep track of them even when they change.

4.2 Use case point analysis⁶

Once system use cases have been identified, the effort necessary to develop them may be estimated. Furthermore, the use cases must be prioritized so that the most important use cases are developed first. In iterative processes such as the Unified Process, a relatively small set of use cases will be developed at each iteration. Thus, planning a project using this kind of process consists of determining a general plan or *phase plan* and a set of *iteration plans* that determines which use cases or risks will be addressed at each iteration. This is why it is important to determine a project's priorities and estimate the effort necessary to develop them. During Inception, only the phase plan and the plan for the first iteration of Elaboration are built. Each further iteration is planned while the previous one is being performed.

Phase and iteration planning are typical project management activities. Effort estimation is a planning tool that indicates, among other things, how much time and personnel is needed by the project.

The technique known as *use case points* (Karner, 1993) is based on *function points* (Albrecht & Gaffney Jr., 1983), specifically on the *MK II* or *Mark II* counting technique (Symons, 1988), which is a relatively simpler approach than that of the IFPUC (*International Functional Point Users Group*).⁷

Use case point analysis is simpler than function point analysis, but it still suffers from

not having strong reports on its application to a wide number of projects, such as function point analysis and COCOMO II do. Another problem with use case point analysis is that the team comprehension on what constitutes a use case may produce disparate estimations. COCOMO II suffers from a similar problem because it is based on a crude estimation on the number of lines of code to be produced, but it allows estimators to use a *backfire table* ([Center for Software Engineering USC, 2000](#)) to convert function points into lines of code (with different conversion rates depending on the language used). Moreover, function point analysis also depends on the capacity of the estimators and the capacity of the analysts to identify the correct and complete set of requirements. The conclusion is that each technique has a weak point; however, a team that is well tuned, with historical records on estimates for past projects, and that applies the same patterns and conventions when developing software would produce better estimations than an inexperienced team, regardless of the estimation method they use. All methods depend on a good estimation effort history and the quality of the parameters obtained for the current project (lines of code, functional requirements, or use cases).

The technique of use case point analysis was incorporated to the Unified Process because that process is strongly based on use cases. As said before, one of the biggest difficulties to applying it is not the method itself, which is very simple, but the lack of standardization on what is effectively a use case ([Anda, 2001](#)). Due to the absence of an international standard, the team may create and use its own understanding of the adequate granularity for use cases.

Use case point analysis is based on the quantity and complexity of system actors and use cases, which corresponds to the *unadjusted use case points* (UUCP) value. The application of technical and environmental factors leads to the *UCP*, or *adjusted use case points*.

UUCP is calculated from the estimated complexity of the actors (*UAW – unadjusted actor weight*) and use cases (*UUCW – unadjusted use case weight*), as discussed in the next subsections.

4.2.1 UAW – unadjusted actor weight

The method for assigning complexity to actors is quite simple. Each actor is counted once, even if it is associated to many use cases. In the case of actors that are specializations of more generic actors, only the most elementary specializations are counted, that is, actors that do not have subtypes.

Once the actors have been identified, their complexity is defined as follows:

- Human actors that interact with the system through a graphic user interface are considered complex and receive 3 use case points.
- Other systems that interact with the system through a protocol like TCP/IP and human actors that interact with the system only through a command line are considered of medium complexity and receive 2 use case points.
- Other systems that are accessed by application programming interfaces (API) are

considered of low complexity and receive 1 use case point.

The value of *UAW* (*unadjusted actor weight*) is simply the sum of the points assigned to all system actors.

In the example of [Figure 3.11](#) the following actors were considered:

- Deposit manager: 3 points.
- Acquisition manager: 3 points.
- Sales manager: 3 points.
- Customer: 3 points.
- Credit card operator <<system>>: 2 points.

Thus, for that example, $UAW=3+3+3+3+2=14$.

The use of *UAW* for effort estimation, however, is not unanimously accepted. Sometimes, it is suggested that it can be ignored in the equations. Usually, even *UCP* counting tools consider *UAW* an optional measure and using it or not is a decision left to the team.

4.2.2 UUCW – unadjusted use case weight

Use cases are also counted once. Only complete processes should be counted. That means that extensions, variants, and other fragments, in general, should not be counted.

In the original proposal of [Karner \(1993\)](#), the complexity of a use case is defined based on the estimated number of *transactions* (information being sent to or received from the system), including alternate sequences (see [Chapter 5](#) for more details). Then, use cases could be characterized as follows:

- *Simple use cases*: No more than three transactions.
- *Medium use cases*: Four to seven transactions.
- *Complex use cases*: More than seven transactions.

However, if we are working with high-level use cases, it may be hard to know for sure how many transactions each use case really has, and that makes that estimation a little risky.

An alternate way to estimate the complexity of a use case is by the number of classes necessary to implement the use case functions. Thus:

- Simple use cases must be implemented with five classes or less.
- Medium use cases must be implemented with six to ten classes.
- Complex use cases must be implemented with more than ten classes.

The number of classes necessary for implementing a use case will depend also on detailed requirements that usually are not known during Inception, given that the use cases have not been yet expanded. That makes this approach also somewhat risky.

Another way to estimate the complexity of a use case is by the analysis of its risk. Thus:

- *Reports* usually have only one or two transactions and low risk, because they do not change data. They can be considered simple use cases.
- *Pattern use cases* such as CRUD have a number of transactions that are known and limited. Their risk is medium, because although their internal logic is known, obscure business rules may affect them, and they can be considered medium complexity.

- *Nonpattern use cases* have an unknown number of transactions, and high risk, because besides the fact that the business rules may be unknown or misunderstood, even the internal structure of the use case is unknown (its main flow and alternate sequences). Thus, this kind of use case is considered complex.

The *UUCW* (*unadjusted use case weight*) value is given by the sum of the value assigned to each use case using the following criterion:

- Simple use case: 5 use case points.
- Medium use case: 10 use case points.
- Complex use case: 15 use case points.

Applying that technique to the example of [Figure 3.11](#), there are eight use cases stereotyped as reports, which receive $8 \times 5 = 40$ use case points. There are also three CRUI use cases (pattern use cases), which receive $3 \times 10 = 30$ points. Finally, there are 10 use cases without any stereotype with higher risk, which receive $10 \times 15 = 150$ use case points. Thus, the *UUCW* of this example is $40 + 30 + 150 = 220$.

There are also other counting techniques. But most of them are based on the detailed version of a use case. Some of them are presented in [Section 4.2.9](#).

4.2.3 UUCP – unadjusted use case points

The value for *UUCP*, or *unadjusted use case points*, is calculated as the sum of *UAW* and *UUCW*:

$$UUCP = UAW + UUCW$$

In the running example, $UUCP = 14 + 220 = 234$.

As mentioned before, actors' weights may be excluded from the computation of *UUCP*, and that would make *UUCP* immediately equal to *UUCW*, that is, $UUCP = UUCW = 220$ in the example.

4.2.4 TCF – technical complexity factor

The technique of use case point analysis proposes that the *UUCP* should be adjusted by two criteria: *technical factors* (which are associated to the project), and *environmental factors* (which are associated to the team).

Each factor receives a rating from 0 to 5, where 0 means no influence on the project, 3 is nominal influence, and 5 is maximum influence on the project.

Surprisingly, this is the only information usually given in literature about the technical factors of use case point analysis, and that makes the assessment of these factors highly subjective, and even sometimes contradictory.⁸

There are 13 technical factors, each one with a specific weight, as shown in [Table e4.3](#).

Table e4.3

Technical Factors for Adjusting use Case Points

Code Factor Weight

T1

Distributed system

2

T2

Response time/performance objectives

1

T3

End-user efficiency

1

T4

Internal processing complexity

1

T5

Design aiming for code reusability

1

T6

Easy to install

0.5

T7

Easy to operate

0.5

T8

Portability

2

T9

Design aiming for easy maintenance

1

T10

Concurrent/parallel processing

1

T11

Security

1

T12

Access for/to third-party code

1

T13

User training needs

1

Therefore, each of the technical factors has a rate between 0 and 5 that is multiplied by its weight. The sum of those products corresponds to the *TFactor* value, that is, the

impact of the technical factors, which ranges between 0 and 70.

The *TCF* (*technical complexity factor*) may be calculated by adjusting *TFactor* to the range 0.6 to 1.3 with the following formula:

$$TCF = 0.6 + (0.01 * TFactor)$$

For a better reference on the meaning of the ratings 0 to 5, it is suggested that one use the directions below, adapted from the function point analysis technique (Albrecht & Gaffney Jr., 1983).

T1 – Distributed system: Is the system architecture centralized or distributed?

0. The application ignores any aspect related to distributed processing.
1. The application generates data that will be processed by other computers with human intervention (for example, spreadsheets or preformatted files sent by media or email).
2. Application data are prepared and transferred automatically for processing in other computers.
3. Application processing is distributed, and data are transferred in just one direction.
4. Application processing is distributed and data are transferred in both directions.
5. Application processes must be executed in the most appropriate processing core or computer, which is dynamically determined.

T2 – Response time/performance objectives: What is the importance of the application response time to its users?

0. No special performance requirement was defined by the client.
1. Performance requirements were established and revised, but no special action must be taken.
2. Response time and transfer rates are critical during peak hours. No special design for processor core use is necessary. The deadline for most processes is the next day.
3. Response time and transfer rates are critical during commercial hours. No special design for processor core use is necessary. Requirements regarding deadlines for communication with interfaced systems are restrictive.
4. In addition to 3, performance requirements are sufficiently restrictive for requiring performance analysis tasks during design.
5. In addition to 4, performance analysis tools must be used during design, development, and/or implementation, in order to meet the client's performance requirements.

T3 – End-user efficiency: Is the application designed to allow final users just to do their job or is it designed to improve their efficiency?

0. The application does not need any of the items below.
1. The application needs one to three of the items below.
2. The application needs four to five of the items below.
3. The application needs six or more of the items below, but there is no requirement related to user efficiency.
4. The application needs six or more of the items below, and the user efficiency requirements are so strong that the design must include features to minimize typing, maximize defaults, use templates, etc.

5. The application needs six or more of the items below, and the user efficiency requirements are so strong that the design activities must include tools and special processes to demonstrate that the performance goals are obtained.

The following items must be considered for the assessment of the end-user efficiency item:⁹

- Navigational help (for example, dynamically generated menus, adaptive hypermedia, etc.).
- Online help and documentation.
- Automated cursor movement.
- Predefined function keys.
- Batch tasks submitted from online transactions.
- High use of colors and visual highlights in screens.
- Minimizing the number of screens to reach the business goals.
- Bilingual support (counts as four items).
- Multilingual support (counts as six items).

T4 – Internal processing complexity: Does the application need complex algorithms?

0. None of the options below.

1. One of the options below.

2. Two of the options below.

3. Three of the options below.

4. Four of the options below.

5. All five options below.

The following options need to be considered for assessing internal processing complexity:

- Careful control (for example, special audit processing) and/or secure processing specific to the application.
- Extensive logical processing.
- Extensive mathematical processing.
- Lots of exception processing resulting from incomplete transactions that needs to be reprocessed, such as incomplete automated teller machine transactions caused by interruption of communication, missing data values, or failed data change.
- Complex processing to manage multiple inputs and output possibilities, such as multimedia or device independency.

T5 – Design aiming for code reusability: Is the application designed so that its code and artifacts will be highly reusable?

0. There is no concern about producing reusable code.

1. Reusable code is generated for use inside the same project.

2. Less than 10% of the application must consider more than the user needs.

3. 10% or more of the application must consider more than the user needs.

4. The application must be specifically packaged and/or documented for facilitating reuse, and the application must be customizable by the user at the level of source code.

5. The application must be specifically packaged and/or documented for facilitating

reuse, and the application must be customizable by the user with the use of parameters.

T6 – Easy to install: Will the application be designed so that its installation is automatic (for example, in the case of users with low or unknown technical capacity), or is there no special concern about it?

0. The client established no special consideration, and no special setup is necessary for installation.
1. The client established no special consideration, but a special setup is required for installation.
2. The client established requirements for data conversion and installation, and conversion and installation guides must be provided and tested. The impact of conversion in the project is not considered important.
3. The client established requirements for data conversion and installation, and conversion and installation guides must be provided and tested. The impact of conversion on the project is considerable.
4. In addition to 2, tools for automatic conversion and installation must be provided and tested.
5. In addition to 3, tools for automatic conversion and installation must be provided and tested.

T7 – Easy to operate:¹⁰ Are there special requirements regarding the operation of the system?

0. No special considerations about the operation of the system besides normal backup procedures were established by the user.
1. One of the items below applies to the system.
2. Two of the items below apply to the system.
3. Three of the items below apply to the system.
4. Four of the items below apply to the system.
5. The application is designed to operate in nonsupervised manner. “Nonsupervised” means that human intervention is not necessary for keeping the system operational, even if crashes occur, except maybe for the first startup and final turn off. One of the application features is automatic error recovery.

For the evaluation of the easy to operate factor, the following items must be considered:

- Effective processes for initialization, backup, and recovery must be provided, but operator intervention is still necessary.
- Effective processes for initialization, backup, and recovery must be provided, and no operator intervention is necessary (counts as two items).
- The application must minimize the need for data store in offline media (for example, tapes).
- The application must minimize the need for dealing with paper.

T8 – Portability: Is the application or parts of it designed to work on more than one platform?

0. There is no user requirement to consider the need for installing the application on

more than one platform.

1. The design must consider the need for the system to operate in different platforms, but the application must be designed to operate only in *identical* hardware and software environments.
2. The design must consider the need for the system to operate in different platforms, but the application must be designed to operate only in *similar* hardware and software environments.
3. The design must consider the need for the system to operate in different platforms, but the application must be designed to operate in *heterogeneous* hardware and software environments.
4. In addition to 1 or 2, a documentation and maintenance plan must be elaborated and tested to support operation in multiple platforms.
5. In addition to 3, a documentation and maintenance plan must be elaborated and tested to support operation in multiple platforms.

T9 – Design aiming for easy maintenance: Does the client require that the application must be easy to change in future?

0. None of the items below.
1. One of the items below.
2. Two of the items below.
3. Three of the items below.
4. Four of the items below.
5. Five or more of the items below.

To evaluate this factor, the following items are considered:

- A flexible report structure must be provided to deal with simple queries such as logical binary operators applied to just one logical archive (count as one item).
- A flexible report structure must be provided to deal with medium complexity queries such as logical binary operators applied to more than one logical archive (count as two items).
- A flexible report structure must be provided to deal with high complexity queries such as combinations of logical binary operators applied to one or more logical archives (count as three items).
- Business control data are kept in tables managed by the user with interactive online access, but changes must only be effective on the next day (count as one item).
- Business control data are kept in tables managed by the user with interactive online access, and changes are effective immediately (count as two items).

T10 – Concurrent/parallel processing: Must the application be designed in order to deal with problems related to concurrency such as, for example, data and resource sharing?

0. No concurrent access to data is expected.
1. Concurrent access to data is expected sometimes.
2. Concurrent access to data is expected frequently.
3. Concurrent access to data is expected all the time.
4. In addition to 3, the user indicates that a lot of multiple accesses are going to happen,

forcing performance analysis tasks and deadlock resolution during design.

5. In addition to 4, the design requires the use of special tools to control access.

T11 – Security. Are the security needs just nominal or are a special design and additional specifications required?

0. There are no special requirements regarding security.

1. The need for security must be taken into account in design.

2. In addition to 1, the application must be designed so that it can be accessed only by authorized users.

3. In addition to 2, access to the system will be controlled and audited.

4. In addition to 3, a security plan must be elaborated and tested to support access control to the application.

5. In addition to 4, a security plan must be elaborated and tested to support auditory.

T12 – Access for/to third-party code: Is the application going to use code already developed, such as commercial off-the-shelf (COTS) components, frameworks or libraries? High reuse of good quality software reduces the value of this item as it implies less development effort.

0. Highly reliable preexistent code will be used extensively for developing the application.

1. Highly reliable preexistent code will be used in small parts of the application.

2. Preexistent code that eventually needs to be adjusted will be used extensively for developing the application.

3. Preexistent code that eventually needs to be adjusted will be used in small parts of the application.

4. Preexistent code that needs to be fixed or is hard to understand will be used in the application.

5. No preexistent code will be used in the application or questionable quality code will be used in the application.

T13 – User training needs: Will the application be easy to use, or must extensive training be given to future users?

0. There are no specific requirements for user training.

1. Specific user training requirements have been mentioned.

2. There are formal specific user training requirements, and the application must be designed to facilitate training.

3. There are formal specific user training requirements, and the application must be designed to support users with different levels of training.

4. A detailed training plan must be elaborated for the transition phase and executed.

5. In addition to 4, the users are geographically distributed.

Applying the technical factors to the current example, a result similar to the one presented in [Table e4.4](#) could be obtained.

Table e4.4

Assessment for the Technical Factors of Livir

Code	Factor	Weight	Rate	Weight × Rate	Justification
T1	Distributed system	2	0	0	The application ignores any aspects related to distributed processing.
T2	Response time/ performance objectives	1	2	2	Response time and transfer rates are critical during peak hours. No special design for processor core use is necessary. The deadline for most processes is the next day.
T3	End-user efficiency	1	3	3	The application needs the following items, but there are no special requirements related to user efficiency: <ul style="list-style-type: none"> • Navigational help (for example, dynamically generated menus, adaptive hypermedia, etc.) • Online help and documentation • High use of colors and visual highlights in screens • Minimizing the number of screens to reach the business goals • Multilingual support (counts as six items)
T4	Internal processing complexity	1	0	0	None of the options apply.
T5	Design aiming for code reusability	1	0	0	There is no concern about producing reusable code.
T6	Easy to install	0.5	0	0	No special considerations were established by the client and no special setup is necessary for installation.
T7	Easy to operate	0.5	5	2.5	The application is designed to operate in nonsupervised manner. "Nonsupervised" means that human intervention is not necessary for keeping the system operational, even if crashes occur, except maybe for the first startup and final turn off. One of the application features is automatic error recovery.
T8	Portability	2	3	6	The design must consider the need for the system to operate in different platforms, but the client application must be designed to operate in heterogeneous hardware and software environments.
T9	Design aiming for easy maintenance	1	0	0	None of the items apply.

(Continued)

Code	Factor	Weight	Rate	Weight × Rate	Justification
T10	Concurrent/ parallel processing	1	1	1	Concurrent access to data is expected sometimes.
T11	Security	1	3	3	The need for security must be taken into account in design. The application must be designed so that it can be accessed only by authorized users. The access to the system will be controlled and audited.
T12	Access for/to third-party code	1	5	5	No preexistent code will be used in the application.
T13	User training needs	1	0	0	There are no specific requirements for user training.
TFactor				22.5	

The technical factors value, *TCF*, for the example is obtained by:

$$TCF = 0.6 + (0.01 * 22.5) = 0.825$$

As that value is less than 1, the technical factors for this project indicate a development time that is smaller than nominal (82.5% of nominal). However, the environmental factors still have to be determined.

4.2.5 EF – environmental factors

One aspect that distinguishes use case point analysis from function point analysis is that use case point analysis uses an adjustment factor for the characteristics of the development team. Thus, the same project, with the same technical factors, may have a different number of adjusted use case points for different teams.

There are eight environmental factors that assess the working environment. Each one has its own weight and two of them have negative weights. Again, ratings range from 0 to 5, but their meaning is a little different from those of the ratings attributed to the technical factors. While the ratings of the technical factors evaluate the *influence* of those factors on the project, the ratings of the environmental factors evaluate the *quality* of those factors in the working environment. For example, a 0 rating for the “motivation” factor means that the team is not motivated, a 3 rating means that motivation is average, and a 5 rating means that the team is highly motivated.

The environmental factors are defined in [Table e4.5](#).

Table e4.5

Environmental Factors for Adjusting Use Case Points

Code
Factor

Weight

E1

Familiarity with the development process

1.5

E2

Experience in the application

0.5

E3

Object-oriented experience

1

E4

Lead analyst experience

0.5

E5

Motivation

1

E6

Stable requirements

2

E7

Part-time workers

-1

Difficulty with programming language

-1

Thus, the ratings multiplied by the weight of the environmental factors may range from -10 to 32.5. This value is called the *EFactor*.

EFactor is adjusted to the interval 0.425 to 1.7, generating the *EF* (environmental factor) by the following formula:

$$EF = 1.4 - (0.03 * EFactor)$$

Ribu (2001) presents a reference for assigning ratings to the environmental factors, which is described below with some adaptation.

E1 – Familiarity with the development process This factor assesses the team's experience with the development process they are using. Originally the factor mentioned the Unified Process, and some authors even mention UML here. But this has been adjusted to reflect the fact that other processes and modeling languages could be used instead.

0. The team does not have experience with the development process or does not use any process.
1. The team has theoretical knowledge about the development process, but no experience.
2. A few members of the team have already used the process in one project.
3. A few members of the team have used the process in more than one project.
4. Up to half the team has used the process in many projects.
5. More than half the team has used the process in many projects.

E2 – Experience in the application This factor assesses the familiarity of the team with the application area or domain. For example, if the application is about e-commerce, have the team already worked with systems in the same area?

0. No team member has any experience in projects in the same area.
1. Some team members have 6 to 12 months of experience in projects in the same area.
2. Some team members have 12 to 18 months of experience in projects in the same area.
3. Most team members have 18 to 24 months of experience in projects in the same area.
4. Most team members have greater than two years of experience in the same area.
5. All team members have greater than two years of experience in the same area.

E3 – Object-oriented experience This factor must not be confused with either *familiarity with the development process* (E1) or *experience in the application* (E2): this is about the experience of the team in doing object-oriented analysis, modeling, design, and programming, independently of the application area and the development process used.

0. The team has no experience in object-oriented techniques.

1. All team members have some experience in object-oriented techniques (up to one year).
2. Most team members have 12 to 18 months of experience in object-oriented techniques.
3. All team members have 12 to 18 months of experience, or most team members have 18 to 24 months of experience in object-oriented techniques.
4. Most team members have more than two years of experience in object-oriented techniques.
5. All team members have more than two years of experience in object-oriented techniques.

E4 – Lead analyst experience This factor measures the experience of the lead analyst with requirement analysis and object-oriented modeling.

0. The lead analyst has no experience.
1. The lead analyst has previous experience in a single similar project.
2. The lead analyst has about one year of experience in more than one similar project.
3. The lead analyst has about two years of experience in similar projects.
4. The lead analyst has more than two years of experience in similar projects.
5. The lead analyst has more than three years of experience in a variety of projects.

E5 – Motivation: This factor describes the team's motivation.¹¹

0. The team has no motivation at all. Without constant supervision the team becomes unproductive. The team only does what is strictly asked.
1. The team has very little motivation. Constant supervision is necessary to keep productivity at acceptable levels.
2. The team has little motivation. Management interventions are necessary from time to time to maintain productivity.
3. The team has some motivation. Usually the team has initiative, but management intervention is still necessary sporadically to keep productivity.
4. The team is well motivated. The team is self-managed usually, but the existence of supervision is still necessary because productivity can be lost without it.
5. The team is highly motivated. Even without supervision everyone knows what has to be done, and pace is maintained indefinitely.

E6 – Stable requirements This factor evaluates if the team has been able to keep requirements stable in past projects, minimizing their change during the project. This factor may vary from project to project, because there are domains where requirements change often independently of the capacity of the analysts. This factor must assess only software changes that were caused by the team's inability to discover the right requirements.

0. There is no historic data about requirement stability or, in the past, poor analysis caused big changes in requirements after the project started.
1. Requirements were predominantly unstable in the past. Clients asked for many changes caused mainly by incomplete or incorrect requirements.
2. Requirements were unstable in the past. Clients asked for some changes caused by incomplete or incorrect requirements.
3. Requirements were relatively stable in the past. Clients asked for changes in secondary

functionalities with some regularity. Changes to main functionalities were asked for seldom.

4. Requirements were mostly stable in the past. Users asked for little changes, especially cosmetic ones. Changes in main or secondary functionalities were unusual.
5. Requirements were totally stable in the past. Little changes, if any, had no impact on projects.

E7 – Part-time workers This is a negative factor, that is, contrary to the first six environmental factors, higher values here represent more development time, not less. A team with many members involved in other projects or activities is less productive than a dedicated team.

0. No team member is part time on the project.
1. Up to 10% of the team is part time.
2. Up to 20% of the team is part time.
3. Up to 40% of the team is part time.
4. Up to 60% of the team is part time.
5. More than 60% of the team is part time.

E8 – Difficulty with programming language: This is another negative factor, meaning that high values here are bad for the development time.

0. All team members are very experienced programmers.
1. Most team members have at least two years of experience in programming.
2. All team members have at least 18 months of experience in programming.
3. Most team members have at least 18 months of experience in programming.
4. A few team members have some experience in programming (not more than one year).
5. All team members are inexperienced programmers.

In order to continue with the Livir example, let us assume that the company is formed by a group of five professionals who recently obtained Computer Science degrees and studied object-oriented techniques, but do not have significant professional experience. Three of them are fully dedicated to the project, while the other two have other activities. Livir will be the first project they are going to perform professionally together. With this fictional company, the evaluation of the environmental factors could be something like [Table e4.6](#).

Table e4.6

Environmental Factors Evaluation for a Fictional Team Developing the Livir Project

Code	Factor	Weight	Rate	Weight × Rate	Justification
E1	Familiarity with development process	1.5	1	1.5	The team has theoretical knowledge about the development process but no commercial practice.
E2	Experience in the application	0.5	0	0	No team member has previous experience with similar projects.
E3	Object-oriented experience	1	3	3	All team members have about 12 to 18 months of (academic) experience with object-oriented techniques.
E4	Lead analyst experience	0.5	0	0	The lead analyst has no experience.
E5	Motivation	1	5	5	The team is highly motivated. Even without supervision, each member knows his/her job and maintains the pace indefinitely.
E6	Stable requirements	2	0	0	There is no trustable history for the team.
E7	Part-time workers	-1	3	-3	40% of the team is part time.
E8	Difficulty with programming language	-1	5	-5	Although they have academic experience, all programmers are novices.
<i>EFactor</i>				1.5	

Applying the adjustment formula to the company's environmental factors the following is obtained:

$$EF = 1.4 - (0.03 * 1.5) = 1.355$$

This indicates that the environmental factors are bad for this team of beginners, and that their effort will be around 35% greater than nominal.

4.2.6 UCP – adjusted use case points

Once actor and use case weights as well as technical and environmental factors are calculated, the adjusted use case points may be obtained by simple multiplication:

$$UCP = UUCP * TCF * EF$$

For the running example we obtain:

$$UCP = 234 * 0.825 * 1.355 = 261.583$$

4.2.7 Effort

The key goal of all estimation techniques is to predict the *effort* necessary to develop a project. In the first place, it is necessary to define the extension of the effort that is being estimated. The COCOMO II method, for example, when applied with the Unif

Process, estimates the effort to be spent during Elaboration and Construction only.

Besides this, does the effort include administrative activities, sales, etc.? Or does it include only the effort spent on software engineering activities? Literature on use case points usually is not as detailed as COCOMO II and function point analysis regarding the kind of effort that is being estimated. As use case point analysis is based on function point analysis Mk II (UKSMA Metrics Practices Committee, 1998), we can assume that by default the same definitions apply, such as “The project effort should include staff who is clearly allocated to the project and who work according to the project requirements. The time of a user being interviewed, for instance, would not be included, whereas the time of a user allocated for a significant amount of time to the project team would be included.” Similarly, the following definition can also be applied: “The work effort should be measured in units of a ‘productive’ or ‘net’ work-hours, which is defined as: ‘One hour of work by one person, including normal personal breaks, but excluding major breaks, such as for lunch’.” Mk II also states that a project starts when “it has been agreed and approved that work should be undertaken to deliver an IS¹² solution [...] one or more staff have been allocated to the project, and start working on it” (this happens during the Inception phase), and that a project ends when “the application is made ‘live’ by moving it into the production environment for the first time, and the users make use of it” (this happens during the Transition phase).

The effort that is calculated usually must include all risks and unexpected situations. Estimation is not a precise science though. In some situations things may get out of control and adjustments must be made on the fly to keep the project on the rail. We can say that the final schedule would define the *maximum time tolerable* for delivery of a product with the *minimum scope acceptable*. That is, if we are lucky and risks cause little trouble, we can deliver the product earlier or deliver a better product than expected.

Once the number of use case points is calculated, they have to be transformed into an effort measure by using a *team productivity index (TPI)*. If the TPI is measured as the time one team member spent on a single use case point¹³ then the effort may be given by:

$$E = UCP * TPI$$

As *UCP* is a scale-independent value, it must be decided in which scale the effort will be measured. The use case point literature usually refers to *staff hours* per use case point. However, other methods such as COCOMO II (Boehm, 2000) prefer *staff months*.

Because most formulas that are usually applied to productivity in the software industry are defined in terms of *staff months*, and because the unit chosen may affect results, this book adopts *staff months* as the productivity unit, but refers to *staff hours* when necessary.

Considering that a month corresponds to 158 working hours on average (holidays and weekends disregarded), then x staff hours per use case point correspond to $x/158$ staff months per use case point.

Originally, Karner (1993) estimated that 20 staff hours (approximately 0.127 staff months) per use case point¹⁴ would be a good guess when a trustable history is lacking.

Later, [Ribu \(2001\)](#) determined that that value could range from 15 to 30 staff hours (0.095 to 0.190 staff months) per use case point.

The *TPI* value can also be estimated by taking past projects, looking for the total staff time spent and dividing it by the number of use case points. Thus, for example, if the total effort spent on a given project was 9 staff months and the total project UCP was 60, then the team productivity index in that case is 9/60, that is, 0.15 staff months per use case point.

Other work ([Schneider & Winters, 1998](#)) proposes that the number of environmental factors E1 to E6 rated below 3 must be added to the number of environmental factors E7 and E8 rated above 3, and:

- If the result is 2 or less, assume 20 staff hours (0.127 staff months) per use case point.
- If the result is 3 or 4, assume 28 staff hours (0.177 staff months) per use case point.
- If the result is greater than 4, environmental factors present too high a risk for the project, and must be improved before any development commitment is assumed, or else assume 36 staff hours (0.228 staff months) per use case point.

In the current example there are five negative environmental factors: E1, E2, E4, E6 and E8. The recommendation, in this case, would be trying to improve the quality of the environment before developing any project. However, if the team decides to take the risk, the project should assume the worst estimate of 0.228 staff months per use case point. Thus, the total effort for the Livir project is $E=261.583*0.228=59.64$ staff months.

4.2.8 Calendar time and average team size

The effort value obtained for the example in the last section (59.64 staff months) refers to the total effort for developing the project from its start until the delivery of the product. However, more than one staff member will be working on the project and thus calendar time (linear time with weekends and holydays disregarded) will probably be much smaller.

There is a simple rule of thumb formula that allows one to approximate the ideal calendar time for a project from its total effort ([McConnel, 1996](#)). The formula is:

$$T = 2.5 * \sqrt[3]{E}$$

In this formula, *T* is the ideal calendar time for developing the project.

This formula only works if *E* is expressed in *staff months*. The use of other units such as staff weeks, or staff hours, would cause distortions in the result due to the use of the cubic radix.

Other formulae based on function points ([Jones, 2007](#)) or lines of code ([Boehm, 2000](#)) are available. But, in order to be applied with use case points, some conversions are needed. If a more accurate approximation is desired maybe this is worth the effort because Boehm's and Jones's formulae use information about the project complexity and team capacity as parameters to produce the calendar time estimate.

Applying the rule of thumb formula above to the current example produces:

$$T = 2.5 * \sqrt[3]{59.64} \cong 9.768 \text{ months}$$

The average size of the team (P) is calculated by:

$$P = E/T$$

For the running example, we obtain:

$$P = 59.64/9.768 = 6.106$$

Thus, the average size of the team, in that case, must be approximately equal to six. In summary, the project would be ideally developed by six people in about 10 months. For this example, the development company could choose to hire a new professional (especially a more experienced professional that could help to improve the environmental factor ratings); trying to develop the project with five people could take a time longer than the ideal. For this example, it is possible to calculate the time a team of five would take for developing the project as $59.64/5=11.928$. In that case, the project would be completed by five people in about 12 months. We said “it is possible” because the number of people considered is smaller than the recommended size for the team. However, it would be unrealistic to apply this division to calculate the calendar time for a team larger than the ideal. For instance, it would not be realistic to assume that 20 people could complete the project in $59.64/20=2.982$ months. Any team size higher than the ideal means that the total effort (E) must be adjusted to a higher value. The COCOMO II effort multiplier *SCED* (*Required Development Schedule*) indicates that running a project on a time that is 85% of the ideal time implies adding 14% to the total effort, and running a project on a time that is 75% of the ideal time implies adding 43% to the total effort (Boehm, 2000).

It may appear that those times are too high for the project presented in the example. However, it must be considered that the environmental factors are very bad, because the team has very little experience. They will make errors and possibly will have to redo a lot of work. Most important, the lack of experience may lead the team to misunderstand requirements and have to fix them later, increasing development time significantly.

If instead of those novices we had a *dream team* with the best rates possible for environmental factors, then EF would be 0.425 (the best value possible). In that case, $UCP=234*0.825*0.425=82.046$. Using the recommendation of Schneider and Winters (1998), it can be assumed that the dream team could work at a rate of 20 staff hours or 0.127 staff months per use case point. Thus, $E=82.046*0.127=10.42$. Applying the formula for calendar time and average team size the result is $T=5.46$ and $P=1.908$. That is, a dream team of two people could complete the project in about five and a half months. That seems a very reasonable estimate considering the size of the project and the capacity of the team.

However, it is important to mention that those formulas are somewhat “magical” because there are a great range of projects and development teams. Before using them

seriously in a company it is always useful to test and adjust them to the local reality.

4.2.9 Counting methods for detailed use cases

The use case point analysis technique presented in previous subsections considers only a guess on the complexity of the use cases because it deals with high-level use cases, which are represented only by their name and, eventually, one or two explaining sentences.

There are techniques for counting use case points with expanded use cases (explained in [Chapter 5](#)), but their utility is more limited because, with the Unified Process, use cases are usually only detailed during the iterations. During Inception, the team usually only have high-level use cases.

Even so, those techniques may be useful if the use cases are already detailed, or if effort estimation must be refined when an iteration starts. This is why those techniques are summarized in this section.

[Robiolo and Orosco \(2008\)](#) suggest that the text of the detailed use case must be analyzed and the entities (classes) and transactions (steps) counted. The size of an application in unadjusted use case points would be the sum of those values.

[Braz and Vergilio \(2006\)](#) propose a technique named FUSP (Fuzzy Use Case Six Points), where counting is based on the complexity of:

- Actors that participate in the use case.
- Use case pre- and postconditions.
- Number of entities necessary for the use case.
- Number of transactions of the use case.
- Complexity of the alternate flows of the use case.

Thus, the measure is quite detailed, but a lot of work is necessary to apply it and the use cases must be expanded for it to work.

[Mohagheghi, Anda, and Conradi \(2005\)](#) propose a technique (*Adapted Use Case Points*) that assumes that every actor has medium complexity, and every use case is complex in the beginning. Later, use cases must be split into smaller use cases and reclassified as simple or medium as the refining process goes on. Extended use cases and alternate flows are counted as well.

[Kamal and Ahmed \(2011\)](#) present an extensive comparison among the aforementioned and other counting methods.

The *Full Use Case Size* method ([Ibarra & Vilain, 2010](#)) was proposed with the objective of measuring the size of use cases. The method measures functional and nonfunctional aspects and, to do that, it needs high-level use cases and a list of nonfunctional requirements annotated for each use case. Measurement is based on three components:

- The kind of use case.
- The number of system operations (transactions) and business rules.
- The number of interface requirements.

Details about these techniques may be found in the original works cited above.

4.3 Planning an iterative project

The goal of project planning is to build a *plan* for the project as a whole. Among other things, it is important that the person or group responsible for planning use the best tools available to assess the amount of effort that should be expended in the project. That estimate will lead to the project schedule and budget.

There are several aspects to consider at the start. The scope declaration has already defined the goals of the project and acceptance criteria. Also, the necessary business modeling has already been completed, and the high-level requirements were incorporated to the system use cases.

With the technique of use case point analysis, the total effort for developing the project may be estimated, as well as the ideal calendar time and the average size of the team; these are global measures for the project.

Now it is necessary to define the duration and number of iterations.

4.3.1 Estimating the duration of iterations

An iteration begins with planning and finishes with a new version of the system being released internally or delivered to the client. The duration of an iteration in UP and most agile methods usually ranges from one to eight weeks,¹⁵ and it depends, among other factors, on the complexity of the project and the size of the team.

Small teams with up to five people¹⁶ may do the planning together on a Monday morning, perform the work during the week, and generate a release on Friday.

Groups with more than 20 people might need more time to distribute and synchronize activities. Consider that the amount of work is naturally higher. Moreover, the generation of a release takes more time, because there will be a higher amount of software parts to be integrated, verified, and fixed. Thus, in that case, iterations with three to four weeks would be allowed.

Groups with more than 40 people need to work on a more structured and formal environment, with more intermediary documentation, so that the information flow will naturally be slower. Thus, an iteration with six to eight weeks would be advisable.

Other factors that may affect the duration of the iterations are the following:

- Automatic code generation and integrated life cycle software tools allow for shorter iterations.
- A team that is very competent in UP and analysis, design, and coding techniques allows for shorter iterations.
- If quality is a critical concern and consequently tests and revisions are intense, then iterations must be longer, unless extensive automation and effective quality coding patterns are used.

4.3.2 Number of iterations

The number of iterations of a project depends on the calendar time divided by the duration of each iteration. For example, for the Livir project, with the novice team, the project would take about 10 months with a team of six people. Thus, an iteration of two

weeks would be used, with a total of 20 iterations.

The number of iterations that would be considered for Elaboration and Construction depends fundamentally on the need to address architectural problems. This can be roughly estimated by the proportional quantity of complex use cases that should be studied and developed. As the milestone of the Elaboration phase is a stable architecture, it can be assumed that Elaboration will not terminate while there are complex high priority use cases that need to be analyzed. [Kruchten \(2003\)](#) estimates that the Elaboration would take about 30% of the linear time of a typical project and Construction would take about 50% of the time of a typical project, while Inception and Transition would take about 10% each.

Another question that comes to mind at this point is whether the effort estimation includes the Inception phase or not (because Inception is practically finished when the estimate is made). The COCOMO II method, for example, when applied to the Unified Process, estimates only the effort to be spent in Elaboration and Construction, leaving Inception and Transition to be calculated as a percentage of the total effort. Usually, it is believed that use case point analysis estimates the total effort of a project, from its beginning to the delivery of the product. However, if the team uses another interpretation, the important thing is to keep it coherent from one project to another. Any discrepancy in estimation measures would be absorbed by the team productivity index.

4.3.3 Effort per use case point

In order to find the effort to develop each kind of use case (simple, medium, or complex), the following procedure may be used:

1. Take the total effort for the project (E), and divide it by the total unadjusted use case weight ($UUCW$) – ignore the actors' weight.
2. The resulting value is the effort estimated per unadjusted use case point for the specific project and team ($E_{UCP}=E/UUCW$).
3. Finally, find the effort for each kind of use case by multiplying E_{UCP} by 5 in the case of simple use cases, by 10 for medium, and by 15 for complex ones.

For the running example we have:

$$E_{UCP} = \frac{59,64}{220} = 0.271 \text{ staff months per use case point}$$

Thus:

- Simple use cases demand $5 \cdot 0.271 = 1.355$ staff months.
- Medium use cases demand $10 \cdot 0.271 = 2.71$ staff months.
- Complex use cases demand $15 \cdot 0.271 = 4.065$ staff months.

Notice that the dream team would produce values much lower than those. In the case of the dream team:

$$E_{UCP} = \frac{10,42}{220} = 0.047 \text{ staff months per use case point}$$

Thus:

- Simple use cases demand $5 \times 0.047 = 0.235$ staff months.
- Medium use cases demand $10 \times 0.047 = 0.47$ staff months.
- Complex use cases demand $15 \times 0.047 = 0.705$ staff months.

4.3.4 Team load capacity

Other information that can be calculated is the *team load capacity (TLC)* for each iteration. This is calculated as the number of team members multiplied by the iteration length (expressed in months):

$$TLC = P * \text{iteration length (in months)}$$

In the running example, considering the novice team (six people) and two week iterations (about 0.5 months), $TLC = 6 \times 0.5 = 3$. That means that each iteration for that project has a team load of three staff months.

With a two-week iteration, the team would not be able to finish a complex use case (it demands 4.065 staff months, as seen above). Two options may be considered:

- Extend the duration of the iterations to three weeks (about 0.75 months).
- Slice use cases so that they may be partially implemented in two-week iterations.

Although agile developers would prefer shorter iterations, if we choose to extend the iterations, the calendar time for the project (10 months) divided by the iteration duration (0.75) results in 13.3 iterations, which can be adjusted to 14, unless the project is to be hurried. This way, the team load per iteration would be $6 \times 0.75 = 4.5$ and one iteration would be sufficient to deal with a complete complex use case.

4.3.5 Defining use case priority

One of the main techniques for reducing risk in a project is dealing first with the most complex use cases, especially if they are those that represent the most critical business processes, because with them the team may learn more about the system than with other use cases. Pattern use cases such as CRUD (medium complexity) and reports (smallest complexity) can be addressed later during Construction.

Planning indicates when each use case is going to be analyzed, designed, and implemented. The Unified Process, like most agile techniques, does not expect that the general project plan defines when each use case is to be implemented. However, they are placed in a dynamic priority list, and although it may be updated on the fly, it is possible to know when each use case is supposed to be completed. Planning an iteration begins by choosing which use cases, risks, or change requirements would be addressed on the next iteration. When planning an iteration, the following aspects must be considered when choosing which use cases to address:

- *Use case complexity*: Use cases of higher risk and complexity should be addressed first in most cases, unless special conditions (for example, a relatively simple use case with high technological risk) justify choosing different use cases.
- *Dependencies among use cases*: Use cases that have a strong dependence to the ones that were already accommodated in the iteration should be addressed together if possible, because it avoids fragmenting the work on the classes.
- *Team load capacity*: The development work must be assigned to team members based on their *TLC*.

One suggestion for prioritizing use cases for the Livir example presented in [Figure 3.11](#) is given in [Table e4.7](#).

Table e4.7

Suggestion for Use Case Priorities for the Livir Example

1. Order books
12. Manage books <<crud>>
2. Pay order
13. Manage publishers <<crud>>
3. Deliver order
14. Order status <<report>>
4. Register delivery confirmation
15. Past sales <<report>>
5. Receive books
16. Book sales by period <<report>>
6. Resend order
17. Books available for sale <<report>>
7. Register order return
18. Book returns by period <<report>>

8. Discard books

19. Upcoming orders by period <<report>>

9. Cancel order

20. Deliveries by period <<report>>

10. Create/remove special offer

21. Discarded books by period <<report>>

11. Manage customers <<crud>>

The order of this list is most critical for the complex use cases, because a wrong choice in terms of their priorities could lead to otherwise unnecessary refactoring in subsequent iterations. We considered that the most important use cases for this business are those related to the process of selling books, because that is what generates profit for the company. However, the order of use cases with lower risk such as *cruds* and reports is not so critical.

4.3.6 Planning phase and iterations

With the information obtained in previous sections, it is possible to elaborate the preliminary release plan utilizing the use case priorities, effort, and team load capacity.

The goals of an iteration may be of three types:

- Implementing totally or partially one or more *use cases*.
- Mitigating one or more *risks*, generating or performing a probability reduction, impact reduction, or disaster recovery plan.
- Implementing totally or partially one or more *changes* requested. As the system architecture evolves during iterations, changes may be asked for due to nonconformities to requirements or design errors. Incorporating these changes into the software may be one of the goals of an interaction. This kind of goal usually only arises after the first iteration.

For each element (use case, risk, or change requested) there must be effort estimation. The elements will be selected for inclusion in one iteration or another depending on their priority (priorities may change as the project progresses – therefore, planning may change too). Higher priority must be given to the more complex and risky elements, which are the ones with greater potential for learning about the system. The suggestion is to pick first:

- Use cases that represent the most critical business processes, for example, those

which help the company achieve its organizational goals, such as obtaining profit.

- High-importance (or high-exposure) risks, that is, those with high impact and a high probability of becoming problems.
- Urgent change requests, for example those that demand architecture refactoring.

When considering the elements of highest priority, other elements with lower priorities, but closely related to the higher priority ones, may be added to the same iteration for convenience. The important thing is that the total effort assigned to the iteration does not surpass the staff month value that corresponds to the team load capacity.

Risk mitigation and some change requests may not be suitable for effort estimation because their size cannot be measured in terms of use case points (or any other metric). For example, finding and fixing a bug may take seconds or weeks. In these cases, usually a fixed amount of time is assigned, such as, for example, investing one team member during one week to work on the mitigation of a given risk. At the end of the time box¹⁷ previously defined the result is evaluated and it is decided if the activity must continue or not.

Let's consider that the project has 10 months (40 weeks) and the iteration duration is three weeks. Then, Inception and Transition would take about four weeks (10% of the total time), Elaboration would take about 12 weeks (30% of the total time) and four three-week iterations, and Construction would take about 20 weeks (50% of the total time) and close to seven iterations (one of them could be a week shorter, or one week from the Transition phase could be borrowed for the last Construction iteration – the choser alternative presented below).

Kroll (2004) presents a suggestion for a generic phase plan that is adapted for the Livir example in Table e4.8. At the moment change requests were not included because they appear only when the project is running, not at the beginning. Thus, the plan in this case was made only based on use case release and risk mitigation.

Table e4.8

Generic Phase Planning for the Livir Example

Phase	Iteration	Primary Objective (Risks and Use Cases Addressed)	Schedule
Inception	I1	Define vision Determine project scope Define candidate architecture Create business case Create software development plan	Weeks 1 to 4
Elaboration	E1	Install and test architecture components Validate requirements details Implement priority use cases. Test proposed architecture	Weeks 5 to 7
	E2-E4	Mitigate architectural risks. Complete architecture installation and test Implement additional use cases Load test architectural elements	Weeks 8 to 16
Construction	C1-C6	Describe and implement additional use cases Integrate product and validate	Weeks 17 to 34
	C7	Describe and implement additional use cases Integrate product and validate Plan beta test Plan user manual	Weeks 34 to 37
Transition	T1	Deliver beta to client Analyze feedback from beta users Apply corrections Finalize user manual and other artifacts Deliver to client	Weeks 38 to 40

After phase planning concludes (usually at the end of Inception), the plan for the first Elaboration iteration *E1* can be detailed, with use cases being chosen and activities detailed and assigned to developers. Only when that iteration is running should the planning for the second iteration begin.

The purpose of an iteration plan includes the following (Kroll 2004):

1. Define the scope of the iteration by defining its goals in terms of artifacts to be produced such as:
 - Detailing and implementing specific use cases.
 - Mitigation of known risks.
 - Performing requested changes.
2. Create a detailed plan on how the iteration must unfold.
3. Create, identify, and manage tasks dependencies.
4. Assign tasks to people.

For example, consider that you are planning the first Elaboration iteration *E1* for the Livir project. Consider the general goals presented in Table e4.8, and the current state of the system. Some iteration goals could be defined such as:

- Expand and implement use case 01: *Order books*.
- Test and refine the preliminary architecture.
- Perform mitigation plans for risk K1: *Unstable requirements*.

Then, planning may continue by defining the iteration deliverables and the tasks

necessary to produce them, and assigning them to developers. Agile teams might prefer to produce this plan in a team meeting, while teams following RUP may prefer to instantiate some of the RUP workflows in order to determine which tasks should be performed in order to achieve the iteration goals. Explaining this is out of the scope of this book, but readers interested in learning more may start by consulting [West \(2002\)](#) or [Crain \(2004\)](#).

4.4 The process so far

	Inception	Elaboration
Business Modeling	Build a general view of the system: <ul style="list-style-type: none"> • Build a business use case diagram and determine the automation scope for the project. • Build preliminary activity diagrams for business use cases. • Build preliminary state machine diagrams for key business objects. 	
Requirements	Prepare the system use case diagram (functional requirements): <ul style="list-style-type: none"> • Identify the system actors from the business use case model. • Identify the system use cases from the business use case model, and activity and state machine diagrams from business modeling. Identify nonfunctional requirements as use case annotations: <ul style="list-style-type: none"> • Identify the main business rules associated to use cases. • Identify the main quality issues associated to use cases. Identify supplementary requirements.	
Analysis and Design	Prepare the preliminary conceptual model by observing system use cases and the concepts needed by them.	
Implementation		
Test		
Project Management	Estimate the total effort, ideal calendar time, and average team size for the project. Estimate the duration and quantity of iterations for each phase. Prepare the phase plan and iteration plan for the first iteration.	

4.5 Questions

1. Explain the differences among COCOMO II, function point analysis, and use case point analysis. What are their relative advantages and disadvantages?
2. In use case point analysis what is the difference between technical and environmental factors? Why does use case point analysis consider them as separate groups?
3. Let E be the total effort for developing a project, and T the ideal calendar time recommended for the same project. What happens with E when the project must be developed in a time shorter than T ? Does the same occur when the project may be developed in a time longer than T ?
4. How do you determine the ideal duration of an iteration for a given project?
5. How do you establish a list of priorities for use cases?

¹In the actual Fibonacci series, each number is the sum of the previous two. Thus, the original series is 1, 1, 2, 3, 5, 8, 13,

21, 34, 55, 89, etc. However, for estimation purposes an approximate series is used because as numbers grow the estimation is less accurate.

²To minimize this uncertainty, *backfire tables* that convert function points into KSLOC may be used (Boehm, 2000).

³Capability Maturity Model Integration.

⁴Software Process Improvement and Capability dEtermination.

⁵Some definitions also consider *positive* risks as situations that may happen and help the project.

⁶Readers not interested in knowing the details on how to estimate effort and develop a project plan may want to skip from here directly to [Chapter 5](#) where object-oriented analysis and design techniques continue to be explained.

⁷<http://www.ifpug.org/>

⁸For example, regarding the T12 factor, there are authors that say that more third-party access means less effort, because there is code reuse, while other authors say that more third-party access means more effort, because it is necessary to understand code written by others. In this book, a synthesis between those two views is built: it is considered good to have access to well-written code, not so good to have access to code that needs revision, and worse to not have access to any third-party code. Reusing code of bad quality is considered completely undesirable (it is usually easier to develop it from scratch).

⁹Some items such as “menus” and “scrolling” that were originally considered in the function point analysis were removed from the list because they are considered too trivial for current applications.

¹⁰Occasionally this item is referred to as “usability”, but the original concept of usability was already considered in factor T3, end-user efficiency. We prefer to interpret this factor as “easy to operate” for its compatibility with the technical factors of function point analysis, and also to avoid confusion and redundancy with the end-user efficiency factor.

¹¹Surely that is one of the hardest environmental factors to assess, because the real motivation of the team may be masked. Ribu’s (2001) suggestion seemed to help very little in rating this factor, and we took the liberty of reinterpreting the reference.

¹²Information System.

¹³Optionally, the *TPI* could be expressed as the number of use case points a team member can produce in a given unit of time. In that case, the formula should be $E=UCP / TPI$.

¹⁴Remember that a complex use case has 15 use case points and thus requires about 300 staff hours to be fully developed according to Karner (1993).

¹⁵Agile methods prefer the shortest duration possible. Some methods such as XP are more radical by forbidding iterations with more than three weeks, but other methods are more relaxed on that.

¹⁶Or larger teams that are very tuned and have excellent tool support.

¹⁷A *time box* is a fixed period of time allocated for a task.

CHAPTER 5

Expanded Use Cases

This chapter explains how to detail use cases as text and how to evolve them to system sequence diagrams in order to discover which high-level operations should be implemented by the system. It concentrates on explaining what is really necessary for a use case description during Elaboration, when its functionality must be completely understood. Technical and interface details are still avoided. Only information flows, including variant and exception flows, are included in an essential use case description; this chapter explains how to discover them systematically. It also explains how to transform a textual and informal description of a use case into a sequence diagram, which is a higher ceremony artifact.

Keywords

Detailed use case; expanded use case; full-dressed use case; exception in use case; use case alternate flow; essential use case; CRUD use case; system sequence diagram

Key Topics in this Chapter

- Main flow of an expanded use case
- Alternate flows: exception handlers and variants
- Writing recommendations
- System sequence diagrams

5.1 Introduction to expanded use cases

As explained in [Chapter 3](#), use cases may be used to represent system requirements, among other things. They are nowadays one of the leading approaches for requirements because they allow the building of a structure that is easier to understand, communicate, and manage than previous approaches. Ivar Jacobson identifies some reasons why use cases became so popular:¹

- A use case model is a picture that allows a team to describe even a complex system. It tells in simple terms what the system is going to do for its users.
- A use case produces value to a particular user, not to an unidentifiable community of them.
- Use cases are also test cases; when the team finishes organizing requirements in use cases they have also produced the structure needed for test cases.
- Use cases are the starting point to design effective user experiences, such as, for example, a website.
- Use cases drive development from analysis to design, from design to code, and from code to testing.

Frequently, the first analysis activity of Elaboration iterations is to detail the use cases

assigned to the iteration. The process of detailing or *expanding* a use case into a sequence of steps corresponds to refining the requirements analysis, especially the functional requirements. In each iteration, it is possible to expand one or more use cases and analyze and design the necessary structures for the software product to meet the requirements, allowing users to interact with the system by following the steps indicated in the expanded use case.

In order to proceed with the expansion of a given high-level use case, it is necessary to have at hand the system use case diagram, including all annotations, and the supplementary specifications document. It would also be useful, for the first iteration, to have a version of the preliminary conceptual model if it was developed during Inception. For iterations other than the first Elaboration iteration it is very useful to have a version of the evolving conceptual model already refined by the activities of past iterations.

As use case expansion corresponds to the refinement of the requirements analysis, the team must conduct a very detailed exam on the process incorporated into the use case *from the point of view of the user*. The use case must be described step by step: how it is performed and which interactions between actors and system do exist. Technology and interfaces should be ignored at this point. *Only the information exchanged between the system and the actors is mandatory in these descriptions*.

This step-by-step description, in principle, must not be structured with branches. It must be based on a *default sequence*, or *main flow* ([Section 5.2](#)), which describes what happens when everything goes right during the use case interaction. This flow is also called the “*happy path*” because in it there are no error, failure, or exception.

After describing the main flow of the use case, the business rules or nonfunctional requirements annotated for the use case must be checked and analyzed from the point of view of the user, in order to discover what could possibly go wrong in a given step, which could generate an *exception*. After identifying a possible exception, a description of a procedure to correct the problem should be described ([Section 5.3.3](#)). There may be at least one exception for each business rule. The use case is then completed with *alternate flows* that resemble the *handlers* used for exception handling in programming.

Besides the above, the team can verify if the user may be able to execute the use case in a number of different ways or *scenarios* ([Section 5.3.1](#)). If its execution structure varies greatly, then *variants* to the main flow may be identified ([Section 5.3.2](#)).

5.2 Main flow

The *main flow* is the most important section of an expanded use case. It is the description of the use case process when everything goes right, that is, when no alternate flow occurs.

In the bookstore example, a first draft of the main flow of the *Order books* use case may include the customer identifying herself, searching and selecting books, and finally proceeding to payment.

Regarding user identification, a recurrent problem arises here: What if the user has already identified herself on a use case that was executed just before the current one?

This is a common situation and must be adequately dealt with. As at this point user identification is optional, it may be left as an alternate flow, as explained in [Section 5.3](#). Thus, it is assumed for the main flow of this use case that the user is already known by the system; if this is not the case, an alternate flow where the user identifies herself should be performed.

Therefore, the actions that better describe the main flow of use case *Order books* are the following: a user searches and selects books for buying and proceeds to payment. However, this is not a one-way communication, and the system must present to the user which books are available for sale, as well as other information that is important to the user such as the books' prices, and the total value of the order.

[Figure 5.1](#) presents a draft for the main flow of the *Order books* use case, which is considered the highest priority use case of the Livir system ([Section 4.3.5](#)).

Use case 01: <i>Order books</i>
<ol style="list-style-type: none">1. The customer provides keywords for searching books.2. The system generates a list of books for sale that matches the keywords, including at least title, author, price, page count, publisher, ISBN, and cover image.3. The customer selects books from the list and indicates the desired quantity for each one.4. The system generates the order summary (title, author, quantity, unit price, and subtotal for each book) and total value.5. The customer finishes the order.

FIGURE 5.1 First draft for the main flow of the *Order books* use case.

At step 1 the user may choose not to provide a keyword, and then at step 2 she would receive a list with all the books available at the bookstore. For now it is not necessary to decide on the interface format. For example, if the list of books is too long, maybe a limited list with scrolling commands would be preferable to a complete list. This can be decided when the interface is being designed. The interaction of the user with the scrolling commands of the interface is something that should not be included in the essential use case description.

In order to execute step 5 it is necessary that the user is identified and validated, or else how would she resume the order later? The main flow here assumes that she is already identified, but if this is not the case, an exception would be raised as explained in [Section 5.3.3](#).

After step 5 it is assumed that the use case finishes and the shopping cart has been saved (as it is an internal operation, it is not necessarily mentioned in the use case, as explained in [Section 5.4.6](#)). At this point, the user can chose to log off and resume shopping some other day, or she can proceed immediately to the *Pay order* use case ([Figure 5.2](#)).

1. The system generates the pending order summary (title, author, quantity, unit price, and subtotal for each book), as well as the total price of the order, and a list of registered delivery addresses for the customer.
 2. The customer selects a delivery address.
 3. The system presents the delivery fee, the scheduled arrival date, and a list of credit cards already registered for the customer (flag and last four digits of the credit card number).
 4. The customer selects one of her credit cards for payment.
 5. The system sends to the respective credit card operator the following data: card number, owner's name, validity, security code, total purchase value, and code of the store.
 6. The credit card operator approves the sale by sending an authorization code.
 7. The system informs the customer that the sale was approved and supplies the tracking number of the package.
-

FIGURE 5.2 First draft of the main flow of the *Pay order* use case.

Note that these use cases are drafts because we still must discuss some issues and make some improvements in their formulations, which will occur in the upcoming sections. For example, the alternate flows that will be presented later are related to the possible exceptions and variants identified by the team. For example, step 3 of *Order books* may present an exception such as insufficient quantity of the book in stock. Another evident exception may happen in step 5 of *Pay order*, in which the credit card operator approves the sale: although this is expected for the success of the use case, it may not happen.

Exceptions, however, instead of producing conditional tests in the main flow, must be ignored by it, and handled separately. It would be harder to understand the purpose of a use case that has in its body an algorithmic structure that considers all possible exceptions, like the one shown in [Figure 5.3](#).

1. If the customer does not have a record, then she registers, indicating her name, phone, address, social security number, and at least one credit card. If she is already registered, then she identifies herself, unless she was already identified before.
 2. If the customer still has not paid for the last sale, then the system shows the shopping cart, so that it can be paid, canceled, or resumed.
 3. The system presents the options of books for sale, indicating at least the title, author, price, page count, publisher, ISBN, and cover image.
 4. The customer selects books, and indicates the desired quantity of each one. If the desired quantity is not available in stock, then the system must ask the user to redo the order considering the available quantity. If the quantity is zero, then the customer must have the option of taking that item off the order. It is possible still for the customer to split her order into two parts, receiving as soon as possible the available books and receiving a second delivery with the missing books when possible.
 5. ...
-

FIGURE 5.3 Use case with exceptions inadequately handled in the main flow.

The problem with the use case flow in [Figure 5.3](#) that makes it so unnecessarily complex is the quantity of “*if-then*” statements in its formulation. The main flow should not be sprinkled with conditions. It should indicate what would happen in an ideal situation in which all information passed is correct and complete. Exceptions and other variations to the main flow should be handled as alternate flows, as seen in the following section.

5.3 Alternate flows

A use case specifies a process that can be performed in real life in a number of different ways. Two people buying books would take different sequences of steps. If the sequences are sufficiently similar,² then the same flow must be used to describe both. But in some situations alternate flows must be used to indicate sequences that could occur in quite different ways.

There are basically two kinds of alternate flows: *variants* ([Section 5.3.2](#)), which indicate different ways to reach the same goal, and *exception handlers* ([Section 5.3.3](#)), which indicate how to deal with problems that appear during a use case execution.

However, sometimes the variants are so different that they can be considered individual use cases. Then, variants may be simple scenarios ([Section 5.3.1](#)) of a single use case, or, sometimes, they may be the key to discovering brand new use cases.

5.3.1 Scenarios

A use case can be understood as a description or general specification that supports a set of different *scenarios*. Each scenario is a particular realization or *instance* of the use case.

Usually, a use case has a *main scenario* (execution of the main flow) and *alternate scenarios* (executions of the main flow that pass by one or more alternate flows). However, the notion of main flow variants sometimes creates doubt on what should really be a use case. In the bookstore example, considering the process for buying books, at least two different scenarios come to mind:

- A customer orders books and saves the shopping cart to continue shopping later.
- A customer orders books and pays for them.

Is it a single use case with two alternative ends? Or should it be split into two use cases?

Both options are equally valid, although one of them is possibly more useful than the other. The choices are:

- Create more use cases (one for each scenario). Each use case has a simpler structure, but there would be a high number of similar use cases.
- Group similar scenarios in a single use case. There are a smaller number of use cases, but each of them would be more complex.

At this point, it is more important to discover the information exchanged between the actors and the system than to decide if two scenarios consist of one or two use cases. Thus, it is not so important, with regard to the final result, if the operations are discovered or described in a single use case with two scenarios or in two use cases each with a single scenario.

The advantage of joining alternate scenarios into a single use case is that the description of some steps must not be repeated in different use cases. This is the same advantage that is obtained when the attributes of different classes are shared by a superclass ([Section 6.6.1](#)).

However, use cases must not be considered similar to procedure calls: they are linear descriptions of scenarios that may happen in the real world where a user is interacting with a system.

5.3.2 Variants

It is admitted, in principle, that the main flow is a strict sequence of interaction steps: it is neither branched nor nested. However, sometimes it could be useful to represent the main sequence in a not-so-linear way.

As mentioned in the previous section, use cases for ordering books and paying for them could be considered a single use case with two scenarios. In that case, each scenario would be an alternate end for the use case: *Save shopping cart* or *Proceed to checkout*.

These two possible endings may be represented as *variants* of the main flow of a single use case.

As shown in [Figure 5.4](#), the use case *Buy books* has two possible ends: in the first, the cart is saved so that the order can be resumed later, and in the second, the customer pays for and finishes the order.

1. The customer provides keywords for searching books.
2. The system generates a list of books for sale that matches the keywords, including at least the title, author, price, page count, publisher, ISBN, and cover image.
3. The customer selects books from the list and indicates the desired quantity for each one.
4. The system generates the order summary (title, author, quantity, unit price, and subtotal for each book) and total value.
5. The customer finishes the order.
6. The customer decides:
 - Save shopping cart: Variant 6a.
 - Proceed to checkout: Variant 6b.

Variant 6a: *Save shopping cart*

- 6a.1. The customer informs the system that the shopping cart should be saved.
- 6a.2. The system informs the customer how many days the cart will be available.

Variant 6b: *Proceed to checkout*

- 6b.1. The system generates a list of registered addresses for the customer.
 - 6b.2. The customer selects a delivery address.
 - 6b.3. The system presents the delivery fee, the date scheduled for arrival and a list of credit cards already registered for the customer.
 - 6b.4. The customer selects one of her cards for payment.
 - 6b.5. The system sends to the respective credit card operator the following data: card number, owner's name, validity, security code, total purchase value, and code of the store.
 - 6b.6. The credit card operator approves the sale by sending an authorization code.
 - 6b.7. The system informs the customer that the sale was approved and supplies the tracking number.
-

FIGURE 5.4 Main flow of a use case with variants.

Saving the cart to resume the order later is not an exception (in the sense given in [Section 5.3.3](#)) but an *option* for the customer. That is not a condition that prevents the use case from concluding, because the use case still produces something: the cart is stored to be used later. This is just a main flow variant, and this is why both variants are shown with the same status in the flow.³

Some analysts might choose one of the variants to be in the main flow and the other to be an alternative flow in the form of a variant. In [Figure 5.5](#) the variant *Proceed to checkout* was chosen to be in the main flow, while the variant *Save shopping cart* was left out of the main flow. An advantage of this choice is that it allows the variant to be associated to

more than one step in the main flow. For example, consider that the variant *Save shopping cart* may be executed when the user is in any step from 1 to 8. In this case the variant code would be numbered 1-8*a*. If the variant could be executed at any time during the main flow, then its number would be **a*.

Use case 01/02: *Buy books*

1. The customer provides keywords for searching books.
2. The system presents a list of books for sale that matches the keywords, including at least the title, author, price, page count, publisher, ISBN, and cover image.
3. The customer selects books from the list and indicates the desired quantity for each one.
4. The system generates the order summary (title, author, quantity, unit price, and subtotal for each book) and total value.
5. The customer finishes the order.
6. The system generates a list of registered addresses for the customer.
7. The customer selects a delivery address.
8. The system presents the delivery fee, the scheduled arrival date and a list of credit cards already registered for the customer.
9. The customer selects one of her cards for payment.
10. The system sends to the respective credit card operator the following data: card number, owner's name, validity, security code, total purchase value, and code of the store.
11. The credit card operator approves the sale by sending an authorization code.
12. The system informs the customer that the sale was approved and supplies the tracking number of the package.

Variant 1-8a: *Save shopping cart*

- 1-8a.1. Customer informs the system that the shopping cart must be saved.
- 1-8a.2. System informs the customer how many days the cart will be available.

FIGURE 5.5 Alternative way to represent a use case with variants where one is kept in main flow and the others are alternate flow.

This style, although predominant in literature, forces the use case analyst to make a choice between variants that may be artificial in some cases. If there is not a clear predominance among the different variants, then a style such as the one presented in [Figure 5.4](#) seems preferable, because it gives the same status to each variant.

5.3.3 Exception handling

After describing the main flow of a use case and adding variants (if any), the analyst must concentrate on identifying the *exceptions* that could occur and create an alternate

flow for each one. There are basically two techniques for identifying exceptions:

1. Each business rule annotated for a use case may generate an exception. For example, if the use case for ordering books has a business rule that states that the value of the order cannot be lower than 50 dollars, then when an order is confirmed there will be an exception if it does not reach that value. To deal with this exception it will be necessary to add more books to the order, or it will be canceled.
2. Additionally, the analyst may look at each of the steps of the main flow, especially those that represent information flowing from actors to the system. In those flows there will be possibly data that the system cannot accept by definition, which constitutes an exception. For example, in the step where the credit card operator is asked to authorize the sale it could send a refusal code – this would be an exception, because the use case cannot be successfully completed in that case. Corrective actions must be taken (outside the main flow), or the use case will be terminated without achieving its goals.

A *exception* (in the sense used in computer science) is not necessarily an event that seldom occurs, but an event that if not handled prevents a process from proceeding. An exception is not necessarily something that prevents a use case from starting, but it is normally something that prevents it from being concluded. The fact that the credit card has reached its limit does not prevent the customer from accessing the bookstore site and starting to shop, but it prevents her from completing the order unless a valid credit card with available credit is provided. Therefore, although the use case may be initiated, it cannot be concluded until every exception that occurs is handled.

Figure 5.6 shows an evolution of the use case *Order books*, now with some exceptions identified after the main flow. Each exception is identified by a number that indicates the line of the main flow where it originates, and a letter that helps differentiate exceptions that may happen in the same step. For example, if step 5 has three exceptions, then they will be identified as 5a, 5b, and 5c. If an exception can occur in any step, then it is identified as *a, *b, etc., and if it can occur in a range of steps, such as 3-7, than it is identified as 3-7a, 3-7b, etc. Additionally, if an exception can occur in nonadjacent steps, such as steps 3 and 6, than it may be identified as 3,6a. A sentence follow the identification code that corresponds to the business rule or situation that is the cause of the exception. Figure 5.6 also introduces variant 5d that allows the user to continue shopping for other books instead of finishing the order. This is not an exception, just a choice.

1. The customer provides keywords for searching books.
2. The system generates a list of books for sale that matches the keywords, including at least the title, author, price, page count, publisher, ISBN, and cover image.
3. The customer selects books from the list and indicates the desired quantity for each one.
4. The system generates the order summary (title, author, quantity, unit price, and subtotal for each book) and total value.
5. The customer finishes the order.

Exception 3a: The quantity requested is higher than the quantity in stock for one or more books.

Exception 5a: The customer is not identified yet.

Exception 5b: The customer does not have an account.

Exception 5c: No book was selected for purchase.

FIGURE 5.6 Example of use case with exceptions identified.

The number of exceptions a team may come up with may be very high; however, we are interested in just a few of them. Most exceptions that could be thought of may be resolved by other mechanisms than the logic of the use case. A key for reducing the use case complexity and still dealing with the system complexity is to reduce the number of exceptions identified here to those that really must be addressed. The exceptions that are really interesting now are the ones where the user tries to send the system some data that cannot be accepted. Any other kind of exception probably can be resolved by other mechanisms. In the example of [Figure 5.6](#):

- In step 1 there can be no exceptions because the system may accept any keyword that the user provides. It just happens that for some choice of keywords there may not be any book to present. In this case, the behavior of the system should show no books, or maybe show the books that most closely meet the keywords (Google does that). This is not an exception that prevents the use case from continuing, just a special case of its behavior.
- In step 2, and any other step where the system is presenting data, there can be no exceptions in logic, because in step 2 the system is not trying to accommodate data sent by the user; if some exception could occur, then it probably would have happened in the previous step and generated an effect here. The system showing a list with no books is not an exception. Again it is a special case of its normal behavior, and the user action would be choosing no book in step 3 and after that repeating step 1 or proceeding to finish the order in step 5.
- Still in step 2, an analyst could think “What if the calculations involved with the query have a defect that makes the system halt when trying to present the result?” Well, if the code has a defect, then it must be corrected, and not incorporated in the use case

logic. This is not the kind of exception we are interested in.

- At any step, an analyst could think that an exception could happen if the communication between the system and the user fails. That is what developers usually understand as a *real world* exception. However, this is not the kind of exception we are interested in here, because it is not specific to the logic of the use case; communication may fail at any time – it does not matter which step of which use case is being executed. Additionally, this exception is related to *technology*, and we are dealing with essential use cases, where references to implementation technology are avoided. Thus, this kind of exception must be implemented by general mechanisms, not by the logic of the use case.
- In step 3, what if the user indicates a negative quantity? What if she writes a letter in the quantity field? Here the idea is that all bad user actions could be eliminated by simple interface widgets. For example, instead of writing a number in a text box, the user selects a quantity by sliding a bar that represents only positive quantities. These interface mechanisms are part of technology and they are not to be mentioned in the essential use case. However, the analyst may know that there are ways to prevent a user from choosing a negative or a “xyz” quantity, and therefore this is not a preoccupation that must be included in the use case structure. As explained in [Chapter 8](#), this kind of *syntactic* user error will be addressed by defining the type of arguments that the system can receive. If the type is defined as “natural,⁴” then the team must not be concerned with any other values; they simply cannot be produced by the user. The interface will not allow it.
- However, there is a logical exception in step 3, which corresponds to the user asking for a quantity that is more than is available in stock. The quantity of books available in stock varies from book to book and varies in time. Thus it cannot be defined as a static type. The system could prevent the user from choosing a quantity that is higher than available, but in that case, the system would have to check for those quantities prior to the user action.
- Finally, step 5 has at least three exceptions that are just the type we are looking for; three problems that are related to the use case logic and that prevent the use case from achieving its goals: a user that is not identified yet, a user that has no valid identification, and an order with no books selected.

Through the discussion above we can conclude that in practice exceptions happen usually in steps that correspond to information being sent *to* the system, because only in those cases are there business rules or situations that prevent the system from accepting the data and proceeding further. Lots of potential exceptions may be treated as type check or technological issues, as explained above. And in most cases, exceptions identified in a step where the system is presenting information should have been dealt with at the previous step where the user was sending information to the system. In summary, *interesting exceptions in a use case flow are those in which the user tries to send information that the system cannot accept.*

Each exception must be *handled* by an alternate flow that corresponds to a branch of the main flow. This alternate flow defines a sequence of steps that should be performed

to eliminate the problem created by the exception. If this sequence of steps cannot be performed or if the actor does not want to perform them, the only option left is to abort the use case, that is, terminate it without reaching any of its goals.

Therefore, each exception identified in a use case has an *exception handler*. An exception handler has at least four parts:

- An *identification code* that, as explained above, consists of the number of the step(s) that originates the exception followed by a letter that identifies the exception.
- The *exception*, which consists of a sentence that states which business rule was broken, because in a single line in the main flow different kinds of exceptions may happen, as in, for example, step 5 of the use case of [Figure 5.6](#).
- *Corrective actions* that consist of an alternate flow, that is, a sequence of actions that should be performed to correct the exception. The corrective actions are numbered sequentially and each step is prefixed by the exception identification. For example, exception 2a has steps numbered as 2a.1, 2a.2, 2a.3, etc.
- *Finalization*, which is the last line of the alternate flow and indicates if and how the use case returns to the main flow after performing the corrective actions.

There are five basic ways to terminate a corrective actions sequence:

1. *Terminate the use case normally after the corrective actions are performed.* By default, if no finalization action is mentioned, it is assumed that the use case terminates at the end of the alternate flow. The use case does not return to the main flow but its goals are reached by the alternate flow.
2. *Go back to a step before the one that caused the exception.* Sometimes it is necessary to go back to the beginning of the use case or move back some steps before the one that caused the exception. This is neither very common nor desirable, because it implies that the user will repeat steps that have already been taken. However in systems that must obtain data in real time or systems with intensive concurrent transactions this may be the case.
3. *Return to the step that caused the exception and perform it again.* This is more usual. One must choose this option when the step that caused the exception could eventually cause other exceptions even if one of them has been already handled. This is the case for the exceptions of step 6 in [Figure 5.6](#), because in addition to not having a valid identification the user could not have selected any books.
4. *Advance to some step after the one that caused the exception.* This can be done when corrective actions have already produced what the step or sequence of steps were supposed to do. However, it is still necessary to check if new exceptions could happen in the main or alternate flow (see the previous option).
5. *Abort the use case.* In this case, the control does not return to the main flow and the use case does not reach its goals. If it is necessary to perform some corrective actions (for example, removing transient records) this may be indicated in the steps of the alternate flow (this approach for handling an exception is known as *organized panic*).

[Figure 5.7](#) presents a new version of the *Order books* use case, now with the corrective actions for the alternate flows defined.

1. The customer provides keywords for searching books.
2. The system generates a list of books for sale that matches the keywords, including at least the title, author, price, page count, publisher, ISBN, and cover image.
3. The customer selects books from the list and indicates the desired quantity for each one.
4. The system generates the order summary (title, author, quantity, unit price, and subtotal for each book) and total value.
5. The customer finishes the order.

Exception 3a: The quantity requested is higher than the quantity in stock for one or more books.

3a.1. The system informs the available quantities for each book where the user ordered more than is available.

3a.2. The user changes the desired quantities to meet the quantities in stock.*

3a.3. Advance to step 4.

Exception 5a: The customer is not yet identified.

5a.1. The customer provides valid identification.

5a.2. Return to step 5.

Exception 5b: The customer does not have an account.

5b.1. The customer registers herself by providing a username, password, and email address.

5b.2. Return to step 5.

Exception 5c: No book was selected for purchase.

5c.1. Return to step 1.

Variant 5d: The customer decides to continue shopping.

5d.1. Return to step 1.

FIGURE 5.7 Example of use case with exception handlers. *Contrary to what was stated in [chapter 2](#), here we assume for simplicity that books out of stock cannot be sold.

In the example, each exception identified and handled is specific for the step where it occurs. For example, the quantity of books available may only be detected in step 3 when the user provides it.

It is important that only exceptions that are specific to some steps are effectively included in the use case. Otherwise, it could quickly become unnecessarily complex. Generic exceptions that may occur at any step of any use case must not be treated as exceptions here. For example, the actor may cancel a process at any step. That kind of situation is more adequately handled by generic mechanisms. For this example, a generic mechanism of rollback should be provided for the system. If this kind of exception was included in every use case, then a huge number of similar exceptions would be

introduced with no practical result. If at any step of any use case the user can cancel the transaction, it is not necessary to repeat that information many times. There could exist hundreds or thousands of steps in the set of use cases of a system. Generic exceptions that are specific to a single use case may be numbered with the "*" notation. Otherwise, if the exception may happen in any use case or in a great number of them, then it must be recorded in the supplementary specifications document, and therefore must be implemented by general mechanisms, not by specific use case steps.

5.4 Writing recommendations

The writing style of a use case should follow a pattern like "actor provides... / system informs...." Styles like "system asks..." should be avoided because what must be represented in the use case is the *flow* of information, not the occasional *demands* that gave rise to those flows; those demands do not exist in every scenario, and in most of them they are not even necessary.

The team must avoid placing tests in the use case main flow. Tests like "if the user has a record *then* the system presents..." should be avoided. That kind of test is unnecessary because exception handlers should already be associated to the steps of the main flow. Then, if an exception occurs, it would be handled in an alternate flow.

These tests must not be placed in the main flow to keep it from becoming a complex flowchart instead of a straight, simple sequence of steps. Lots of "if/then" statements can make the flow obscure, and the stakeholder would not know what the *expected* flow (happy path) is.

The only situation in which a test would be acceptable in a flow is when it is about an optional single step, as, for example, "if the user wants, she may also provide her cell phone number." But even in that case, an alternate flow would be a better choice.

Another common writing concern is that a use case must alternate inputs and outputs of information. In other words, normally a use case has the form "actor provides ... , system informs ... , actor provides ... , system informs"

Sequences such as "actor provides... , actor provides ... , actor provides ..." or "system informs ... , system informs ... , system informs ..." must be avoided unless justified.

The idea behind the alternating pattern is to reduce any discrepancy on the number of steps that different analysts could assign to the same use case. Without that rule one analyst could write:

1. The customer provides her name, ID,⁵ and phone number.

On the other hand, another analyst could write:

1. The customer provides her name.
2. The customer provides her ID.
3. The customer provides her phone number.

The most useful and correct version under this point of view is the first, because, among other things, the second would demand that the information be provided in the order of the steps: after providing her *ID*, the user cannot go back and correct her name. Thus, the first version is more compatible with most information systems, which present

interfaces where information may be entered and edited before it is finally sent to the system.

A sequence of two input steps can be justified only if the first step may cause an exception that can prevent the use case from proceeding further. One example of this is:

1. The customer provides identification.
2. The customer provides the flag, number, expiration date, and security code of her credit card.

In this example, when the customer presents identification, an exception would occur if the identification is invalid. In that case, the information about the credit card must not be given by the user. The actor will only give that information after the first step is performed or corrective actions were taken.

Even in that case, an intermediary validation step could be used to improve the understandability of the sequence, such as:

1. The customer provides identification.
2. The system validates customer identification.
3. The customer provides the flag, number, expiration date, and security code of her credit card.

Finally, some system messages may be avoided in the use case flow because they do not involve the system presenting stored information. For example, the first step for the alternate flow for exception 5a in [Figure 5.5](#) could be “The system informs the user that the customer was not yet identified.” That is not an error, but the step is somewhat redundant in the flow because the exception already states the condition that causes it. When designing an interface for handling that exception the team should be concerned about error messages like that. However, it is not necessary for understanding the flow of information of the use case. Consider that if it was necessary then any exception handler in any use case should start by presenting an exception message, this can already be inferred by the exception condition. It can be assumed by default that the message *will* be shown, and thus mentioning it every time is not necessary.

5.4.1 Essential versus real use case

A frequent doubt about use cases is *what* system must be described? The current one? Or the one that will be built? If the current operations are performed manually, and in the future they will be performed with a computer, what must be the description provided by the use case? The answer is *none*, but *both*! The use case written for analysis purposes must describe the *essence* of the operations, not their concrete realization. Thus, the analyst should always try to avoid mentioning the technology used to perform the process and concentrate only on the exchanged information. Instead of writing “the clerk writes the customer name and address on a paper sheet,” corresponding to a manual technology used normally in nonautomated systems, or “the customer fills in the name and address on the main screen,” which corresponds to an automated technology, the analyst must register the use case simply as “the customer provides a name and address.” The last form is technology-free and interface-independent and represents,

therefore, an essential description for the operation.

All use cases used for requirements purposes should be, in principle, *essential*. Essential in this context means that the use case is described in a language where only the essence of the operations is presented, not its concrete counterpart. In other words, the analyst must describe “what” happens between the user and the system without informing “how” that interaction happens. The analyst should not, therefore, while doing analysis, try to describe the interface technology between the user and system. This will be described during design time in which *real* use cases could be written after the real interface has been designed. Real use cases stay attached to a specific interface technology, while essential use cases are technology-free.

Why essential use cases? According to [Ambler \(2004\)](#), real use cases (those that mention technology) contain too many built-in assumptions, often hidden or implicit about the underlying technology of interface implementation strategies. That may be a good feature during design, but not during requirements analysis. An essential use case, on the other hand, states the user needs or intentions, and not the concrete technology that could support those needs.

As the description of the use case at that moment is made without reference to the interface technology, it is not necessary to decide what the interfaces look like, but only what *information* is exchanged between actors and system. References to menus, windows, buttons, and other graphical devices must be avoided in the text of the essential use cases.

So how do we describe use cases that seem to be particularly concrete such as, for example, drawing money from an automatic teller machine? How do we do that without describing technology? It is possible. Instead of saying “the user passes a magnetic card” one could say that “the user provides identification.” Instead of saying that the machine “prints the account details” one could say that the machine “presents the account details.” Thus, by eliminating the reference to the physical technology, only the essence remains. This opens the way for the design of innovative technologies later, that is, interfaces different than those that can be seen at a first glance. For example, the user could be identified by fingerprints, the receipt could be sent by SMS (Short Message Service), and even the money could be delivered in cash or in credits to a preloaded card.

It is the responsibility of the analyst to study the current processes of the company and produce an essential version of them in the form of use cases. Later, the designer will give a new form to those processes by using technology to produce a new concrete version of the system.

5.4.2 Explicit information

For the sake of understanding use cases as functional requirements, it is recommended that the analyst make explicit the individual data pieces exchanged between actors and the system. Thus, instead of a laconic “the customer provides her data,” the use case should say “the customer provides her name, ID, email, phone, and address.”

Eventually, a term such as “customer” could be externally defined as meaning “name,

ID, email, phone and address.” An expression such as *customer=<name, id, email, phone, address>* could be written. If that definition is adequately registered, for example, in the conceptual model, or in a data dictionary, then it can be used in the use case text to avoid repeating the individual elements every time they are needed.

5.4.3 Identification and selection

A recurrent problem associated to use cases is how to identify people and things. For example, a customer must be identified and validated in order to conclude the *Order books* use case. How does she do it? Username and password? Social security number? Fingerprints? Selecting the name from a predefined list or menu? To keep the use case essential, that particular action must be left generic, and references to possible ways for identifying a customer must be left to design. Furthermore, the computer security community already has established common methods to securely validate an identity, and these proven methods should be used instead of reinventing the wheel. For now, the use case will simply state that the user *provides identification*, without mentioning by which means that identification was obtained and validated. This can be done when security issues are designed into the system.

Another example is the selection of books for purchase. Does the customer submit the ISBN of the book? Select it from a list? Use a natural language speech interface? During use case expansion these issues do not need to be decided (unless they are indeed nonfunctional or supplementary requirements). At the essential level of the use case the user simply *selects books*. Later, the design will present some options to provide the means for the user to make that selection in the most comfortable way.

5.4.4 Mandatory steps

One of the biggest doubts reported by teams that work with use cases is about what steps to include in the expanded use case.

In fact, two people that describe the same use case, unless they have a nicely established convention, may produce different step sequences. For example, in a traditional bookstore, one analyst could state that the clerk asks the customer which books she wants; but another analyst could ignore that step and state that the customer simply picks up the books and give them to the clerk for purchase. The question is: Which one of the two versions is right?

Both may be correct, but one of them is more useful. In any scenario of that use case, it is *mandatory* that the customer selects the books she intends to buy; otherwise the order cannot be completed. However, it is optional for the clerk to *ask* for the books. That may happen just in some scenarios, but not in all of them. Thus, selecting books to buy is mandatory, but asking for that information is optional in the flow of the use case.

Analysts must be encouraged to build correct versions of the use cases, but we should not stop there: different analysts should build similar descriptions for the same process. This can be obtained when the number of optional steps is minimized.

Every use case has *mandatory steps*. These steps include information that is passed from the actors to the system and from the system to the actors (Figure 5.8). Without one of those steps, the use case may not make sense. Those steps must appear in any expanded use case, and their absence implies that the use case is incorrect.

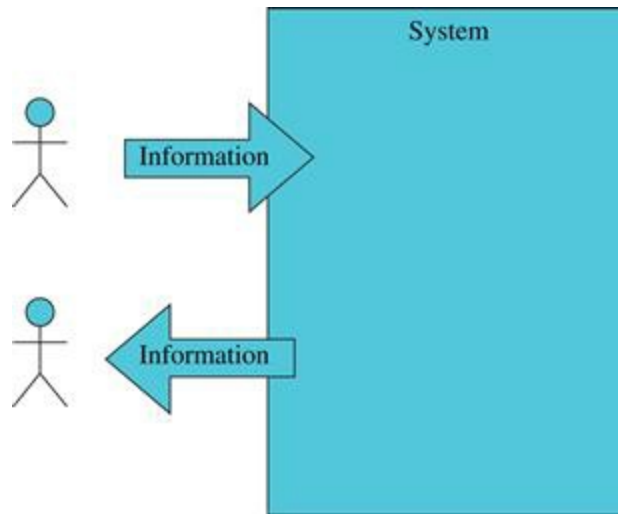


FIGURE 5.8 Mandatory use case steps.

Other steps such as “ask for books” are optional or *complementary*. They may help contextualize the use case, but they are not fundamental because they do not transmit any information through the system boundary.

Thus, an expanded use case for buying books must contain steps that indicate that the customer selects books to be bought. The customer should also inform the system how many copies she wants if this is the case. The system must inform the customer of the total value of the purchase. Why are these steps mandatory? Because without exchanging this information it would be impossible to adequately register a sale. Certainly it would not be of great help if a system registers a sale without mentioning which books were sold or how much was spent. Because that information is so important for the conclusion of the use case it is considered mandatory for the expanded use case.

In a use case description the information does not appear from nowhere. It is transmitted from the actors to the system and vice versa. The absence of any mandatory step may make the use case seem incomplete, as shown in Figure 5.9, where a use case was poorly described because mandatory information was omitted.

1. The customer provides keywords for searching books.
 2. The system presents a list of books for sale that matches the keywords, including at least the title, author, price, page count, publisher, ISBN, and cover image.
 3. The system generates the order summary (title, author, quantity, unit price, and subtotal for each book) and total value.
 4. The customer finishes the order.
-

FIGURE 5.9 An example of a poorly described use case that is missing a mandatory step.

The use case in [Figure 5.9](#) is incomplete because an order would need more information than the use case description provides. How could the system know which books the customer wants to buy? No matter how the information is obtained, it is mandatory data that must be received by the system, and therefore it must be registered in the use case.

There is a clear reason for making all mandatory information appear in the expanded use case: this information will be used later to define what the system operations are, that is, what methods have to be implemented in order to satisfy the requirements. If this information is not complete at this point, incomplete methods would be implemented later, and that will raise a need to redo part of the system when the problem is identified.

Furthermore, the information contained in the mandatory steps of the use cases will be used to refine the conceptual model, which is the basis for the structure of the classes of the system.

Depending on the direction the information goes, mandatory steps may be identified as:

- *System events*: Steps that indicate that some information is transmitted from the actors to the system.
- *System returns*: Steps that indicate that some information is transmitted from the system to the actors.

Special care must be taken when considering system returns. For those steps to be really mandatory they must pass some information that the system stores or can access. That information must be something that in principle the actors do not have access to except by querying the system.

For example, the system must inform the actor of the total value of the purchase in the use case *Order books*. Otherwise, the actor will not know how much to pay. That information could, maybe, be calculated by the customer if she mentally adds the individual prices of the books, but even so she could not be sure about the total value because information about discounts, taxes, and fees must be provided by the system. The responsibility for providing that information belongs to the *system*.

On the other hand, when a customer sends some information to the system, the interface may provide some kind of feedback to make clear the data were received and adequately processed. However, that feedback (normally a message such as “OK”, or

“done”) is not a system return because it is not stored or derived information about books, customers, and orders. This kind of message is just an interface feedback, and therefore it belongs to the interface technology domain. As it does not consist of stored information it must not be considered as a mandatory step.

It may be interesting if system events and returns are explicitly identified in use cases. A reader that is new to use cases may choose to clearly identify mandatory steps by marking inputs with [IN] and outputs with [OUT]. Those markers may be placed just after the line number, as shown in [Figure 5.10](#).

Use case -01: *Order books*

1. [IN] The customer provides keywords for searching books.
2. [OUT] The system presents a list of books for sale that matches the keywords, including at least the title, author, price, page count, publisher, ISBN, and cover image.
3. [IN] The customer selects books from the list and indicates the desired quantity for each one.
4. [OUT] The system generates the order summary (title, author, quantity, unit price, and subtotal for each book) and total value.
5. [IN] The customer finishes the order.

Exception 3a: The quantity requested is higher than the quantity in stock for one or more books.

- 3a.1. [OUT] The system informs the user of the available quantities for each book where the user ordered more than is available.
- 3a.2. [IN] The customer changes the desired quantities to meet the quantities in stock.
- 3a.3. Advance to step 4.

Exception 5a: The customer is not yet identified.

- 5a.1. [IN] The customer provides valid identification.
- 5a.2. Return to step 5.

Exception 5b: The customer does not have an account.

- 5b.1. [IN] The customer registers herself by providing username, password, and email address.
- 5b.2. Return to step 5.

Exception 5c: No book was selected for purchase.

- 5c.1. Return to step 1.

FIGURE 5.10 A use case with mandatory steps marked.

Those marks can also help the analyst to verify if each step is really a single transaction because no step can be marked with [IN] and [OUT] at the same time. It also helps the

analyst to see if there are sequences of [IN] or sequences of [OUT] that could be joined into a single step.

5.4.5 Complementary steps

Another kind of use case step is the optional or *complementary* step, which does not represent an information flow between actors and the system. However, this kind of step can be used sometimes to help understand the use case context.

This kind of step usually corresponds to communication between actors (communication that does not involve the system, as shown in [Figure 5.11](#)) or to the description of actions or attitudes that also does not configure information flow, such as “the customer opens the bookstore home page,” “the customer decides to buy books,” “the system asks the customer for identification,” or “the clerk asks the customer which books she wants to buy.”

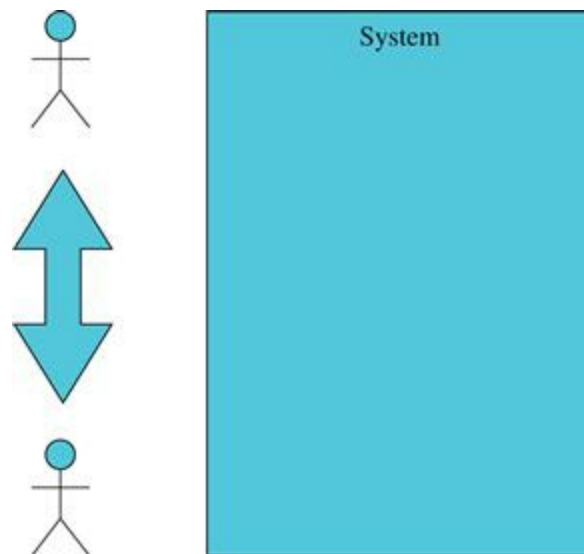


FIGURE 5.11 Complementary use case steps.

Complementary steps are not fundamental in essential use cases because they do not correspond to system events or system returns, as they do not transmit information through the system boundary.

Some of them can be even mapped to navigation operations during interface design. For example, a real use case (design) could have a step like “the customer selects an option.” That line does not correspond to a system event because at that time the customer does not pass any information to the system (such as name, phone, address, book title, etc.). That action corresponds simply to a change of state that will be implemented as interface navigation (a new screen is opened after the option is selected, for example).

5.4.6 Unsuitable steps

The team must keep in mind that, as the use case is a description of the interaction between actors and system, they must avoid including any internal system processes (Figure 5.12), such as “the system stores the information into the database.”. Those steps are considered *unsuitable* for the use case description.

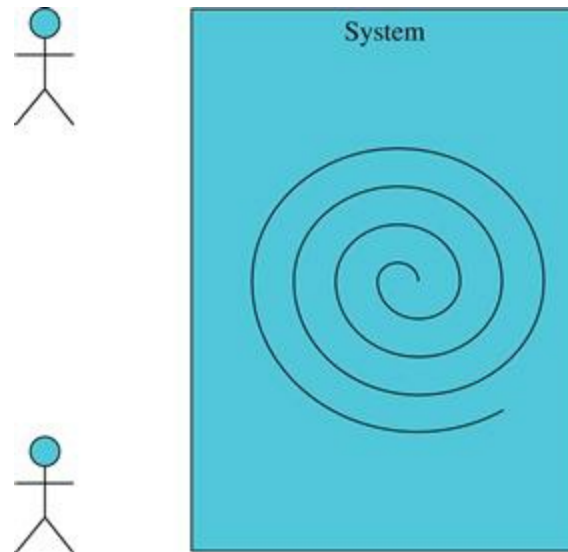


FIGURE 5.12 Unsuitable use case steps.

The use case is a tool for describing the interaction between users and a system, not a tool to describe internal processing. The internal processes of the system will be better described during design with more adequate tools, such as communication diagrams.

In the use case description, the analyst must concentrate on describing the information that is passed from users to the system (for example, “the user selects the books to be ordered”) and from the system to users (for example, “the system presents the total”). Any aspect about internal processing must be omitted. It can be said that the system *presents* the total value, but not the steps taken by the system to calculate that value. In other words, the system still must be seen as a black box at this time.

If the analyst wants to record a formula or algorithm to perform a given calculation that is needed to present some data, then it can be done in the use case annotations and recorded as a nonfunctional requirement.

Figure 5.13 presents a situation where the description of a use case goes beyond the recommended, and unsuitable steps are added. The unsuitable steps are marked with [X].

1. The customer provides keywords for searching books.
 2. [X] The system verifies in the database the books that satisfy the search criteria.
 3. The system generates a list of books for sale that matches the keywords, including at least the title, author, price, page count, publisher, ISBN, and cover image.
 4. The customer selects books from the list and indicates the desired quantity for each one.
 5. [X] The system calculates the total of the sale as the sum of subtotals, which consist of the price of each book multiplied by the respective desired quantity.
 6. The system generates the order summary (title, author, quantity, unit price, and subtotal for each book) and total value.
 7. The customer finishes the order.
 8. [X] The system marks the order as “finished” and stores it in the database.
-

FIGURE 5.13 Example of a use case with unsuitable steps.

The use case in [Figure 5.13](#) includes all mandatory steps but also includes unsuitable ones (steps 2, 5, and 8), which correspond to internal processing, not information input or output.

The fact that the system performs verifications, calculations, and internal queries does not affect its interaction with the user. The user only needs to know that the information she sends is received by the system, and it may be recovered from the system when necessary. How the system is going to do that is an internal problem that will be answered only during design and coding.

5.5 Included use cases and fragments

It is possible for two or more use cases to have coincident parts. For example, many use cases may include a *payment* fragment. In other cases a use case may include not only a fragment, but a complete use case. The *include* relationship can be used only when the duplicated fragment or use case is significant, that is, it should not be used when just one step is duplicated. An included use case or fragment may be called in a flow by using the “*include*” expression. Suppose that *payment* is a fragment that appears in more than one use case in the system. This fragment could then be called by any use case as done in [Figure 5.14](#).

Use case 02: *Pay order*

1. The system generates the pending order summary (title, author, quantity, unit price, and subtotal for each book), as well as the total price of the order, and a list of registered delivery addresses for the customer.
 2. The customer selects a delivery address.
 3. The system presents the delivery fee, and the scheduled arrival date,
 4. Include Fragment 02a: *Payment*.
 5. The system informs the customer that the sale was approved and supplies the tracking number of the package.
-

FIGURE 5.14 A use case that calls an included fragment.

The fragment itself can be defined as in [Figure 5.15](#).

Fragment 02a: *Payment*

1. The system presents a list of credit cards already registered for the customer (flag and last four digits of the credit card number).
 2. The customer selects one of her credit cards for payment.
 3. The system sends to the respective credit card operator the following data: card number, owner's name, validity, security code, purchase total value, and code of the store
 4. The credit card operator approves the sale by sending an authorization code.
-

FIGURE 5.15 A use case fragment that may be included in more than one use case.

Notice that the fragment is not a complete use case in this example. It cannot be executed alone by a user; it only can be performed as part of another use case.

Some care must be taken here. First of all, *use cases are not procedures like the ones used in programming languages*; they are linear descriptions of real world processes. Analysts are not encouraged to structure use cases as they do with code. Included use cases or fragments may be used *only* when the situation really justifies it.

However, the analyst must keep in mind that the goal of the expanded use case in analysis is to help in understanding the nature of the interactions between the system and its actors, and not to structure a computer program to simulate that interaction. Thus the included use case or fragment calls must be used carefully.

5.6 Expansion of stereotyped use cases

Stereotyped or *pattern use cases* present medium or low risk to the software development process because the structure of their flows is already known. The indication of `<<crud>>` or `<<report>>` stereotypes, if adequately documented, may pass to the programmers a clear idea of what has to be implemented.

Ideally, if functional requirements are sufficiently detailed, stereotyped use cases would not need to be detailed. For example, if the specification of a report already presents all information it receives and returns, there is no need to write a detailed use case for it, because it would just repeat information that the analyst already has.

One concern that must be kept in mind during requirements elicitation is whether a use case that was identified really belongs to a pattern. Some analysis effort must be spent on the use cases before deciding if they really belong to the pattern.⁶ For example, reports must never change data. If the use case is presenting data but also changes data then it must not be defined as a *report* use case. It could be a variation of that pattern or a completely new structure that does not belong to any known pattern.

In the case of CRUDs what can be said is that not every entity of the conceptual mode is suitable to be managed by the CRUD pattern. CRUDs are expected to be the simple entities. If they get more complex, a set of non-CRUD use cases is necessary for describing how the user works with them.

For instance, an *address* or a *credit card* are very simple entities and could be nicely managed by a CRUD use case. On the opposite side, an *order* is one of the more complex entities of the bookstore system. Orders may be issued, canceled, completed, sent, resent, received back, etc. That does not characterize it as a CRUD.

In the middle of the scale there are concepts which the team must look at to decide if they may be adequately managed as a CRUD or not. For example, is managing customer a CRUD or not? Some time may be spent on analysis before the team can decide on that. [Section 5.6.2](#) shows that it may be considered a variation of the CRUD pattern.

The expansion models for pattern use cases presented in the following subsections serve as a reference to the reader. They are presented in a form of a template, so that analysts can use them to write their own versions of pattern use cases, or even work on new patterns or subpatterns.

5.6.1 Report expanded

A report is a simple use case that consists of a single access and calculations on data managed by a system. The information sent by the actor consists of arguments for selection, filtering, ordering, grouping, etc. [Figure 5.16](#) presents a general format for an expanded report use case.

Use case template: Report on ... <<report>>

1. User provides ...
2. System presents ..., grouped by ..., and sorted by ...

FIGURE 5.16 A template for the expanded report use case.

[Figure 5.17](#) presents a concrete example of a report, that is, an instantiation for the template, applied to the Livir example.

1. The deposit manager provides initial and final dates.
 2. The system presents a list of returned orders received in the period including the data of the return, the number of the order, the name of the customer, the total value of the order, the reason for the return, and the list of returned books (name, quantity, and price). The orders list is sorted by data, and the returned products list is sorted by name.
-

FIGURE 5.17 An example of an expanded report.

For the reasons explained in [Section 5.3.3](#), this kind of use case does not have alternate flows (otherwise it would not be a report, but a different kind of use case). The user simply sends the arguments and receives a set of organized data in return. If the user passes the wrong arguments (for example, incorrect data), she will receive the right answer for the wrong data. For example, if the user passes by mistake a period in which there were no devolutions, then she will receive an empty list, which corresponds to the right answer for the arguments passed. In that sense, therefore, this kind of use case cannot have exceptions.

However, the team still must assure that the use case is *really* a report before sending it to design and coding, because it could be a variation of that pattern.

5.6.2 CRUD expanded

Just like reports, CRUDs are also use cases that follow a known pattern. This kind of use case consists of the optional execution of one out of four possible operations: create, retrieve, update, and delete.

This kind of use case has a main flow with four variants, because none of the operations can be considered a “happy path” or “exception.” They are distinct operations grouped more by convenience, for being associated to the same entity, than by similarities among their steps. Therefore, the main flow is just a user decision on which operation will be executed. [Figure 5.18](#) presents a possible template for this kind of use case.

Use case template: Manage ... <<crud>>
1. User chooses: <ul style="list-style-type: none"> • Create: Variant 1a • Retrieve: Variant 1b • Update: Variant 1c • Delete: Variant 1d
Variant 1a: Create <ul style="list-style-type: none"> 1a.1. User provides ...
Variant 1b: Retrieve <ul style="list-style-type: none"> 1b.1. User identifies/selects a ... 1b.2. System presents ... of the identified element.
Variant 1c: Update <ul style="list-style-type: none"> 1c.1. Include variant 1b: Retrieve. 1c.2. User provides new values for ...
Variant 1d: Delete <ul style="list-style-type: none"> 1d.1. User identifies/selects a ... to be deleted.
Exception 1a.1a: Inclusion violates business rule ... <ul style="list-style-type: none"> 1a.1a.1. System informs that rule ... prevents inclusion. 1a.1a.2. Return to step 1a.1.
Exception 1c.2a: Update violates business rule ... <ul style="list-style-type: none"> 1c.2a.1. System informs that rule ... prevents the update. 1c.2a.2. Return to step 1c.2.
Exception 1d.1a: Delete violates business or structural rule ... <ul style="list-style-type: none"> 1d.1a.1: System informs that rule ... prevents deletion. 1d.1a.2: Return to step 1d.1.

FIGURE 5.18 A template for an expanded CRUD use case.

Observe that in that use case template variant 1c (*Update*) includes the steps of variant 1b (*Retrieve*).

Exception handler 1d.1a uses a specific strategy that consists of preventing the user from deleting an object when there is some rule against it. In [Section 8.5.4](#), this issue is revisited and other strategies to deal with this exception are discussed.

[Figure 5.19](#) presents an example of expansion of a typical CRUD use case.

Use Case 13: Manage Publisher <<crud>>

1. Acquisition manager chooses:

- Create publisher: Variant 1a
- Retrieve publisher: Variant 1b
- Update publisher: Variant 1c
- Delete publisher: Variant 1d

Variant 1a: Create publisher

1a.1. Acquisition manager provides publisher's name, identification code, address, and email.

Variant 1b: Retrieve publisher

1b.1. Acquisition manager identifies a publisher.

1b.2. System presents publisher's name, identification code, address, and email.

Variant 1c: Update publisher

1c.1. Include variant 1b: Retrieve publisher

1c.2. Acquisition manager provides new values for publisher's name, identification code, address, and email.

Variant 1d: Delete publisher

1d.1: Acquisition manager identifies a publisher to be deleted.

Exception (1a.1,1c.2)a: Identification code provided already exists.

(1a.1,1c.2)a.1: System informs user that there is already a publisher with the provided identification code.

(1a.1,1c.2)a.2: Return to the step that caused the exception.

Exception 1d.1a: There are books associated to the publisher.

1d.1a.1: System informs user that it is not possible to delete a publisher with associated books.

1d.1a.2: Return to step 1d.1.

FIGURE 5.19 An example of an expanded CRUD use case.

In this example, two business rules were used to identify exceptions:

- The identification code for publishers must be unique, which must be verified both when a new instance is included and when an instance is updated. This produces exceptions in step 1a.1 and 1c.2, which may be identified as a single exception (1a.1,1c.2)a.
- If the publisher already has a book associated to it (and that association is mandatory for books, i.e., every book must be linked to a publisher) then it cannot be deleted. This is a structural rule that must appear explicitly in the conceptual model as a mandatory role from book to publisher ([Section 6.4.1](#)).

[Figure 5.20](#) presents the CRUD use case *Manage customers*, which is a variation on the

pattern. A customer can retrieve and modify only her own data. Thus, the use case must be adapted to that fact. On the other hand, it may be desirable that a system manager could access data from any customer. For that manager the use case would be a regular CRUD. Would two use cases be created here? One for the customer to manage her own data and the other for the system manager to manage data from all users? Maybe... but it seems that this approach would be quite repetitive. To make things even worse, imagine that it is possible for a regional manager to access the data of users only from her region. Would this be a third use case? This does not sound like a good solution.

Use Case 11: Manage Customers <<crud>>

1. User chooses:

- Create customer: Variant 1a
- Retrieve customer: Variant 1b
- Update customer: Variant 1c
- Delete customer: Variant 1d

Variant 1a: Create customer

1a.1. User provides customer's name, ID, address, phone, and email.

Variant 1b: Retrieve customer

1b.1. User identifies a customer.

1b.2. System presents customer's name, ID, address, phone, and email.

Variant 1c: Update customer

1c.1. Include variant 1b: Retrieve customer

1c.2. User provides new values for customer's name, ID, address, phone, and email.

Variant 1.d: Delete customer

1d.1: User identifies a customer to be deleted.

Exception (1a.1,1c.2)a: ID already exists.

(1a.1,1c.2)a.1: System informs user that there is already a customer with the given ID.

(1a.1,1c.2)a.2: Return to the step that caused the exception.

Exception (1b.1,1d.1)a: Access not authorized.

(1b.1,1d.1)a.1. System informs user that access is denied.

(1b.1,1d.1)a.2. Return to the step that caused the exception.

Exception 1d.1b: There are orders associated to the customer.

1d.1b.1: System informs user that it is not possible to delete a customer with orders associated to it

FIGURE 5.20 An expanded CRUD use case with access restrictions.

To solve this issue in an elegant way, the use case could be parameterized, indicating that the user can only access data from customers she is authorized to review. When customers are being managed, what must be verified by a given business rule is whether the user really has permission to access or change the data from the customer she is trying to access or change. In [Figure 5.20](#) checking that permission is performed by exception handler (1b.1,1d.1)a: *Access not authorized*.

Notice that the use case in [Figure 5.20](#) is a CRUD variation that could be called *CRUD with access restriction*, because the access to data depends on who is the user.

Other CRUD variations exist, such as:

- **CRUDL**: Including a variant to list the existing elements. Thus, instead of simply identifying an object to retrieve, update, or delete, the user can *select* it from a list.
- **SCRUD**: Including a variant to search elements.
- **RUD**: Where elements may be retrieved, updated, and deleted, but not created.
- **CRU**: Where elements may be created, retrieved, and updated, but not deleted.

Many other variations may be invented depending on the need and the cost/benefit of having more patterns to remember.

There is a simpler way to define a pattern-based use case if it fits perfectly on the pattern. This consists of simply listing the field to be filled and the business rules to be observed. For the rest, the pattern is applied. [Figure 5.21](#) shows an example.

Use case 12: Manage books <<crud>>
Fields:
<ul style="list-style-type: none"> • ISBN, title, author, page count, publisher, price.
Rules for insert and update:
<ul style="list-style-type: none"> • The ISBN is unique, i.e., different books cannot have the same ISBN number.
Rules for delete:
<ul style="list-style-type: none"> • A book that has sold copies cannot be deleted.

FIGURE 5.21 Example of brief expansion for a CRUD use case.

With the information given in [Figure 5.21](#) and the template of [Figure 5.18](#) it is possible to implement the use case without expanding it.

5.7 Other sections of an expanded use case

Since the concept of a use case was created ([Jacobson, Christenson, Jonsson, & Övergaard, 1992](#)), different formats have been proposed. Each proposal includes

different elements. The “main flow” and “alternate flows” sections are fundamental for any detailed use case description. However, other sections can be included if the analyst needs them. Some of the most popular are presented in the next subsections.

5.7.1 Stakeholders

There are often scenarios where groups other than actors are relevant to a use case. Other sectors of the company could have some interest in the use case. For example, in the use case *Order books* the only actor is the customer. But the results of that use case may interest the stock and accounts departments, because the results of a sale may affect both the quantity of books in stock as well as the quantity of money available or to be received. Thus, even if those departments are not participants in the use case, they can be listed as *stakeholders*.

The utility of the stakeholders list for a use case is that the use case must *satisfy all stakeholders*. Thus, the documentation will be useful to remind the analyst to seek information that needs to be stored, processed, or transmitted, so that those interests can be satisfied.

5.7.2 Preconditions

By definition, preconditions are facts considered true before a use case even starts. Preconditions must not be confused with exceptions: exceptions may be detected only after the use case has started. Exceptions are detected during a use case because most of the time it is not possible to verify if the conditions were true or not before the use case begins. For example, it is not possible to assure that the customer has enough credit to pay for the order before starting the *Order books* use case, because the value of the order cannot be known at that time; therefore this is an *exception*. However, it is possible to admit that only a book that was sold and delivered can be returned. Thus, the sale and delivery of the book is a precondition for its return.

As preconditions are accepted as true facts before the use case starts, they will not be checked during the use case. Although there can be other interpretations in literature, these kind of preconditions do not generate exceptions. If they are false it should be impossible to start the use case (otherwise they are exceptions, not preconditions).

5.7.3 Success post-conditions

Success post conditions establish the results of a use case, that is, what will be true after the use case is executed. For example, the *Order books* use case may have as a post-condition the following result: “the order was registered by the system.”

5.7.4 Open issues

Sometimes, without the presence of the client, the team cannot decide about some issue that may depend on company policies. For example, may the customer pay the order in

installments? Are there special offers for customers that buy more than certain quantities?

If the client is not immediately available, these doubts must be recorded in the use case section “open issues” in order to be solved as soon as possible.

At the end of the analysis activities of one iteration it is expected that *all* open issues have been resolved and incorporated into the use case description.

5.8 System sequence diagrams

Among other uses, the text of the expanded use cases may be used as:

- A source of information to discover the concepts and attributes for refining the conceptual model (Section 6.8).
- A source of information to discover *system operations*, which consist of the methods that encapsulate the domain tier of the system from the interface (Chapter 8).

There are two kinds of system operations:

- *System commands*, which change data and must not return data.
- *System queries*, which return data and must not change data.

That follows the *Command-Query Separation* or CQS principle of software engineering (Meyer, 1988), which allows for better code reusability and maintainability. A query that changes data would be less cohesive than a query without collateral effect and a command without return.

System commands are methods that are activated by a *system event*, that is, as a reaction to a user action. System commands, by definition, implement a flow of information from the outside to the inside of the system. Therefore, system commands update the information that is managed by the system.

System queries are methods that correspond to the simple verification of information already stored. That information may be presented exactly as it exists or modified by logical and mathematical operations (such as *sum*, *average*, etc.). By definition, a system query may not be responsible for the insertion of new information into the system. It should also not be responsible for updating or removing information from the system: it can only read information.

The set of system operations correspond to the total functionality of a system, that is, the set of all functions that can be performed by a user with the system.

Use cases are excellent sources for discovering system operations. System commands are basically associated to steps in which the user sends some information to the system (those marked with [IN]). System queries are associated to use case steps in which the system sends information to actors (those marked with [OUT]).

5.8.1 Elements of a sequence diagram

One of the UML diagrams that can be particularly useful for representing the sequence of system events and returns in a use case is the *sequence diagram*. When a sequence diagram is used to represent a use case it may be called a *system sequence diagram*

(Larman, 2004). The sequence diagram has elements that are instances of actors and other system components. In the first version of the system sequence diagram only the actors and the system interface (or application tier) are represented (Figure 5.22).

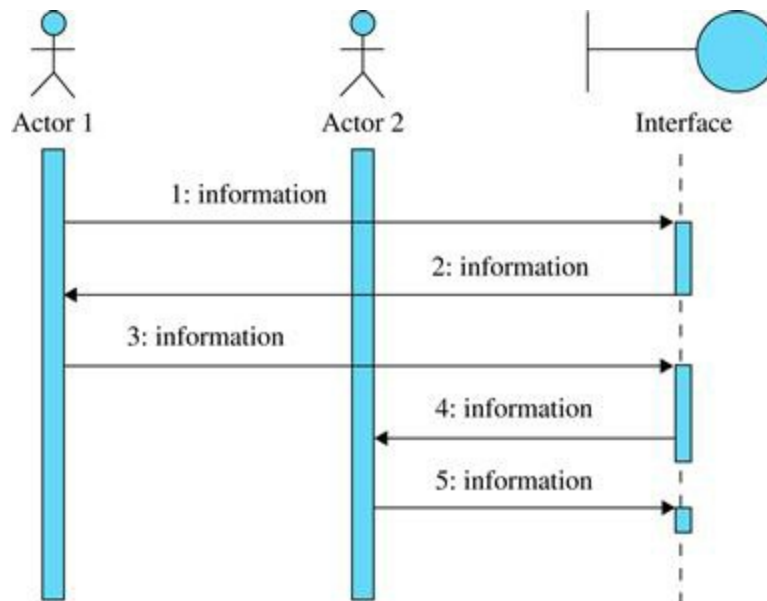


FIGURE 5.22 System sequence diagram.

The activities related to use case analysis still do not deal with objects that are internal to the system. Thus, the system must be represented as a single object: a black box. In this case, it can be represented by the interface symbol $\text{---}\bullet$, as proposed by Jacobson *et al.* (1992). One actor can only communicate with a system through its interface. Messages from the actor cannot reach the objects, as shown in Figure 5.23.

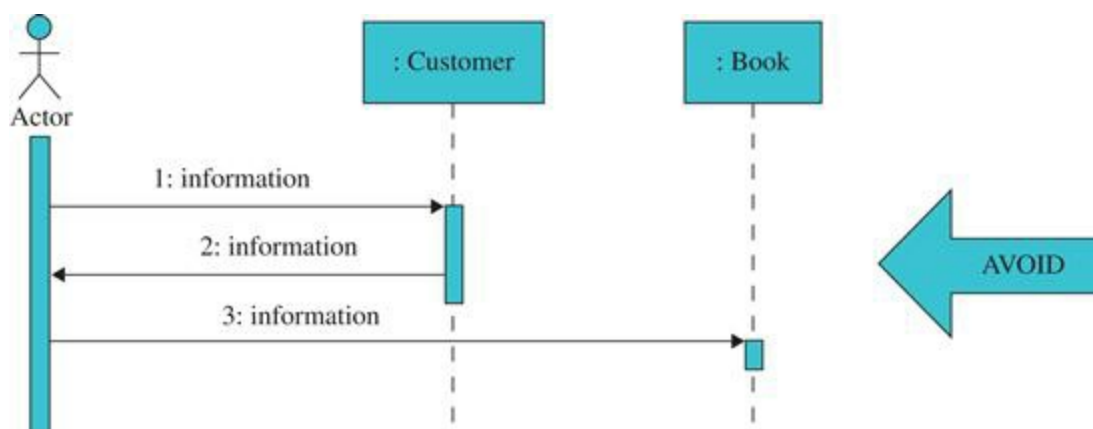


FIGURE 5.23 An example of an inadequate system sequence diagram, where actors communicate directly with the objects.

Actors, interfaces, and other elements of the sequence diagram have a lifeline, represented by a vertical line, where events may happen. When the line is dashed, the actor or system is inactive. When the line is solid, the element is active (processing or waiting for the result of an operation being performed by another element). Human

actors are always considered active.

The horizontal arrows represent the information flow. There are three kinds of information flows in that diagram:

- *Between actors*: Actors' communication among themselves, corresponding to complementary steps of the expanded use case.
- *From the actors to the system*: Corresponding to *system events*, that is, steps that could be marked with [IN] in expanded use cases.
- *From the system to the actors*: Corresponding to *system returns*, that is, steps that could be marked with [OUT] in expanded use cases.

The information exchange between actors does not belong to the scope of the system, but it may be useful to illustrate how information is exchanged from actor to actor until reaching the system.

One thing to keep in mind when these diagrams are being built is that *information is not created during the process; it is only transferred from actors to system and vice versa*. The actor usually has some information that must be passed to the system in order to perform a process. To process a book order, the customer must inform the system who she is and which books she desires. In the beginning, the customer is the only one who holds that information; the system has the registry of all customers and books, but it cannot know who that specific customer is until someone provides that information.

The sequence diagram can be built for the main flow of a use case and completed with alternate flows. Optionally, different sequence diagrams could be built for each flow. However, the most important thing at the moment is to know *what* information is exchanged between the actors and the system. The analyst must build a catalog with all system commands and queries (which are explained in the next sections) identified in the main and alternate flows of the use case. This information will be used later to define contracts that indicate how the system retrieves and transforms information ([Chapter 8](#)).

5.8.2 Expanded use cases as system sequence diagrams

The system sequence diagram is a tool for achieving more formal and detailed use case descriptions. Additionally, developing the system sequence diagram makes the connection between requirements analysis (the expanded use case) and software design (the system operations that will be implemented).

The construction of the system sequence diagram may be done in two stages:

1. Representing the steps of the use case as *information* exchange between *actors* and the system *interface*.
2. Representing system operations as *methods calling* between the *interface* and the *façade-controller*,⁷ which encapsulates the domain tier of the system.

The first step is simple: for each use case input (a step that could be marked with [IN]), there is an equivalent system event in which one actor sends information to the interface, and for each use case output (a step that could be marked with [OUT]), there is an equivalent system return in which an actor receives information from the system ([Figures 5.24](#) and [5.25](#)).

1. The customer provides keywords for searching books.
2. The system generates a list of books for sale that matches the keywords, including at least the title, author, price, page count, publisher, ISBN, and cover image.
3. The customer selects books from the list and provides the desired quantity for each one.
4. The system generates the order summary (title, author, quantity, unit price, and subtotal for each book) and total value.
5. The customer finishes the order.

Variant 5d: The customer decides to continue shopping.

5d.1. Return to step 1.

FIGURE 5.24 A reference use case.

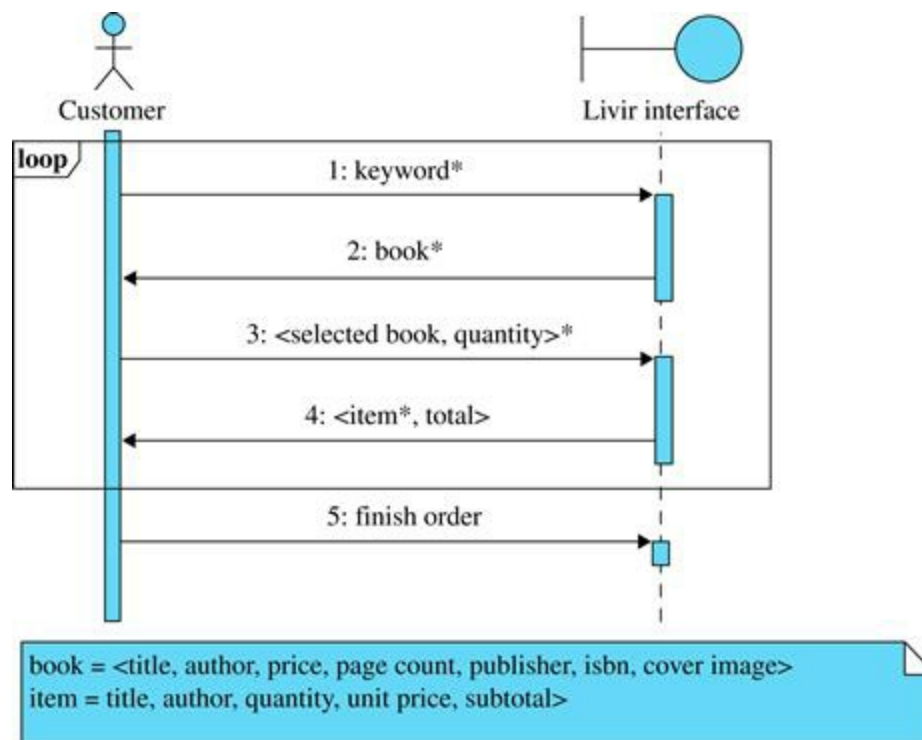


FIGURE 5.25 The representation of a use case main flow as a system sequence diagram.

Note that between actors and interface the flows consist of sending and receiving information. Information may be sent by an actor, for example, by typing it in a form. Information can be received by an actor when it is printed on the screen, for example. At this level, flows are not calling methods because actors are not internal objects.

The words or phrases used in sequence diagram flows to represent information are sometimes a source of confusion and misunderstanding. To avoid such confusion, a labeling pattern is recommended:

- Simple information (alphanumeric data) is represented by one or more words. For

- example, *keyword* (step 1 of Figure 5.25), *quantity* (step 3), and *total* (step 5).
- Complex information may be represented between “<” and “>” in the diagram if it is composed of a few elements (2 or 3). For example, <*selected book, quantity*> (step 3).
- Complex information composed of numerous elements (more than 3) may be represented in a separate note or data dictionary.⁸ Only the name of the complex concept is used in the diagram. For example, *book* (step 2) and *item* (step 4).
- A collection of values is indicated by the suffix “*”. For example, *keyword** (step 1), *book** (step 2), <*selected book, quantity*>* (step 3), and *item** (step 4).
- When one object is selected from a list presented by the system, the selection is indicated by the word “selected,” as in step 3, where “*selected book*” stands for a book that was selected from the list *book* presented in step 2.

Some other considerations on the diagram may follow:

- In step 1, the team could choose to use *keywords* instead of *keyword**. However, in this case, the fact that the information being sent is a list of words would not follow the proposed notation.
- The information contained in a *book* is probably a subset of the attributes of the *Book* class of the conceptual model; it does not necessarily correspond to the complete set of attributes of the *Book* class.
- In step 3, when the user informs the system about the list of books she has selected it is not necessarily the complete information about a *book* that is being sent by the user to the system. Technologically speaking, the user just has to select a book from a list and the interface sends to the system the correct identifier for the selected book (the user does not even have to know which identifier it is).
- In step 4, an item was created to represent different aspects of a book: instead of *page count*, *publisher*, *ISBN*, and *cover image*, an item presents *quantity* (which is not an attribute of a book), and *subtotal* (which is calculated from price and quantity).
- Step 5 does not represent alphanumeric information being sent or received as in the previous steps. *Finish order* is a control step. It just informs the system that the state of the order has changed without passing any other new alphanumeric information.
- Variant 5d is a choice between finishing the order or resuming shopping. It is represented by the *loop fragment* that includes steps 1 to 4.

5.8.3 Connecting the interface to the façade-controller

System events are actions that a user performs over the system interface. When a web interface is used, for example, these actions consist of filling forms, pressing buttons, etc. These are not operations in the sense of programming languages, that is, these flows from an actor to the system and vice versa do not represent method calling.

However, *system commands* and *system queries* are computational procedures that are necessary to implement system events and system returns, respectively. System commands and queries are activated by messages that are sent from the interface to the façade-controller. Now, we have a system component that invokes another system component. In this case, it is the interface that sends a message, asking the façade-

controller to perform a method that consists of a command or query. The set of all possible messages is equivalent to the whole functionality of the system: all access and updating of the logic for data is done by system commands and queries.

Thus, four kinds of arrows are of interest in system sequence diagrams:

- *System event*: As shown in [Figure 5.26](#), this is an action performed by an actor that sends some information to the system. In the diagram, it is represented by an arrow from the actor to the interface.
- *System return*: As shown in [Figure 5.27](#), this is a flow of information from the system to one actor, represented in the diagram as an arrow from the interface to the actor.
- *System command*: As shown in [Figure 5.28](#), this is a message that is sent from the interface to the controller, usually in response to a system event. The system command must, by definition, change some information stored or managed by the system. In the diagram, this is represented by an arrow from the interface to the controller labeled with a message (the use of parentheses is always recommended to distinguish it from information flows, such as system events and returns).
- *System query*: As seen in [Figure 5.29](#), this is a message that is sent from the interface to the controller in order to obtain some information from the system. Queries must not change data; they just return data. In the diagram, queries are represented by arrows from the interface to the controller labeled with a message with an explicit return value.

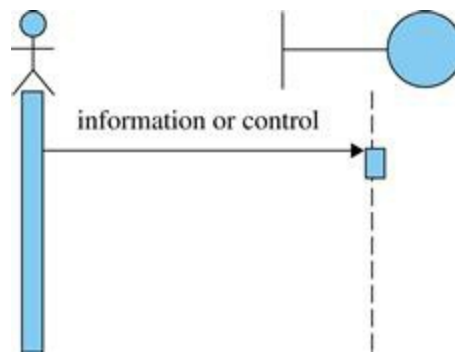


FIGURE 5.26 System event.

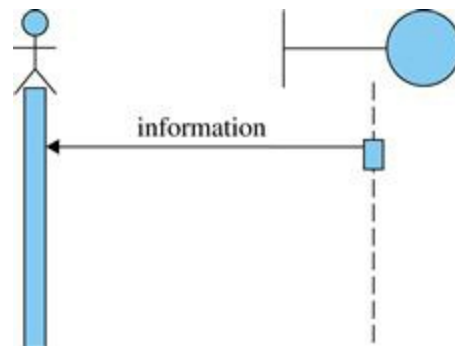


FIGURE 5.27 System return.

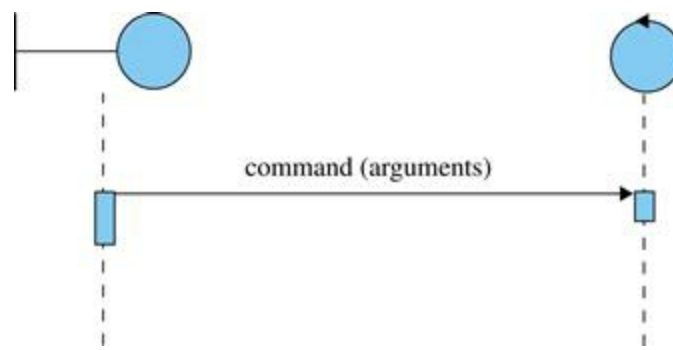


FIGURE 5.28 System command.

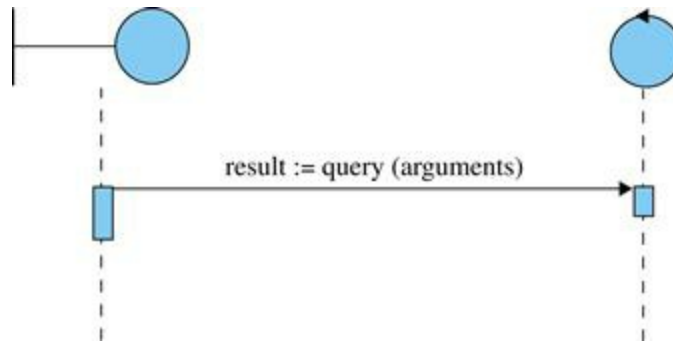


FIGURE 5.29 System query.

Query results could also be represented by dashed lines from the controller to the interface, as shown in [Figure 5.30](#). Both representations are allowed and the team may choose one or the other. The approach of [Figure 5.29](#) creates fewer arrows in the diagram, but may be complicated if complex data must be returned by the query. That problem can be minimized if notes such as those of [Figure 5.25](#) are used to define complex data. Other analysts would prefer the approach of [Figure 5.30](#), which makes the return appear explicitly as an arrow in the diagram. The disadvantage of this approach is that it splits a single query call into two arrows.

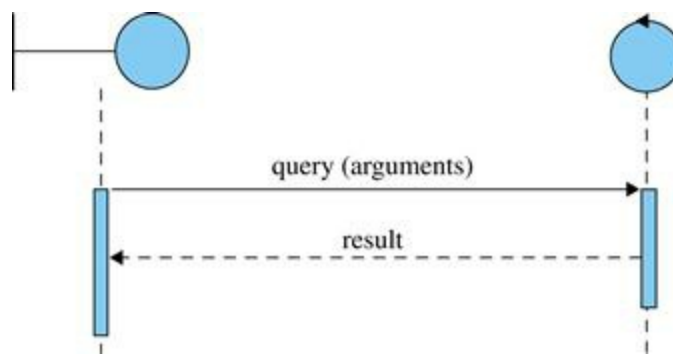


FIGURE 5.30 Alternate representation of a system query.

Communication between actors can be represented in the system sequence diagram as arrows from one actor to another. However, as explained before, these communications

do not affect the implementation of the system.

The domain tier of an application (the part of the system that contains all classes that perform the logical operations on data) is encapsulated by its façade-controller, which is an instance of a class that implements all system commands and queries that must be accessed by a given interface or set of interfaces. There are at least three subpatterns for the façade-controller. The first one, for smaller systems, consists of implementing a single controller for the whole system. However, this would result in a large number of operations implemented in a single class. A solution to this is to divide it into *use case controllers*, that is, a different controller class for each use case. However, this is not a good solution when different use cases call the same methods, which should be implemented in different classes. The most adequate solution for most medium or large systems is to implement *component* or *subsystem controllers*, that is, controllers that encapsulate the functionality of subsystems or components as defined by the system architecture.

The design of messages between the interface and the controller is made after an examination of the system events and returns using the following rules:

- A system event that sends data to the system that must be stored, updated, or deleted, that is, data that causes a change in the internal state of the system, requires at least one system command (Figure 5.31).
- A system return that sends data from the system to the user requires at least one system query, so that the data can be obtained from the controller in order to be presented by the interface (Figure 5.32).
- A sequence of system event and return, where the event just provides arguments for producing the return, requires a system query where the information received from the system event are the arguments (Figure 5.33).

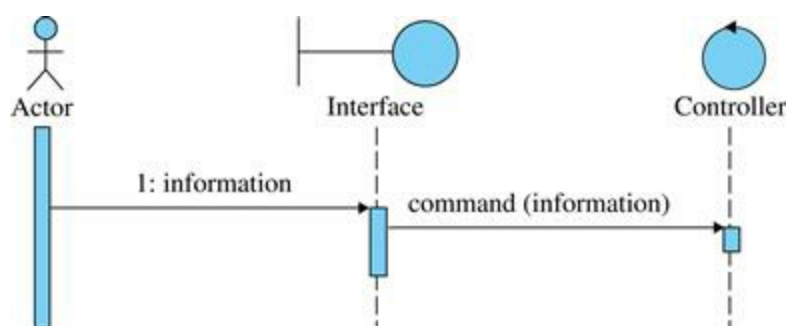


FIGURE 5.31 A system event that requires a system command.

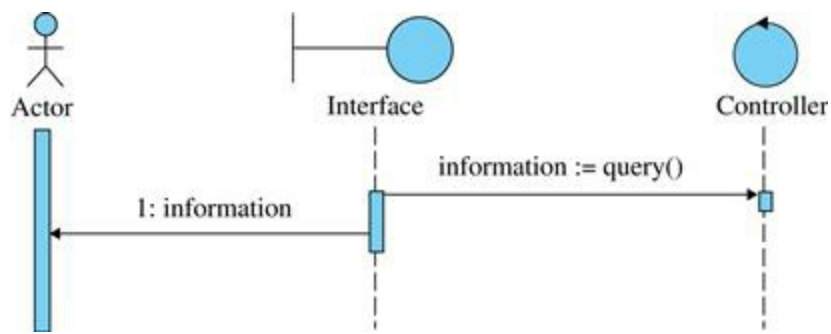


FIGURE 5.32 A system return that requires a system query.

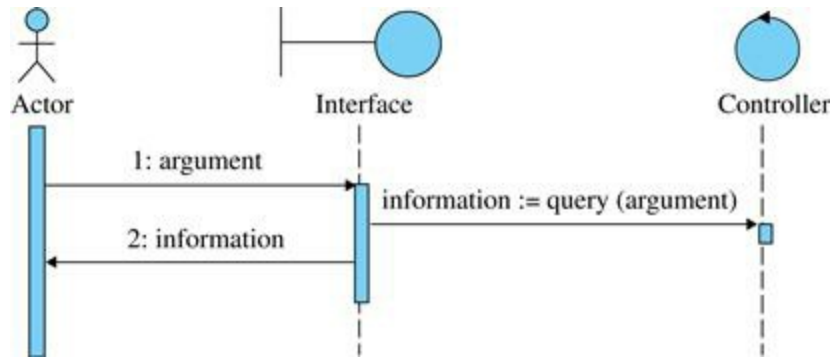


FIGURE 5.33 A sequence of system event and system return that requires a system query as the event just passes its arguments.

The cause/consequence relation between events and commands and between returns and queries is not always so direct. Sometimes, a query can use arguments that were passed to the interface many steps earlier, or information passed by the actor to the interface can be used many times by different system commands and queries.

These rules for deriving queries and commands from returns and events are just a first approach for designing the interface tier. Later, modeling techniques and design decisions may change the way these commands and queries are invoked.

5.8.4 Stateless strategy

When the system sequence diagram is being developed, each piece of information is passed from the actor to the system once. However, in the next level, between interface and controller, many different commands and queries may need the same arguments. For example, the customer *ID* may be necessary to perform many subsequent operations. At this point, the designer must decide if the controller must have transient memory for storing these arguments (*stateful strategy*) or if it is not provided with memory (*stateless strategy*). In the stateless strategy each time a system query or command needs an argument it has to receive it explicitly from the interface.

Figure 5.34 shows how the system sequence diagram would look for the use case *Order books* with a stateless strategy. The information is passed from the actor to the interface just once, but each time a system command or query needs that information, the

interface must send it again as an argument to the controller. Observe that *cartId* is passed by the interface to the controller three times: when *add2Cart*, *getCartSummary*, and *finishOrder* are called.

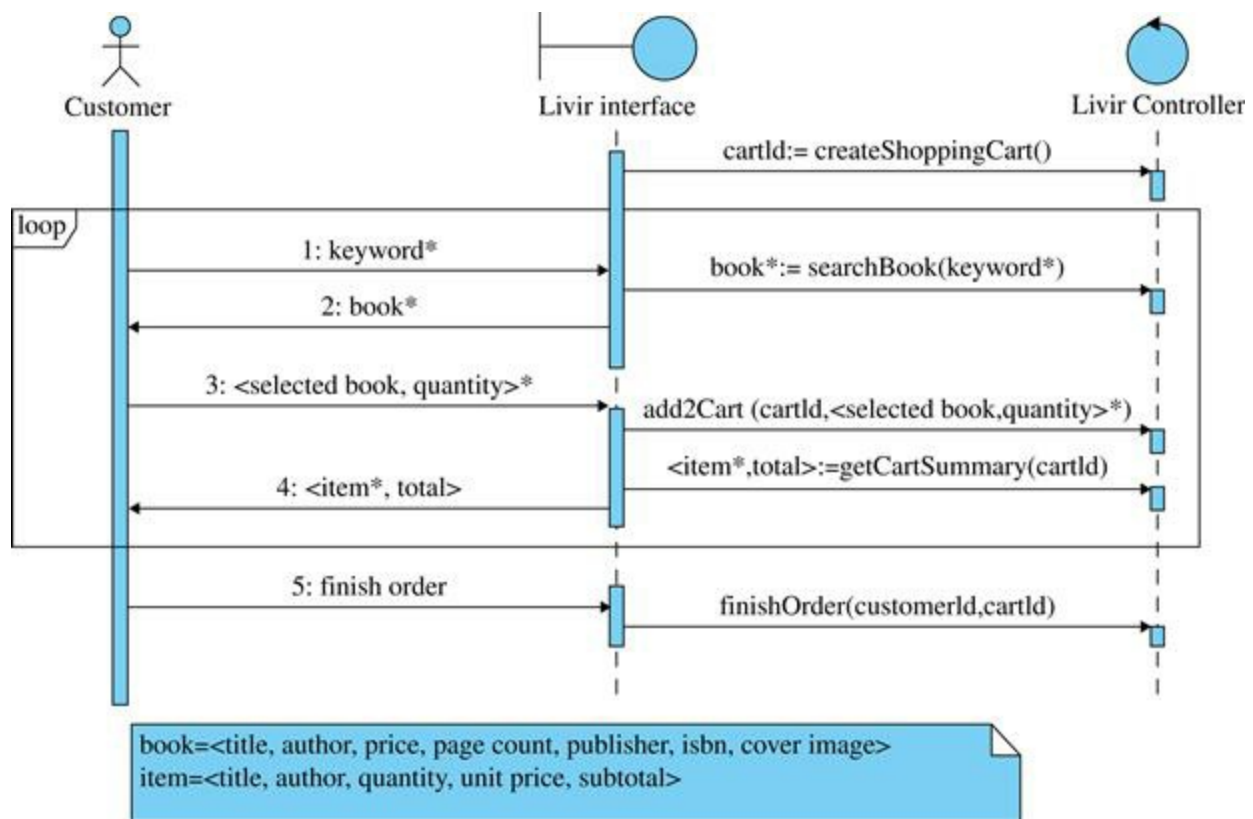


FIGURE 5.34 System sequence diagram for the main flow of use case *Order books* using a stateless strategy.

From this point it can be seen that the team is making a transition from analysis to design, because design decisions must be made in order to accommodate the requirements. For example, the fact that a user must be identified only when the order is finished prevents the system from creating an order before that point (if the order is really a concept that must be associated to a customer). Therefore, a shopping cart is created instead of an order at the initialization of that sequence diagram. The order will be created only when the user chooses to stop shopping (finishing the order). At that point, if the user is not already identified an exception would be raised, as explained later. One point to notice is that as the controller has no transient memory, it does not register the information of who is shopping. It is the interface that holds the customer identification at this point. When the order is ready to be completed, the interface sends to the controller the *IDs* of the customer and of the shopping cart (remember that the shopping cart is not yet associated to any customer).

As the controller does not have state (transient memory), each operation sent to it that refers to the shopping cart must refer to its number (*cartId*). Why is the *cart ID* used and not the cart object itself? Are we not doing object-oriented design? The point is that the domain objects are encapsulated by the controller. This means that the interface should not have direct access to the domain objects. This is why the interface sends an *ID* as a parameter instead of the *cart* object itself.

One remark must be made about the command *createShoppingCart*: as it creates a new domain object and returns its identification code, one could ask if this is not against the command-query separation principle. It is not. It is one of the acceptable exceptions to that principle, because the command just creates a new object and returns its identification code. It can be made because trying to obtain the code of a newly created object by other means would be counterproductive.

5.8.5 Stateful strategy

When the controller may have transient memory for holding working parameters during the execution of a use case, then we are using the *stateful* strategy. Thus, after an argument is sent to the controller by an operation, the subsequent operations would not need to receive the argument again; the controller would keep it in local memory.

This argument is not necessarily persistent information, though; the transient information is not stored in a database or other equivalent structure. It consists of transient information only (for example, who is the current customer, which shopping cart is being used, etc.), which is available only during the use case transaction. [Figure 5.35](#) presents the same use case that was shown in [Figure 5.34](#), but this time with the stateful approach. It can be observed that, among other things, it is not necessary to return the new shopping cart identification, because the controller will remember which one it is.

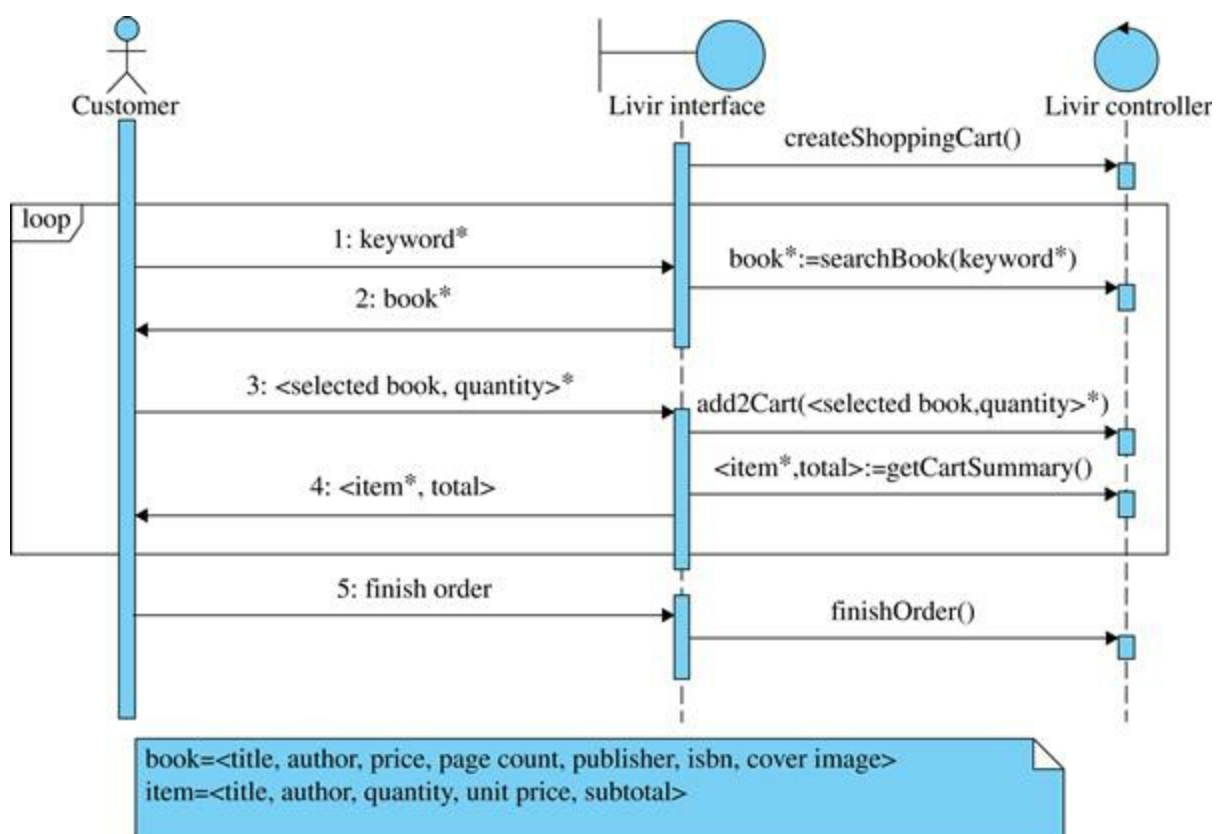


FIGURE 5.35 System sequence diagram for the main flow of use case *Order books* using stateful strategy.

With the stateful strategy it is not necessary to return the reference to the new shopping cart because the controller has an internal mechanism to store that value. In the same way, the methods *add2Cart*, *getCartSummary*, and *finishOrder* also do not need to receive that argument. It is maintained by the controller. In the same way, the identity of the customer, if known, is maintained by the controller, not by the interface.

One option for implementing this transient memory is to use transient associations, as explained in [Section 8.3](#). It is an architectural decision to choose one or another approach. Some pros and cons of them are the following:

- The stateful strategy demands the implementation of a transient (not persistent) memory mechanism in order to store some parameters (the use of transient associations, for example). The stateless strategy does not demand that kind of mechanism.
- The stateless strategy demands more parameter passing between the interface and the controller. When it is the case of passing information through the network this may be inconvenient. With the stateful strategy information is transmitted just once.

Additionally, it can be said that operations are more reusable with the stateless strategy, because each of them receives all the parameters it needs, without depending on operations that were performed before. Also, the stateless strategy scales better than the stateful one. Systems with many concurrent users would be easier to produce and maintain with stateless strategy.

5.8.6 Alternate flows in system sequence diagrams

As seen at the beginning of this chapter, use case steps, especially system events, may have exceptions, which are handled by an alternate flow.

The insertion of those alternate flows in a system sequence diagram may be done in stages. Initially, the exceptions may be indicated in the diagram as well as where they occur. Then *opt* fragments can be produced, that is, optional parts as shown in [Figure 5.36](#). The procedure is to take the complete use case as shown in [Figure 5.7](#) and add each exception as an *opt* fragment at the adequate position of the flow (usually just after the step that could cause the exception).

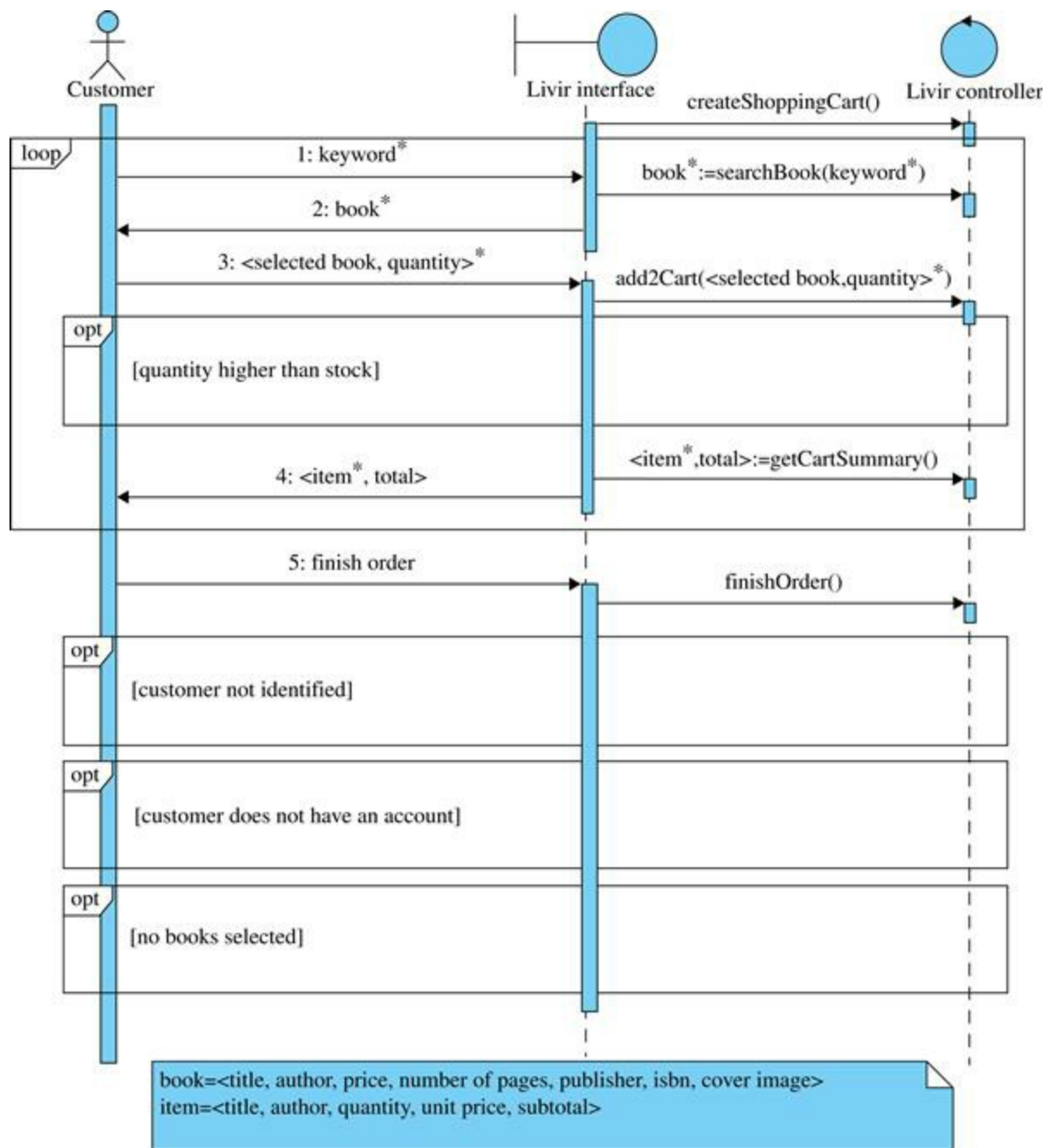


FIGURE 5.36 Sequence diagram with exceptions included.

The diagram of [Figure 5.36](#) shows literally that after step 3 and after step 5 optional events may happen. Those events correspond to exceptions 3a, 5a, 5b, and 6c shown in [Figure 5.7](#).

Now, it is necessary to design what happens inside the optional fragments when each exception is handled. The *opt* fragment contains the steps of the alternate flow, but it is also important to look carefully at the termination of the alternate flow. Some of the alternate flows return to the step that generated the exception and must be treated as repetitive loops that include the step which caused the exception. Others advance to the next step, and should be treated as just an optional fragment of the diagram. When other use cases are referenced or when a fragment is so long that it justifies its definition in a separate diagram, then the *ref* fragment may be used, which indicates that another sequence diagram is being called. The *ref* fragment may receive and return information in the same way a programming function does, and using the same notation, as for

example, *ref otherDiagram(arguments):result*.

Now the diagram of [Figure 5.36](#) is going to be improved with exception handling. This may become a complex task unless some design decisions are made carefully. For example, in step 3, a set of books and respective quantities is sent to the system and only then the quantities are checked. This would require handling an exception where some books are ordered above stock and others are not, which could be inconvenient. Therefore it was decided to change the sequence diagram so that the exception would be checked each time a quantity is provided. Notice that this decision changes a little the way the user perceives the system behavior: instead of complaining about quantities at the end of the selection, the system would check the quantity after each book choice. The resulting diagram is shown in [Figure 5.37](#), where only the first three steps of the use case are represented for simplicity.

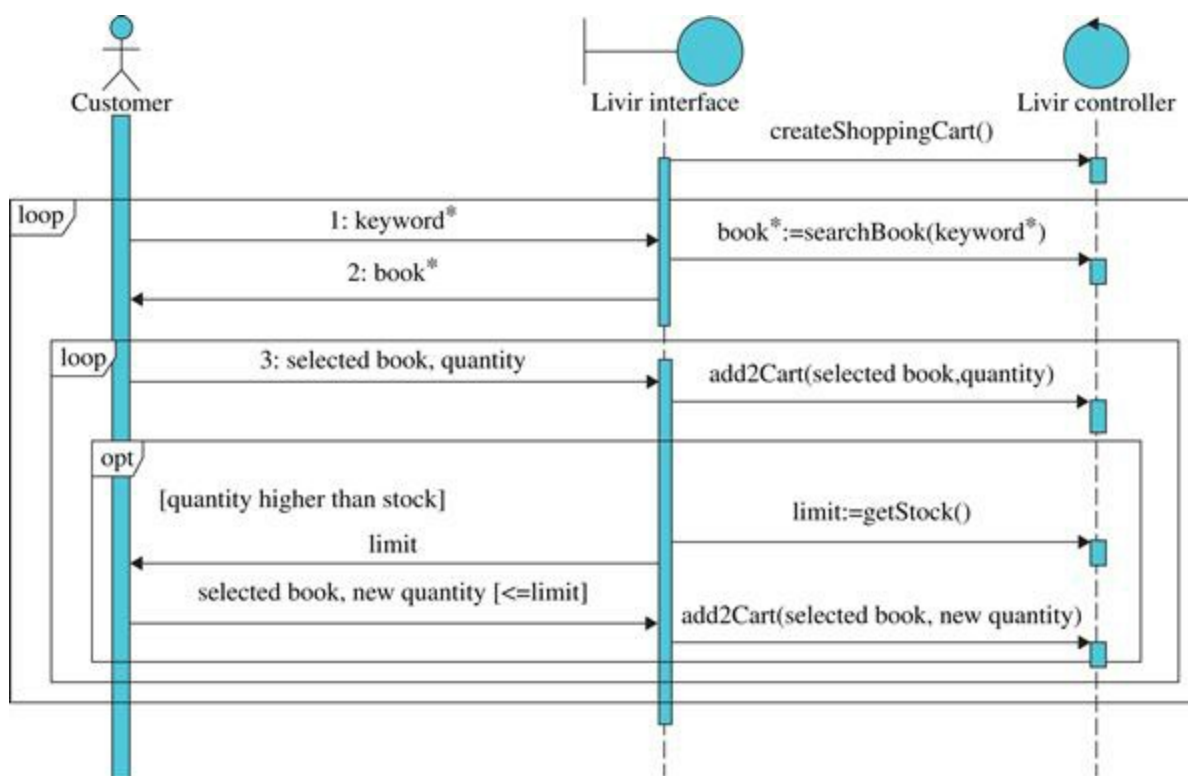


FIGURE 5.37 Part of system sequence diagram with a design change and an exception handler.

Note in [Figure 5.37](#) that the design could be improved again. Why try to perform a command with a quantity that is above stock if the system already knows that information? The information about stock could be returned by the *searchBook* query so that the interface entry field could already be limited to the real quantity available. That information could also be used by the interface to show books that are in the catalog but not in stock as *out of stock*. [Figure 5.38](#) shows the resulting design.

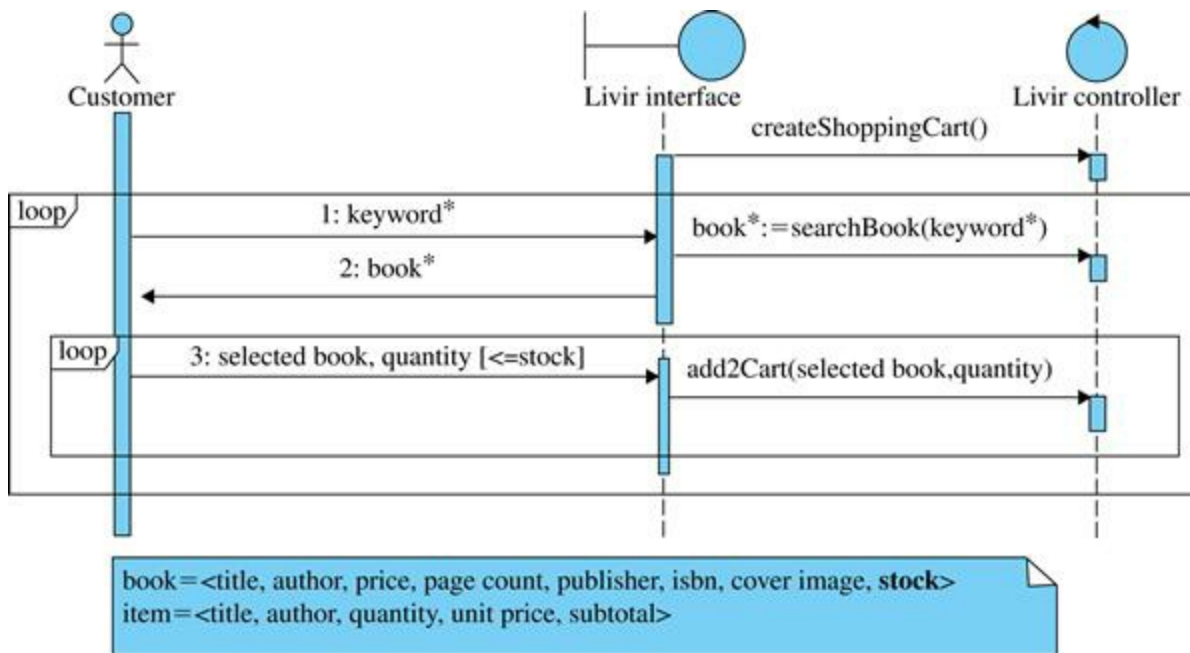


FIGURE 5.38 Part of the system sequence diagram where an exception handler was redesigned.

Observe that in [Figure 5.38](#) the fragment that handles the exception simply disappears. This happened because the exception was changed into a precondition for the system command *add2Cart*. This is a popular technique to improve design that will be revisited in [Chapter 8](#). The technique consists of preventing the user from sending an invalid argument; the user choice is limited to the valid values. Thus, the exception cannot occur anymore.

Exceptions 5a and 5b may be resolved together as they involve user identification. When the command *finishOrder* is executed, an exception would occur if the user is not yet identified. The user then has two choices: entering a valid identification or creating a new one. The partial diagram is shown in [Figure 5.39](#).

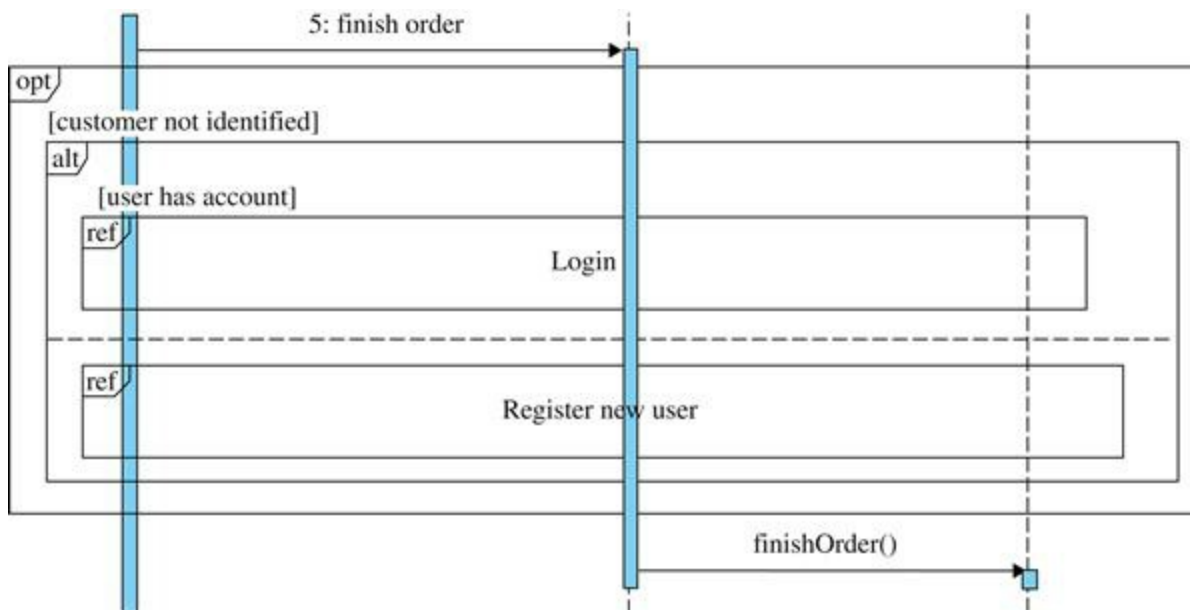


FIGURE 5.39 Dealing with user identification.

As seen in [Figure 5.39](#), it is not necessary to try the command *finishOrder* in order to discover that the user is not yet identified. The condition may be tested after the user decides to finish the order but before trying the command. Thus, if the user is not identified, she has two options, represented inside the *alt* fragment. The *alt* fragment works like an *if-then-else*: if the user has an account, then she logs in, else she has to create a new account. In both cases, as those sequences may be highly reusable, a reference to another sequence diagram is indicated by the fragment *ref*.

Finally, exception 5c states that if no book was selected the order cannot be closed. This can be implemented as a loop fragment placed over the flows from 1 to 4 (excluding flow 5) that would repeat until at least one book is selected. As there is already such a loop in the diagram (see [Figure 5.37](#)), it is enough to add that condition to the existing loop. The complete sequence diagram with all exceptions handled is shown in [Figure 5.40](#).

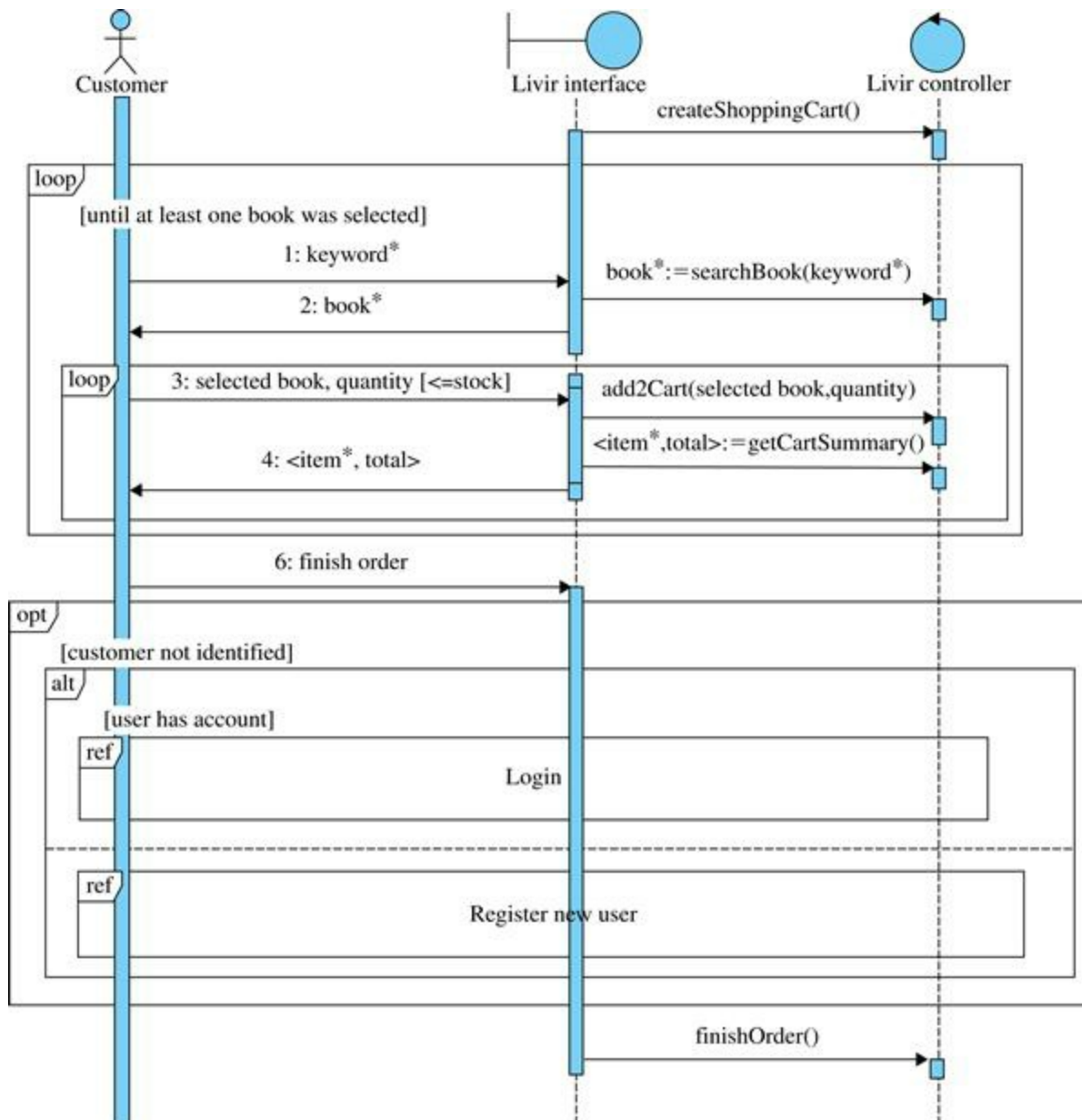


FIGURE 5.40 Complete sequence diagram with exceptions handled.

The main practical result of the system sequence diagram is the discovery and design of the system commands and queries that must be implemented to allow the functionality of the system to be available for the users. According to [Figure 5.40](#), these commands and queries are:⁹

- createShoppingCart().
- searchBook(keyword*):book*.
- add2Cart(selected book, quantity).
- getCartSummary ():<item*, total>.
- finishOrder().

[Chapter 8](#) shows how to use this information to build a set of contracts that define the whole functionality of the system.

5.9 The process so far

	Inception	Elaboration
Business Modeling	<p>Build a general view of the system:</p> <ul style="list-style-type: none"> • Build a business use case diagram and determine the automation scope for the project. • Build preliminary activity diagrams for business use cases. • Build preliminary state machine diagrams for key business objects. 	
Requirements	<p>Prepare the system use case diagram (functional requirements):</p> <ul style="list-style-type: none"> • Identify the system actors from the business use case model. • Identify the system use cases from the business use case model, and the activity and state machine diagrams from business modeling. <p>Identify nonfunctional requirements as use case annotations:</p> <ul style="list-style-type: none"> • Identify the main business rules associated to use cases. • Identify the main quality issues associated to use cases. <p>Identify supplementary requirements.</p>	<p>Detail requirements by expanding use cases:</p> <ul style="list-style-type: none"> • Identify the main flow. • Identify alternate flows: variants and exception handlers.
Analysis and Design	<p>Prepare the preliminary conceptual model by observing system use cases and the concepts needed by them.</p>	<p>Elaborate the system sequence diagrams:</p> <ul style="list-style-type: none"> • Represent the main flow of a use case as a system sequence diagram. • Represent the system commands and queries using stateful or stateless strategies. • Complete the system sequence diagrams with alternate flows.
Implementation Test Project Management	<p>Estimate total effort, ideal calendar time, and average team size for the project. Estimate the duration and quantity of iterations for each phase. Prepare the phase plan and iteration plan for the first iteration.</p>	

5.10 Questions

1. Variants and exception handlers are alternate flows for a use case. In which situations should one or the other be used?
2. What is the difference between an essential use case and a real use case? Why is the essential use case more interesting for requirements purposes?

3. Explain the differences between mandatory, complementary, and unsuitable use case steps.
4. Expand the main flow of use case 03, *Deliver order*, presented in [Figure 3.10](#).
5. Finalize the expansion of the use case of [Figure 5.2](#) by indicating its alternate flows.
6. Design the system sequence diagram for the main flow of the use case of [Figure 5.2](#), choosing one of the following strategies: stateless or stateful. After designing the diagram for the main flow, add the alternate flows identified in Question 5.

¹<http://blog.ivarjacobson.com/use-cases-%E2%80%93-why-successful-and-popular/>

²For example, one customer buys two books and another customer buys five books: both actions may be described simply as “customer buys books” because the number of books is irrelevant for understanding the process.

³However, this style for variants must not encourage the reader to do the same with *exceptions*. As explained in [Section 5.3.3](#), in the case of exceptions there is indeed a *normal flow* and an *alternate* one, which must be described outside the main flow; in the case of variants the idea is that they are only different ways to perform the main flow.

⁴The set {1, 2, 3, ...}.

⁵Here and in further examples in this book the *ID* provided by a customer is not a system-generated internal code, but the number of a personal document that belongs to the customer. In the United States the *social security number* could correspond to the *ID* number, while in other countries other numbers could be used.

⁶In the past the author ran a project in which a use case that seemed to be a simple CRUD at the beginning happened to be the most complex use case of the whole system in the end.

⁷A façade-controller provides “a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use” ([Gamma, Helm, Johnson, & Vlissides, 1995](#))

⁸Remember that it is not always the case that complex information exchanged with the system corresponds to the complete set of attributes of a concept of the conceptual model.

⁹The strategy applied in [Figure 5.40](#) is stateful. If the stateless strategy was used instead, the operations would be slightly different, because they would have more parameters.

CHAPTER 6

Conceptual Modeling

Fundamentals

This chapter presents the basic elements of conceptual modeling with the UML class diagram. There is a detailed discussion of the three main elements of a conceptual model: concepts, attributes, and associations. It covers best practices on how to use those elements so that models are kept clear and maintainable. The chapter also gradually presents OCL (Object Constraint Language), especially in order to define derived attributes and derived associations as well as class invariants, which improve significantly the quality of a conceptual model. Finally, the chapter discusses the organization of the conceptual model; inheritance is not the only structure than can be used: it is best suited for representing concepts that are structurally related, but associative roles and modal classes should also be used to represent different kinds of organization that are not adequately described by inheritance. The chapter also shows how to discover and refine conceptual elements in detailed use cases and system sequence diagrams, therefore making the connection between those analysis artifacts.

Keywords

Conceptual model; class diagram; attribute; association; inheritance; OCL; invariant

Key Topics in this Chapter

- Attributes
- Concepts
- Associations
- Collections
- Organization of the conceptual model: structural, associative, and temporal
- Invariants
- Iterative construction of the conceptual model

6.1 Introduction to conceptual modeling

Domain analysis consists of discovering and modeling information that has to be managed by the system. This means that the team should discover how the information has to be structured and transformed. It starts during Inception with the preliminary conceptual model and continues during Elaboration, when that model is refined and completed as requirements analysis is deepened with the expansion of use cases.

One can analyze two aspects of information: *static* (also called *structural*), which is studied in this chapter and [Chapter 7](#), and *functional*, studied in [Chapter 8](#). The static aspect may be represented in the conceptual model, and the functional aspect in the system operation contracts.

During analysis there is no *dynamic model* for the objects, because analysis takes into consideration only an external view of the system. The dynamic model, consisting of the collaborations among objects, is made during design ([Chapter 9](#)), because only at that time are internal aspects of the system addressed. Thus, the functional model of analysis indicates only what information enters and leaves the system, without saying how the information is transformed. Later, the dynamic model of design will show clearly how object collaborations could change information.

The conceptual model describes information that the system is going to manage. It is an artifact from the *problem* domain, not the *solution* domain. Therefore, the conceptual model should not be confused with the *Design Class Diagram* (DCD), which belongs to the software architecture ([Chapter 9](#)). The DCD, although initially derived from the conceptual model, belongs to the solution domain and therefore serves different purposes.

The conceptual model should not also be confused with the *data model* ([Chapter 13](#)), because the data model emphasizes stored data representation and organization, while the conceptual model aims at representing the comprehension of information by users, not its physical representation. Thus, a relational data model is just one of the possible physical representations of a more essential conceptual model.

An interesting way to comprehend the conceptual model is to imagine that the elements described by it correspond to information that initially exists only in the mind of the user, as represented in [Figure 6.1](#), and not in a physical store system.

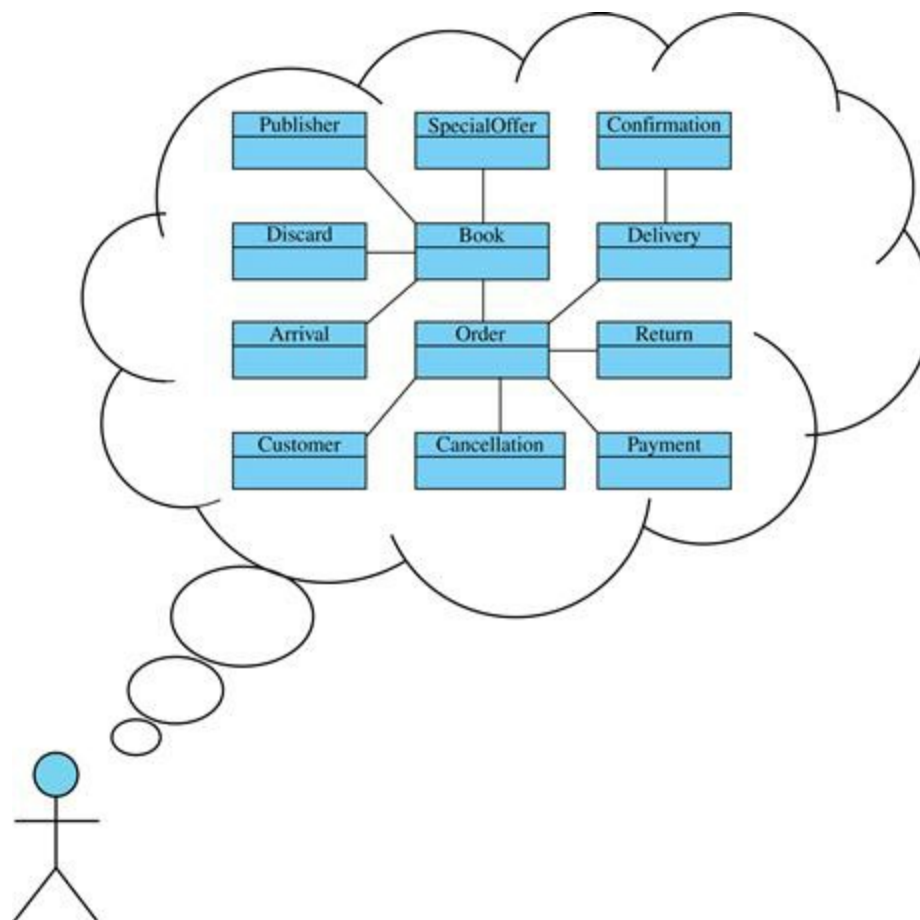


FIGURE 6.1 The conceptual model is a representation of the users' view of the information.

A user sends information to the system and receives information from the system through system events and returns, respectively. At this point, the system does not even need to be considered a *computational* system, because the information exists independently of computer support for storing and managing it.

The goal of the analysis is to study the problem. But the computational system is a solution, and therefore, it belongs to the design. The system solution could even be designed without computer technology. It is possible to analyze a whole situation and then propose a manual solution to implement it, in which, for example, data are stored in paper sheets, and operations performed by workers using pencil, eraser, and staplers.

Just like essential use cases, the conceptual model is independent of the physical solutions that could be adopted, and should contain only elements that belong to the domain of the problem, leaving to the design the elements of the solution, that is, all elements related to technology, such as interfaces, storage, communication, and so on.

The conceptual model represents only the static aspect of the information. Therefore, in the conceptual model, references to operations or dynamic aspects of the system cannot exist. Although the conceptual model is represented by the UML class diagram the analyst *should not yet include any methods*. [Chapter 9](#) shows how to identify and design methods using a systematic approach.

When the class diagram is used for conceptual modeling, there are precisely three kinds of elements to use for representing information:

- *Attributes*: Simple alphanumeric or primitive information, such as numbers, texts, dates, etc. Examples of attributes in the Livir project are customer name, payment date, book title, and total value of an order. An attribute is always connected to a more complex element: the concept.
- *Concepts*: The representation of the complex information that has coherent meaning in the domain. Concepts usually aggregate attributes and cannot be described merely as alphanumeric. Concepts also may be associated to each other. Examples of concepts in the Livir project are book, customer, order, and payment.
- *Associations*: A kind of information that links different concepts. However, associations are more than mere links: they *are* information. In the Livir example associations should exist between orders and customer, and also between orders and books, for example.

These three elements are detailed in the next sections. It is practically impossible to explain one without mentioning the others, because the three are strongly intertwined.

6.2 Attributes

Attributes are, in the conceptual model, the alphanumeric and primitive elements such as date, money, number, string, interval, etc.

Although most programming languages allow attributes to be defined as data structures such as lists, arrays, sets, trees, etc., it is not advisable in conceptual modeling because those structures are better modeled as associations, as explained in the

following sections.

Complex concepts (classes) should also not be used as attributes of other concepts (although, again, programming languages allow that). For example, a book should not be an attribute of an order. If a relationship between books and orders exists, then an *association* should be used instead, because books and orders are complex concepts. An exception to this recommendation occurs when a primitive type (Section 6.2.5) or enumeration (Section 6.2.4) is used to label an attribute; primitive types and enumerations are defined as stereotyped classes, but they do not behave like concepts. They are used as attribute types. For example, a date is composed by parts such as year, month, and day, however a birth date is considered an attribute of a person, not a complex concept associated to a person.

Attributes in UML are always represented inside a class, as shown in Figure 6.2, where the class *Customer* has the attributes: *name*, *id*, *address*, and *phone*.

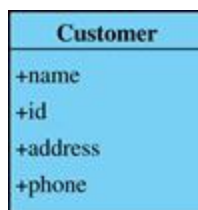


FIGURE 6.2 Attributes represented inside a class.

6.2.1 Attribute types

Attributes may have a type, although this is not mandatory for the conceptual model. Figure 6.3 presents a version of the class of Figure 6.2 with typed attributes.

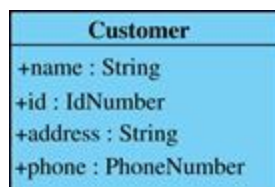


FIGURE 6.3 Typed attributes.

Types in conceptual models have the same meaning they have in programming languages. In Figure 6.3, we have the classical type *String*, and primitive user-defined types *IdNumber* and *PhoneNumber*.

When an attribute is defined by formation rules, as in the case of *id* and *phone*, it is advisable to define a primitive type especially for it, as is done in Figure 6.3.

The *address* is something special. Is it an attribute or a complex concept? Is it simply a string or complex concept composed of street, ZIP code, number, city, etc.? This case, as many others, is decided by analyzing the information needs of the users. If addresses are

used just to print envelopes and send mail, then they behave merely as strings, and may be represented as attributes typed as strings. However, if addresses are used to calculate distances and routes, or if they are used to group customers that live in the neighborhood, then they behave as complex concepts and should be modeled as a primitive type. Furthermore, if they have a structure of associations between them, then they should be modeled as complex concepts.

6.2.2 Initial values

An attribute may be declared with an initial value, that is, every time a new instance of the concept is created, that attribute will automatically receive a defined initial value, which can be changed later if necessary.

In the Livir system, an order may be created, for example, with a total value that is initially zero. This can be defined in the class diagram, as shown in [Figure 6.4](#).

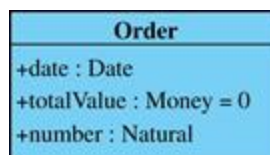


FIGURE 6.4 A class showing an attribute with an initial value.

The definition of an initial value for an attribute may also be created with the use of the *Object Constraint Language (OCL)* ([Object Management Group, 2010](#)).

Every OCL expression should be declared in a *context* that corresponds to a class. Most OCL declarations also have a *subcontext*, consisting of a class property, that is, an attribute, association role, or even a method. When declaring the initial value for the attribute *totalValue* in class *Order*, the context is *Order* and the subcontext is *totalValue*. Thus, the OCL expression starts as

```
Context Order::totalValue
```

Optionally, the type of the attribute may be added:

```
Context Order::totalValue:Money
```

In order to indicate that the OCL expression is defining an initial value for an attribute, the clause *init:* is used, followed by the expression that when evaluated produces the initial value for the attribute. As in the example the initial value is the constant 0, the expression may be defined as

```
Context Order::totalValue:Money
```

```
init: 0
```

The expression may also be simplified by omitting the type of the attribute:

```
Context Order::totalValue
```

init: 0

An attribute may have its initial value defined by more complex expressions. Later, examples of more complex OCL expressions are presented, which can be used to initialize attributes with values produced by mathematical and logical operations.

6.2.3 Derived attributes

Derived attributes are understood here as derived alphanumeric values only. If a “derived attribute” refers to objects or data structures, then they are called *derived associations* (Section 6.4.3). Derived attributes differ from normal attributes because they cannot be changed directly. In other words, they are *read only*.

A derived attribute is calculated and must be defined by an expression. In the class diagram, a derived attribute may be represented by a slash (/) preceding the attribute name (and type), followed by “=” and the (OCL) expression that defines the attribute. Figure 6.5 shows an example in which *profit* is defined as the difference between *price* and *cost*.

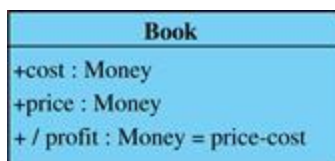


FIGURE 6.5 A class showing a derived attribute.

Derived attributes may also be defined by OCL expressions¹. The expression again has a class as context and an attribute as subcontext. It is followed by the word *derive*: this indicates that the expression that follows it defines how the value of the derived attribute is calculated. The expression of the example of Figure 6.5 defines *profit* by using the values of other two attributes of the same class: $profit = price - cost$.

For now, we can consider that the OCL expressions after clauses such as *init*: and *derive*: start with an expression that denotes an instance of the context class. That instance is referred to by the word *self*.

The notation *self.property* allows one to access the value of a property (attribute, association role, or method) belonging to the object. For example, in the context of the class *Book*, the expression *self.cost* denotes the value of the attribute *cost* of a given instance of *Book*.

Thus, the OCL expression that defines the derived value for the attribute *profit* of class *Book* is

```
Context Book::profit
derive:
    self.price-self.cost
```

In OCL it is possible to omit the expression *self* when the context is not ambiguous. In the example above, the expression could be simplified to

```
Context Book::profit
derive:
    price-cost
```

In the *Book* context, *price* and *cost* may be nothing else but attributes of *Book*. Then, the word *self* may be omitted from the expression.

Derived attributes may not be directly updated. In the *Book* class of [Figure 6.5](#) only the attributes *cost* and *price* may be directly updated. The derived attribute *profit* is read only: it is calculated as the result of its defining expression.²

6.2.4 Enumerations

Enumerations are a balance between concepts and attributes. They are basically *strings*, and behave like that. But there is a predefined set of *valid* strings that constitutes the enumeration domain. For example, a calendar day can only assume a value in a set of seven possibilities: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday. Thus, an attribute defined as a *CalendarDay* (typed with that enumeration) would only assume one out of those seven values.

Enumerations may appear in UML diagrams as stereotyped classes. It is suggested that enumerations that are domain independent are not placed in the same diagram that contains the conceptual model, but in a special package intended to contain the enumerations, as shown in [Figure 6.6](#), because domain-independent enumerations (such as *CalendarDay*) are very reusable from application to application. Only domain-specific enumerations should be kept together with the other classes of the conceptual model so that people can see their meaning more easily.

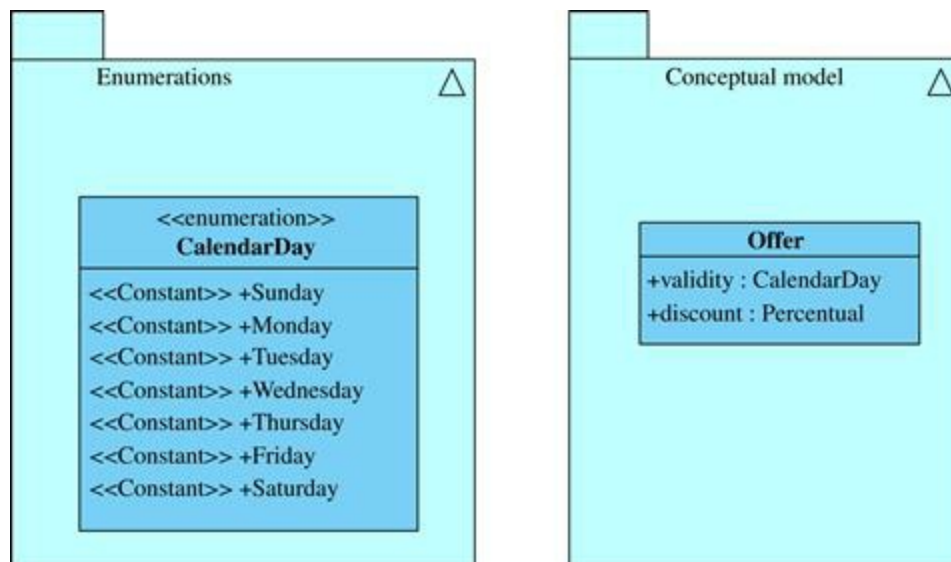


FIGURE 6.6 Definition and use of an enumeration.

In [Figure 6.6](#), the attribute *validity* of class *Offer* may have assigned to it one out of the seven possible values for the enumeration *CalendarDay*.

Enumerations may be considered stereotyped classes. While they may have associations and attributes of their own, the analyst is not encouraged to use those features because they could simply turn the enumeration into a complex concept and it would probably lose reusability. For example, consider the *CalendarDay* enumeration. It is simply a list of seven names. Suppose now that the analyst adds some attributes to the *CalendarDay* enumeration, such as, for example, the working hours. Then, for example, Monday to Friday may have working hours defined from 9 am to 5 pm, Saturday from 9 am to noon, and Sunday may have no working hours at all. Different applications could define working hours in a different way. In this case, what is the difference between this enumeration and a complex concept as it also has attributes and possibly associations and even methods later? Instead of creating attributes or associations for an enumeration it would be advisable to create a new complex concept that accommodates the enumeration and the other attributes. In the example, a new concept *working day* could be created with two attributes: *calendar day* (still an enumeration), and *working hours* (normal attributes as explained above). Thus, enumerations that are thought to incorporate extra functionalities should be maintained without that structure and included in normal classes.

To remain reusable, an enumeration should not be able to access another class through an association, but any other class should be able to access the enumeration. The fact that *Offer* in [Figure 6.6](#) has an attribute labeled with *CalendarDay* allows the normal class *Offer* to access the enumeration *CalendarDay* but not vice versa. This keeps the enumeration uncoupled from the classes that use it and therefore it is reusable.

Enumeration values in OCL may be represented by the enumeration name followed by “:” and the value name, as, for example, *CalendarDay::Tuesday*.

6.2.5 Primitive types

The team can and should define *primitive types* when they deal with attributes that have formation rules, as in the case of ISBN. Primitive types may be defined as class stereotyped with `<<primitive>>`, as shown in [Figure 6.7](#).



FIGURE 6.7 Example of a primitive type.

The *ISBN* class, which is stereotyped in such a manner, is a *type* that can be used to define attributes. It is not a complex concept like *Book*, *Customer*, etc. It has a value with public access (marked with “+”), and a validation predicate that is private (marked with

“-”), meaning that it can be accessed only by methods inside the class itself. That predicate can be called to verify if the ISBN value is valid or not, considering its formation rule.³ That predicate can be used in the constructor (the method that creates instances of the class) to guarantee that only valid ISBNs are created. Remember that conceptual classes do not have methods, but primitive types are not part of the conceptual model. They are user-defined, reusable low-level components.

Although the value of the primitive type has public access it is normally accepted that it is immutable. That means that the value of the primitive type can be defined only at creation time, and not updated later. This is to make it consistent with the idea of primitive data elements as *constants*, and not as *objects*. Objects may change their internal state, and constants usually do not.⁴

Some primitive types, such as *Date* and *Money*, are already available in some programming languages and database management systems. Other not-so-common types, such as ISBN, ZIP code, odd numbers, etc., may be created.

The primitive type formation rule must be defined syntactically, that is, by evaluating expressions that do not need to consult data managed by the system. If a formation rule depends on consulting attribute or association values, then it is not, probably, a primitive type, but a normal class. For example, the *prime number* formation rule may be checked without consulting objects and associations, but a *registered customer name* may not be a primitive type, because to check if a string is a registered customer name, it is necessary to query existing names of current customers.

A primitive type usually is *application independent*, that is, its meaning usually is independent from the system or project being developed. An ISBN has the same meaning and formation rule in any application domain that uses it. Concepts such as *Customer*, *Book*, and *Order* may have different definitions in different applications and cannot usually be reused without some refactoring effort. However, primitive types are usually 100% reusable, without adjustments or modifications.

6.3 Concepts

Concepts are more than alphanumeric values. They are also more than a bunch of attributes, because they carry meaning and may be associated to each other. Conceptual modeling may begin with concepts and associations only, as shown in [Section 3.6](#), that is, attributes are not needed right away. However, when examined in detail, a concept usually contains a coherent group of attributes.

6.3.1 Unique attributes

It is assumed that different instances of a concept are distinguishable from each other just by the fact that they were created independently. This is not the case with primitive types. Two dates, such as 05/03/2013 and 05/03/2013, are the same. But two customers named John Smith are not necessarily the same person. Some programming languages indeed use two comparison predicates, one to check if two objects are *equal* (they have

the same values for their attributes) and another to check if they are *identical* (they share the same space in memory).

Different instances are therefore always distinguishable from each other, but additionally they may have attributes that are *unique* to them. Once an attribute has been stereotyped as `<<unique>>` two instances of the concept cannot have the same value for that attribute. One example of a unique attribute is the ISBN for books, which is unique for each book (Figure 6.8), while the other attributes may have identical values for different books.

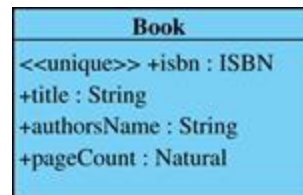


FIGURE 6.8 Class showing an attribute stereotyped as `<<unique>>`.

Unique attributes must not be confused with primary keys, `<<pk>>`. Primary keys are used especially for database design to provide a unique identification for an instance. But `<<unique>>` and `<<pk>>` are not identical, because a concept may have only one primary key, but more than one unique attribute. Unique attributes are closer to the idea of *candidate keys* in relational databases (Date, 1982).

6.3.2 System control class

Usually, the conceptual model is expected to be a connected graph, that is, every concept has a path (a sequence of associations) connecting it to every other concept. When that is not the case, it is supposed that something is missing, that is, important associations or concepts have not been discovered yet.

Although some authors oppose it, a concept that represents the system as a whole (in our example *Livir*) may be useful in the conceptual model if some precautions are taken. First of all, that concept should be declared as a *Singleton*,⁵ and second, it should not have attributes, being just a control class. This class corresponds to the *system controller* or *façade-controller* already mentioned in Chapter 5. It does not represent any data; it serves only as a point of reference to add associations to other classes, as seen in the following sections.

All concepts are associated directly or indirectly to the controller class in order to be accessible by the application.

The system controller class may be stereotyped with `<<control>>` or drawn using Jacobson's (1994) icon, as shown in Figure 6.9.



FIGURE 6.9 A system controller class.

In [Chapter 9](#), the advantages of using the controller class are explained in detail. There are some situations that are nicely modeled as associations from the control class.⁶

6.4 Associations

When complex concepts are related, it is said that there is an *association* among them. Associations usually happen only between two concepts, but they can also be defined among three or more concepts. There may also be reflexive associations between instances of the same class. For example, the association that links parents to children is defined as a reflexive association from *Person* to *Person*.

When classes are associated, then their instances may be *linked*. For example, an order is linked to the books being ordered, and also to the customer that creates the order. A payment, if it exists, is linked to an order in the bookstore example.

Associations may be tricky to identify in use case texts, especially because sometimes deciding if something is a concept or association may be fuzzy. For example, is a *book reservation* an association between a book and a customer? Or is it a concept? If it is considered a concept, what are its own associations?

Moreover, use case texts often mention *operations* that are not exactly static associations. Operations such as buying books, or making a reservation, are dynamic transformations on data. They could produce concepts to represent their existence, but usually representing them as associations would not be adequate. It is necessary to take into account the difference between the (static) associations and (dynamic) operations:

- An *association* is a static relation that may exist between complex concepts, complementing the information about them in a given moment (a snapshot of their state), or referring to new associative information (for example, neighborhood or parent/children).
- An *operation* is the act of accessing and transforming existing information.

As a description of a dynamic process, a use case text is usually full of explicit references to operations, but associations usually have to be inferred from the text.

At a used car dealer there are people who buy cars from other people. When someone buys a car she is performing an operation. That operation changes the links between objects: one ownership link ceases to exist, while another is created in its place. It would not be appropriate to represent an operation as an association, as shown in [Figure 6.10](#).

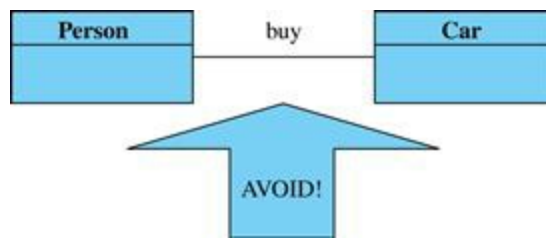


FIGURE 6.10 An operation inadequately labeled as an association.

Why is that interpretation not adequate? Because *to buy* is not a state of the information, it is a transition between states.

The state of the information about cars and people in this case is the ownership: a given person owns a car or not. This can be represented as an association between *Person* and *Car* (Figure 6.11).



FIGURE 6.11 Representation of a static association between two concepts.

However, there are different ways people and cars may be associated besides ownership. A person may be *passenger* of a car, or its *driver*, or the association may represent merely that someone *likes* a given car. To eliminate such ambiguities, it is convenient in many cases to use a *role name* on one or both association ends in order to indicate which role a class plays. Role names are not assigned to the association as a whole, but to both of its sides. Thus, a binary association has two role names: one for each participating class. In Figure 6.12, for example, a person is in the role of *owner* related to a car and a car belongs to the role *fleet* of a person.



FIGURE 6.12 An association with role names.

Different roles may be represented by different associations, as shown in Figure 6.13.

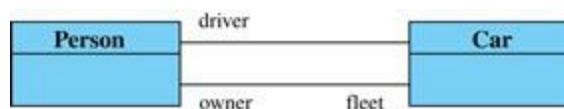


FIGURE 6.13 Multiple associations between the same concepts.

When an explicit role name is missing in the diagram, it is assumed that the class name (not capitalized) is the default role name. In the case of Figure 6.13, a car has two

roles related to people: *owner* and *driver*. From the point of view of a person, there are two roles for cars: the *fleet*, which corresponds to the cars a given person owns, and a default role name *car*, which corresponds to the car a person drives.

According to the UML specification, associations can also have names (Figure 6.11 presents an example), which are placed in the middle of the edge that links the classes. However, such names, besides being difficult to give (novice analysts use to fill their diagrams with lots of associations labeled with “has” or synonyms), do not have much practical use.

It may be easier and more useful to work with role names instead of association names. In practice, one can ignore the fact that associations may have a name. Conceptual models can be perfectly clear with role names replacing association names. Role names are easier to give because they define navigation possibilities between concepts and are useful also for naming derived programming code elements such as the variables that would represent the association when the code is generated. If by any means an association name is still needed it is always possible to create it by composing the role names; the reverse is not true. For example, an association between a book and a customer could be called *book2customer* if it indeed needs to have a name (for example, for reference purposes).

Sometimes, it may be interesting to keep information about operations or transactions that were performed. For example, it would be useful for the car sales system to store information about sales transactions, as for example, date, value, etc. In this case, the information about the transaction is represented in the diagram, as a complex concept (Figure 6.14), and it represents statically the operation that was performed.

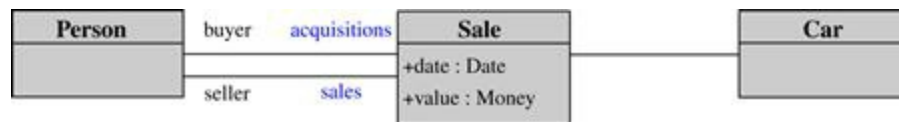


FIGURE 6.14 A transaction represented as a concept.

In this case, the transaction is represented as a concept, while its associations represent static relations between concepts.

6.4.1 Role multiplicity

In a conceptual model, it is fundamental to know how many elements a role accepts. For example, in the association between *Person* and *Car* with role name *driver* in Figure 6.13, how many cars can someone drive? How many drivers can a car have?

The answer depends on a study of the nature of the problem and on the real meaning of the association, especially if it represents the present time or the past. For example, if the association of Figure 6.13 represents the present, then the car may have just one driver (one at a time). But if the association represents the past, then it is possible that a car has had a number of drivers who drove it during different periods of time.

Thus, it is fundamental that the team decides clearly what the association means before deciding on its role multiplicity. They have to remember that associations are static constraints; they represent whatever can exist between the instances of the associated classes. Therefore, depending on the meaning of the association, it may have different role multiplicities.

There are, basically, two decisions to be made about the multiplicity of an association role:

1. Is the role mandatory or not? For example, should a person have at least one car?
Should a car have at least one driver or owner?
2. Does the number of instances that may be linked through the role have a defined conceptual bound? For example, is there a maximum number or minimum number of cars that a person may drive or own?

The answers to these questions may be tricky. Regarding the first decision, for example, returning to the Livir project, it is expected that each order must have a payment. But that does not qualify the payment role for an order as mandatory, because an order can exist without a payment for a period of time. Eventually, any order will hopefully be paid for, but not necessarily soon. Thus, this role is *not* mandatory for the order.

The other tricky issue concerns the maximum number of elements a role allows. Physically speaking, the maximum number of cars that a person may own is the number of cars that exist on planet Earth (the number is increasing, but it is defined at a given moment of time, even if it is hard to count). But as new cars are built, they may be purchased. Thus, although there is a physical limit for the ownership role, there is no conceptual or logical limit. Thus, the role should be considered virtually without upper bound.

Role multiplicity in UML is represented by a numeric expression where

- The asterisk "*" means "any quantity," and indicates that there is no upper bound.
- The comma "," means "or."
- Two dots ".." means "up to."

The following are examples of the usual multiplicity limits for conceptual models:

1 – exactly one

0..1 – zero or one

* – zero or more

1..* – one or more

2..5 – two up to five

2,5 – two or five

2,5..8 – two or from five up to eight

The multiplicity limits that include zero represent roles that are optional for a concept. All the other limits are mandatory.

Figure 6.15 shows an example where one person may have a fleet with an undetermined number of cars (optional), and a person may be a driver of one car at most (also optional). On the other hand, it shows that a car must have a single owner (mandatory) and that it may have a driver or not (optional).

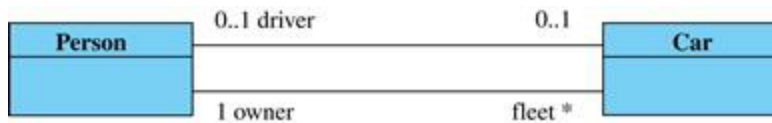


FIGURE 6.15 Example of associations with role multiplicity.

According to many countries' laws, a car may have more than one owner, or it may be owned by a corporation; in some situations a car may also have no owner at all. The conceptual model (such as the one in Figure 6.15) represents not necessarily a general truth, but the information needs for a given application. For some information systems it could be mandatory that a car, in order to be registered, have a single owner and that this owner must be a person. In this case, the conceptual model should represent the specific and pragmatic information needs for that system.

6.4.2 Association direction

An association in a conceptual model should be *nondirectional*, that is, the navigability of the association should not yet be determined (Booch et al., 2007). This is due to the fact that for analysis purposes, it is only necessary to know whether two concepts are associated or not; dynamic design will decide on navigability later.

There is, usually, little benefit to a premature (before dynamic design) definition of navigability direction. *Unidirectional* associations in general have the following advantages over bidirectional ones:

- It is not necessary to define a role name for both sides (just one).
- They are easier to implement.

Only the first advantage has any meaning for conceptual modeling, but it is too weak to justify making decisions prematurely.

6.4.3 Derived association

Just as useful as derived attributes are *derived associations*, that is, associations that instead of being represented physically are calculated from available information. For example, suppose it is often necessary to know which books a given customer has already bought in the Livir system. As books are not linked directly to the customer in the conceptual model, referring to that information could be a bit more complicated than expected. Figure 6.16 shows an example of how a conceptual model for Livir could relate books and customers indirectly through orders and purchasing items; in this case, the set of all books bought by a customer would be referred as *the set of books linked to the items linked to the orders linked to the customer*.

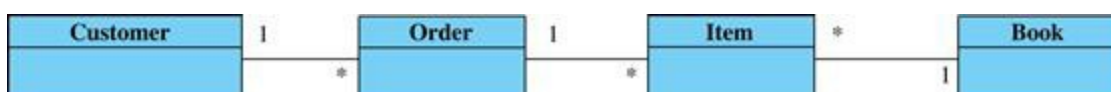


FIGURE 6.16 A customer indirectly associated to books.

Compared to the preliminary conceptual model of [Chapter 3, Figure 6.16](#) introduces the concept *Item* between *Book* and *Order*. Items are necessary if the concept *Book* means the published title, not the physical copy. In this case, what is ordered is an *Item*, which represents one or more physical copies of a given *Book*. The book may have its price updated regularly, but a change in the book price should not affect the price of former orders. Thus, the price of an item remains the same even if the price of the book was changed.

Note that with this model there is no direct link between the customer and the books she has already bought. Creating a new straight association between *Customer* and *Book* would not be a good solution because it would be redundant to the model, as the set of books may already be obtained indirectly. Furthermore, a new association between *Customer* and *Book* could associate *any* customer with *any* book, not only those that were already bought by the customer through her orders. This model allows the representation of inconsistent information, which would require some checking mechanism (for example allowing only books that were bought to be associated to the customer).

A modeling solution for that case, when it is relevant to have straight access to a collection of objects that can already be derived from existing information, is to use a *derived association*, as depicted in [Figure 6.17](#).

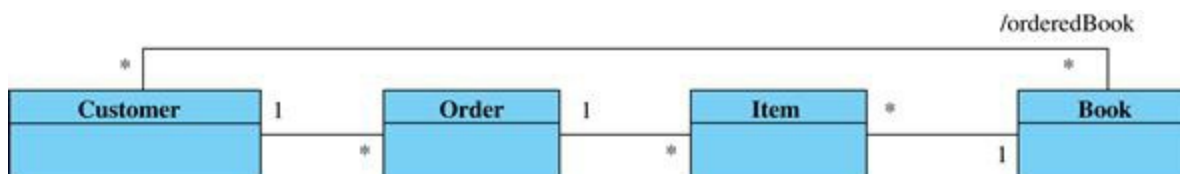


FIGURE 6.17 A derived association.

UML allows derived associations to be navigated in both directions. It is assumed that both roles are derived. In the case of [Figure 6.17](#) the opposite role *Customer* must also be considered derived, even if its default role name does not indicate this in the diagram.

The multiplicity of both roles of the derived association of [Figure 6.17](#) is multiple because by navigating on the normal associations it can be seen that a customer may have ordered zero or more books, and that a book may have been ordered by zero or more customers. Usually, the multiplicity of a derived association is defined by the product of the multiplicities of the roles that must be navigated to reach the target concept. The following rules may be used to determine that multiplicity:

- The lower bound of the derived multiplicity is the product of the lower bounds of the roles in the navigation path.
- The upper bound of the derived multiplicity is the product of the upper bounds of the roles in the navigation path.

For example, if the multiplicities found in the path are 1..5, 2..* and 1, then the derived multiplicity is 2..*, because the lower bounds are 1, 2, and 1 ($1 * 2 * 1 = 2$), and the upper

bounds are 5, *infinite*, and 1 ($5 * infinite * 1 = infinite$).

However, if filters are applied to the definition of the derived association, the resulting multiplicity may be different. For example, if the derived association is defined so that it links a customer to the most expensive book she has already bought, then the derived multiplicity is 1, and not *.

As opposed to normal associations that allow links between individual objects to be added and removed, a derived association may only be consulted (similar to derived attributes, which are read only).

A derived association may be defined by an OCL expression as well. The example of [Figure 6.17](#) may be defined as

```
Context Customer::orderedBook
```

```
derive:
```

```
self.order.item.book
```

The following observations can be made about this OCL expression:

- The context is the class at the origin of the derived association (*Customer*). The subcontext is the role name of the derived association (*orderedBook*).
- The clause *derive:* is used similarly to the case of derived attributes. What determines if the expression is defining a derived attribute or association is the context and the type of information that is returned by the expression. Derived attributes are defined by expressions that return an alphanumeric, enumeration, or primitive type, while derived associations are defined by expressions that return an object or a collection of objects (data structure).

In object-oriented programming languages, usually the “.” notation can be used only over a single object. OCL, however, allows it to be used over collections of objects.⁷ If x is a collection, then the meaning of $x.y$ is the collection of y that can be obtained as properties of x . For example, *self.order* denotes the set of orders of a customer in the current example; *self.order.item* is the set of items linked to the orders of a given customer; and *self.order.item.book* is the set of all books linked to the items that are linked to the orders that are linked to the customer.

A derived association may also be used to define a subset of objects from a set of linked objects by the application of a filter. For example, imagine that the bookstore has a record on special clients that have bought above 1000 dollars worth of books. Let us call them *gold customers*. Let us also assume that the *Customer* class has a derived attribute *totalSales* with the sum of all orders of a customer. The definition of that derived attribute is not important for the example and it will be ignored now. A derived association between the controller and the *Customer* class may be used to model the gold customers as a subset of the customers of Livir. [Figure 6.18](#) shows the partial conceptual model for this example.

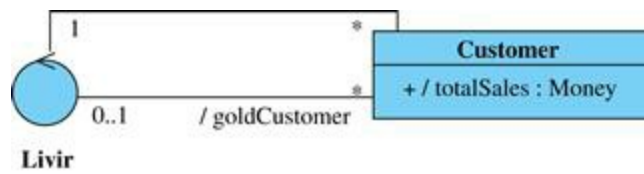


FIGURE 6.18 Derived association defining a subset of a normal association.

The OCL expression for this derived association has as the context the *Livir* controller itself, because it is at the origin of the association. The expression that defines the derived association has to denote the set of customers that bought more than 1000 dollars, that is, the instances of *Customer* whose *totalSales* value is higher than 1000.

The OCL expression to obtain a subset of elements that satisfy a given Boolean expression like this is *select*. The Boolean expression is placed inside parentheses after the *select* command. The Boolean expression usually starts with an iteration variable that in the case of the following expression was named: *aCustomer*.

```
Context Livir::goldCustomer
```

```
derive:
```

```
self.customer->select(aCustomer|aCustomer.totalSales>1000)
```

In the case of this expression, the iteration variable is called “*aCustomer*” but it could have any other name. It denotes each one of the customers for the evaluation of the expression *aCustomer.totalSales>1000*.

The arrow notation “->” is used before *select* instead of the “.” notation because the *select* function is applied over the structure of the set, not over its elements.⁸

OCL also allows the iteration variable to be omitted if the context of the expression is not ambiguous. Thus, the expression above can be simplified to

```
Context Livir:goldCustomer
```

```
derive:
```

```
customer->select(totalSales>1000)
```

Observe that inside the parentheses after *select* the context is potentially ambiguous. There, the property *totalSales* could refer both to a customer and to the instance of *Livir*. What allows us to eliminate the ambiguity is the fact that the property *totalSales* exists only in the *Customer* class; it does not exist in the *Livir* class. If it existed in both classes, then the nonabbreviated notation should be used in order to eliminate the ambiguous expression.

6.4.4 Aggregation and composition

Some associations may be considered *stronger* than others in the sense that they define one kind of object that is composed of others. A house, for example, is composed of rooms, a book of chapters, an undergraduate career of courses, etc.

If the association is exclusive, in the sense that an object may not be part of another object at the same time, then it is considered a strong part-whole association, which is called a *composite aggregation*,⁹ and it is drawn with a black diamond on the role that represents the whole (Figure 6.19).



FIGURE 6.19 An example of composite aggregation.

The multiplicity of the role that represents the composite (the side where the diamond is), in the case of a composite aggregation, should be 1 or 0..1, and nothing else, because composites do not share parts, even with objects of the same class.

Composite aggregation indicates exclusivity in the part-whole association. When this exclusivity is not mandatory (the part may be part of other aggregates), then a white diamond is used instead, and the association is called a *shared aggregation* (Figure 6.20).

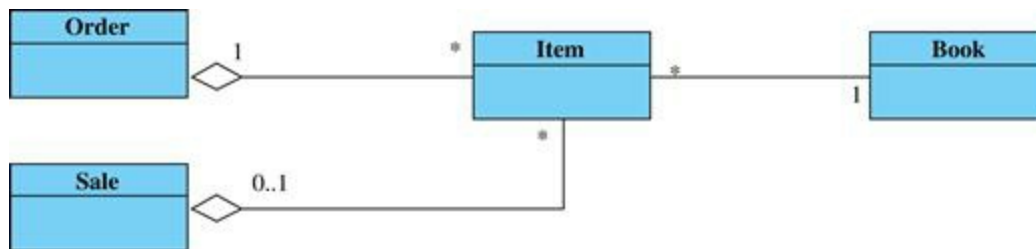


FIGURE 6.20 An example of shared aggregation.

The white diamond indicates a shared aggregation where the part may belong to different wholes at the same time. In Figure 6.20, a given *Item* must be part of an *Order*, but also *may* be part of a *Sale*.

Composite and shared aggregation are special associations and should be used with much parsimony, that is, they should be used only when the team is sure that it is the case that an object is really *part* of another, and not just a normal association. Even the UML specification (Object Management Group, 2011) states that the precise semantics of a shared aggregation varies between application areas and modelers.

It is common to see aggregation and composition being abused in models, when objects that are not part-whole related are linked by that kind of association. For example, a customer is not part of an order, if not for any other reason but the fact that a customer is a person and an order is not a physical thing, but a transaction. Composite and shared aggregations should unite elements of the same nature: physical with physical, and abstract with abstract. An order may be associated to a customer, but the order is not *made* of customers.

There are few real advantages in using aggregation in conceptual modeling. This is another reason to minimize or even abolish their use in conceptual models. Among the

advantages is that composite or shared aggregation parts usually have attributes that are combined and derived in the whole. For example, the total value of an order is the sum of its items; the weight of a package is the sum of the weight of each book; when a car is sold, all its parts are sold too; and so on. However, these concerns usually appear only at design time.

6.4.5 *n*-ary associations

Most associations are binary, that is, most associations have only two roles. However, there are situations where associations with three or more roles are needed. In UML these associations can be represented by using a diamond that connects all the roles of the association, as in [Figure 6.21](#).

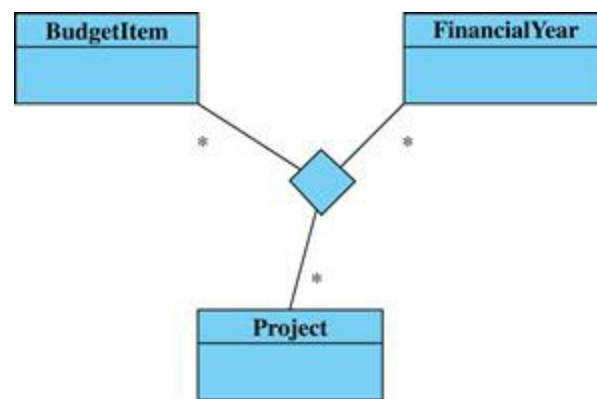


FIGURE 6.21 An example of a ternary association.

The example in [Figure 6.21](#) was taken from a budget control application. Each project may be associated to multiple budget items and multiple financial years. Each financial year may have a number of budget items per project. A ternary association (that is, an association with three roles) represents this situation, as seen in [Figure 6.21](#).

These cases are not frequent, but they may appear in a project. Before deciding to use a ternary or *n*-ary association (which may be a headache during design and implementation), the team should verify if they are not confusing it with a situation where a combination of binary associations should be used instead. For example, the case shown in [Figure 6.22](#) is not correctly modeled as a ternary association, because the relation between the author and the publisher happens only when the author publishes a book with a publisher. Then, the association between the author and the publisher would be a derived one.

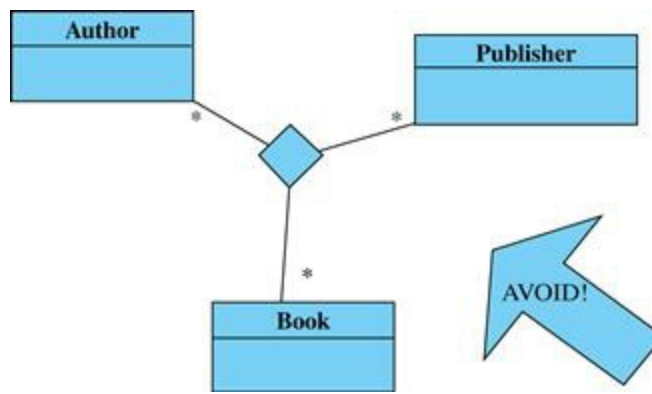


FIGURE 6.22 An unsuitable ternary association.

The model represented in [Figure 6.23](#) is more precise than the one in [Figure 6.22](#) for the case being discussed. Only in a scenario where the same author could publish the same book using a different publisher and the publisher could publish the same book with a different author (!) would there be a case for a ternary association.

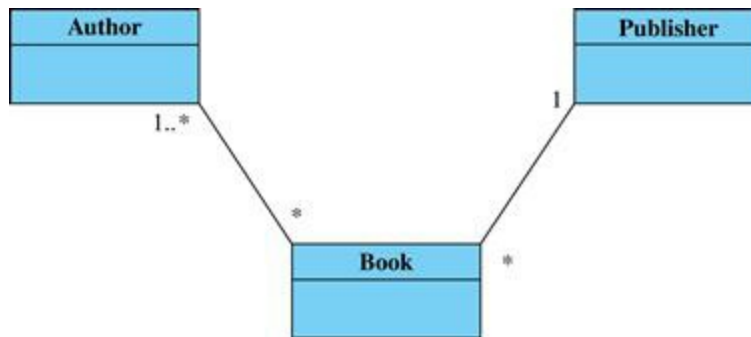


FIGURE 6.23 The model of [Figure 6.22](#) correctly represented with two binary associations.

Real ternary associations, however, may still be replaced by a combination of binary ones. Maybe the simplest approach to accomplish this is to replace the ternary (or n -ary) association by a new concept. Sometimes, naming the new concept may be tricky, but it can be done. For example, the model of [Figure 6.21](#) could be replaced by the one shown in [Figure 6.24](#).

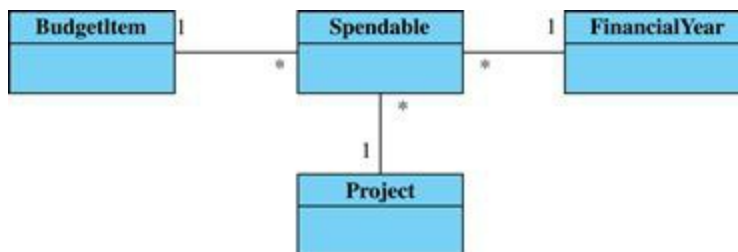


FIGURE 6.24 Model where a ternary association is replaced by a new concept.

However, there is a subtle semantic difference between the models in [Figures 6.21](#) and [6.24](#). In the case of [Figure 6.24](#), there may be two or more *Spendable* instances with exactly

the same budget item, financial year, and project. This cannot happen in the case of [Figure 6.21](#). Thus, for this simplification to have the same semantics as the ternary association, it is necessary to complement it with an invariant (see [Section 6.7](#)).

6.5 Collections

Simple collections of objects that do not have any particular meaning or structure should not be, in principle, represented as concepts, but as associations. Any association role *already* represents a collection of objects of the associated class. Thus, considering again the example of [Figure 6.15](#), *fleet* is a role that associates a set of cars to a person. The model presented in [Figure 6.25](#) is, therefore, unnecessary and redundant.

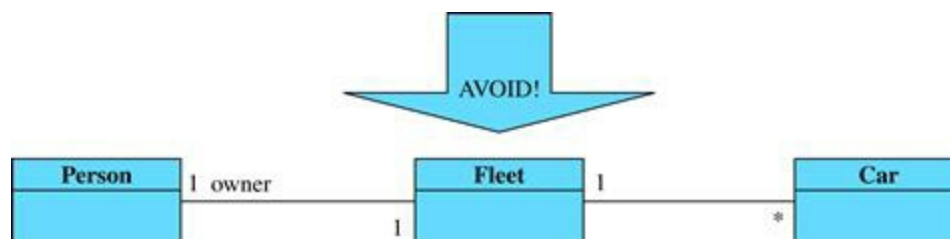


FIGURE 6.25 A simple collection being inadequately represented as a class.

Thus, unless the fleet has its own attributes or associations (for example a person having multiple fleets located in different places) the correct form to represent a simple collection of objects is through an association role, not through a class ([Figure 6.26](#)).



FIGURE 6.26 A simpler model for the situation presented in [Figure 6.25](#).

Thus, the team must be aware that if the collection has particular attributes or associations, then it must be represented as a concept. For example, an *Order* is initially understood as a set of items, but an *Order* also has attributes such as issue date, total value, discount, etc., and associations such as the one with the customer, who may have multiple orders. If that is the case, the collection is a concept. However, a collection with no particular attributes or associations is just a collection, and therefore it is modeled as an association role.

Association roles may be thought as representing *abstract data types* ([Liskov, 1974](#)) in conceptual models. In design models they would also represent *concrete* data types. Abstract data types are defined only by behavior, not by structure. They appear, for example, in the following:

- The *set* structure, in which elements do not repeat and have no defined position.
- The *bag* structure, in which elements may repeat but still do not have a defined

position.

- The *ordered set* structure, in which elements do not repeat but have a defined position.
- The *sequence* structure, in which elements may repeat and have a defined position.

The *concrete* data types, on the other hand, are physical implementations for the abstract versions. For example, a list may be implemented in many forms, such as an array, binary tree, linked list, etc. In the case of concrete data types, besides behavior, there is also a physical structure that defines them.

6.5.1 Set

An association role, by default, is a *set*, that is, a structure where elements do not repeat and where there is no defined position for them. In [Figure 6.26](#), the *fleet* is an example of a set.

If someone tries to link the same car to the same person a second time, nothing happens, because it already belongs to the set.

6.5.2 Ordered set

Suppose a book that is not yet in stock has a list of people interested in buying it as soon as it is available. If it cannot be assured that the quantity of books received will fill all the orders, then the fairer way to attend to these customers is by establishing a line from the oldest reservation to the newest. This requires that the customers have a position in the line.

If the same customer cannot reserve the same book more than once, then it is still a structure with no elements repeated, but now with defined positions. This leads to the *ordered set* structure. In the diagram, the role may be marked with the constraint $\{ordered\}$ to indicate that it is a collection where elements do not repeat but have defined positions, as seen in [Figure 6.27](#).



FIGURE 6.27 An ordered set of reservations for a book.

The constraint $\{ordered\}$ over an association role means that the elements linked on that role have position (first, second, third, ... , last), but still cannot be repeated.

6.5.3 Bag

There are situations in which the position of the elements does not matter, but each element may appear more than once in the collection. That structure is called a *bag*.

For example, one may need to know how many people viewed the details of a book, and it may be important also to know how many times each one viewed it. It is not

necessary to know who viewed it first, but the information on the number of views is necessary. That is a typical case for using a bag. Each time a person views the details of a book, a new link is added on the association defined as a bag. The model is shown in [Figure 6.28](#).



FIGURE 6.28 An example of a bag.

6.5.4 Sequence

The *sequence* allows elements to be repeated and also assigns a position to them. For example, imagine that people that buy a given value worth of books qualify for receiving a gift, but a limited quantity of gifts is available. Thus, a list of people that qualify to receive the gift may be built and the gifts will be distributed to the first members of the list as soon as gifts become available. If someone again buys more than the limit established to receive a gift, she will enter the list again. Thus, in that structure the position of an element is relevant and each element may appear more than once: this corresponds to a sequence structure, as seen in [Figure 6.29](#).

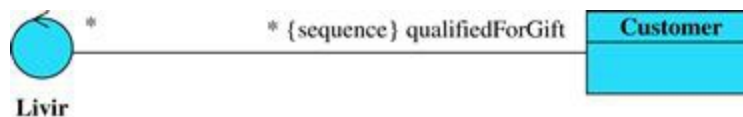


FIGURE 6.29 An example of a sequence.

The multiplicity *** on the *Livir* side means that not all customers are qualified for a gift and that they may qualify more than once.

The sequence has two important special cases. When the first element to be removed is always the oldest in the sequence, as in the case of [Figure 6.29](#), then the structure is also a *queue*. UML does not define a constraint for queues, but a stereotype on the sequence constraint would do the job: `{sequence} <<queue>>`.

When the first element to be removed is the newest in the sequence, then it behaves like a *stack*, and may be stereotyped as such: `{sequence} <<stack>>`. Queues and stacks that do not repeat elements may be defined respectively as `{ordered} <<queue>>` and `{ordered} <<stack>>`.

A general list may have elements removed and inserted at any position. Queues and stacks have elements inserted and removed only at predetermined positions, as explained above, although, in some cases, all of their elements may be browsed.

A curiosity to be mentioned now is that in the real world most collections are strict *sets* and not *sequences*, but even so, the most used programming structure is still the

sequence, and not the set, which indicates a gap between reality and the data structures used in programming.

6.5.5 Map

When a concept has a unique attribute, a *map* may be created from that attribute to the concept, in such a way that it is easier to identify and find specific instances of the concept. For example, as the ISBN is unique for a book, associations to *Book* may be qualified by the ISBN, meaning that instead of having a set of books, the origin class has access to a map that allows immediate identification of any book if its ISBN is known. Figure 6.30 shows an example of qualified association.

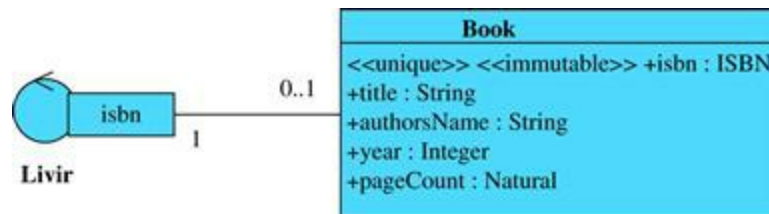


FIGURE 6.30 An example of qualified association representing a map.

The small rectangle at the left side of the association in Figure 6.30 represents a *qualifier* for the class at the opposite side of the association. Notice that instead of “*,” the role of the right side is now bounded by “0..1”; the association should now be read as follows: “for each possible ISBN there is no more than *one* book.” Some valid ISBN values may not be associated to any book, because they were not registered in the system or because the ISBN, although valid, has simply not yet been assigned to a book; however, if one ISBN is taken by a book, no other book may share it. In practice, qualified association is still an association “to many,” but the objects belonging to the role may be accessed by one of their attributes.

The role at the left side (the qualifier side) is marked with “1,” meaning that each book has a single ISBN, not less or more. This has to necessarily be true, because an ISBN is unique and not optional for the class *Book*, and therefore, it is mandatory and does not allow repetition.

The qualifier is not only a way to access objects in an easier way. It is also a powerful modeling tool. For example, consider a situation where a person may own many phones, but only one of each kind. How do we represent this? An elegant model for that situation is shown in Figure 6.31.

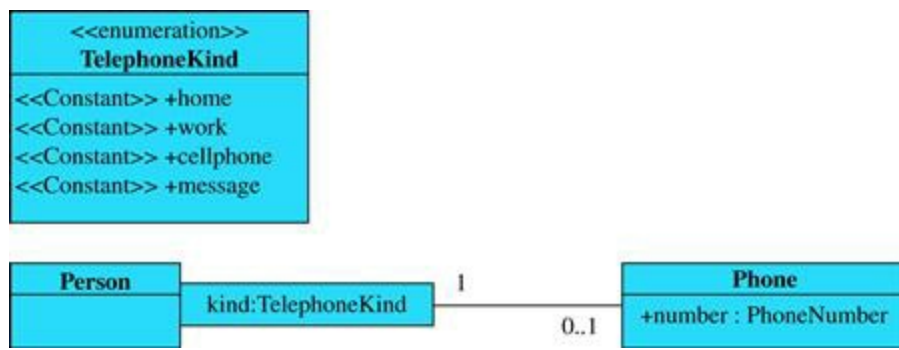


FIGURE 6.31 Use of a qualifier as a modeling tool.

In the case of Figure 6.31 one person may have up to four phones, but just one of each kind. If more kinds are added to the enumeration later, then more phones may be associated to each person. If the role on the right side was 1 instead of 0..1, then one person should necessarily have four phones, each one a different kind. That use of a qualified association with a mandatory role usually makes sense only when the qualifier is an enumeration or a finite set.

In the example of Figure 6.30, the qualifier is an attribute of the qualified class, and in that case it is called an *internal qualifier*. On the other hand, in Figure 6.31 the qualifier is not an attribute of the class, and it is called an *external qualifier*.

6.5.6 Partition

In Figures 6.30 and 6.31, the multiplicity 1 or 0..1 on the right side of the figure means that the association is a mapping, that is, for each value of the key there is at most one instance of the qualified class.

What if instead of “1” or “0..1” the multiplicity was marked with “*”? In this case, the association would read as follows: “for each key there is a *set* of instances of the qualified class.”

For example, books may be classified by genre. As many books may belong to the same genre, the *genre* attribute may not be unique. But it is possible to define a *partition* on the set of books based on their genre, as shown in Figure 6.32.

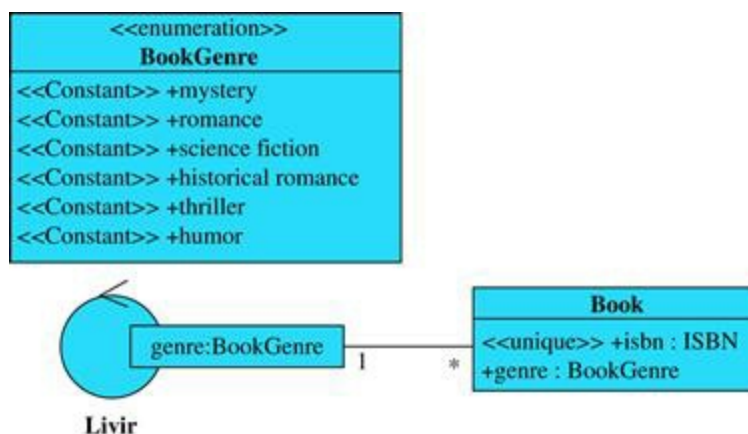


FIGURE 6.32 An example of a partition.

Figure 6.32 states that for each particular value of genre there is a set of books (with zero or more elements). The role multiplicity 1 on the left side indicates that every book has a genre.

6.5.7 Relation

Now, imagine that a book may have more than one genre, as, for example, *The Hitchhiker's Guide to the Galaxy* (Adams, 1979), which can be classified both as science fiction and humor. For representing the possibility of a book having more than one genre in the example of Figure 6.32, it would be sufficient to change the multiplicity on the left side of the association to * (if a genre is optional) or 1..* (if it is mandatory). Additionally, the genre should be removed from the attributes of *Book*, becoming an external qualifier, because the *genre* attribute should allow multiple values. Figure 6.33 shows the resulting relation.

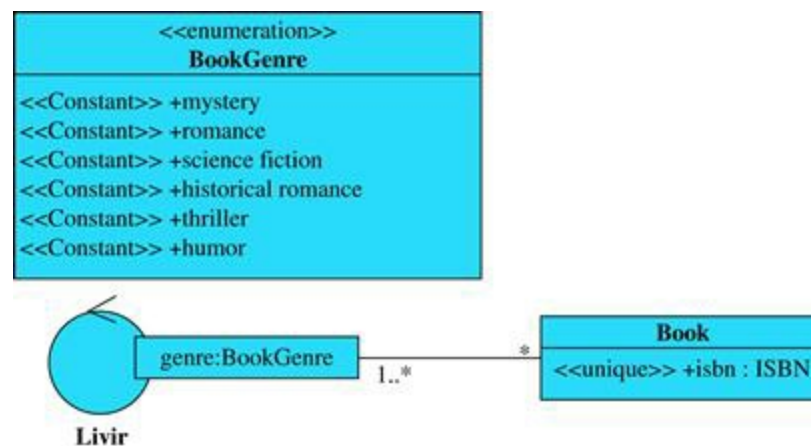


FIGURE 6.33 An example of a relation.

The relation of Figure 6.33 states that a book may have one or more genres and a genre has a set of books associated with it. As a book may have more than one genre, then the qualifier should be necessarily external.

6.6 Organization of the conceptual model

The elaboration of the conceptual model involves more than just putting concepts, attributes, and associations together. For the conceptual model to be a reliable and organized representation of real-world information, it is necessary to use some modeling techniques.

The main organizational techniques for concepts may be distinguished into three groups:

- *Structural*: Representing generalizations among concepts, as for example, *BankAccount* generalizing *SavingsAccount* and *CheckAccount*.
- *Associative*: Representing associative roles that some concepts may play related to

others, as for example, *Student* and *Professor* being roles a *Person* may play related to a school.

- *Temporal*: Representing relations among different states of a concept, as for example an *Order* being in states *AttendingPayment* and *Paid*.

Novice analysts and sometimes even experienced ones tend to think that the only way to factorize information is by the use of inheritance, or generalization.¹⁰ However, it is necessary to apply some judgment before deciding on using this or that technique. Inheritance may be used only when a concept effectively has two or more subtypes. Although UML allows an instance to change its class, only a few languages, such as Smalltalk (Goldberg & Robson, 1989), tried to implement such a feature, maybe because there are some unresolved inherent difficulties associated to the fact that one instance could change its internal structure on the fly. Most object-oriented languages ignore this feature due to the problems it raises and due to the fact that it is usually unnecessary, because most behaviors that could be modeled by this feature also have alternate simpler representations. Thus, for practical terms, it is usually the case that an instance may not change its class: it is born as a member of one class and it will die a member of the same class.¹¹ As students may become professors, they do not qualify as subtypes of people (the best solution for modeling this situation is given in Section 6.6.2).

Another counterexample is *Worker* and *Customer* being modeled as subclasses of *Person*. This is bad first because no one is born a worker or customer; a person *becomes* a worker or customer when *associated* to a company. More than that, the same person may be a customer or worker in *more than one* company. Using inheritance in this case would create conceptual problems that can produce duplicity in records and data inconsistency.

The following subsections present details on the three approaches to the organization of concepts.

6.6.1 Generalization, specialization, and inheritance

For years, inheritance was considered the most important feature of object-oriented languages. Languages such as Smalltalk, for example, were built over a single hierarchy of classes based on inheritance.

As time went by, this emphasis lost strength, because it was perceived that inheritance is not always the best idea for solving modeling problems. Today, inheritance is considered just one of the techniques that help to factorize information that otherwise would be repeated in classes.

Inheritance, in object-oriented systems, occurs when two classes relate through a special association called a *generalization* (or a *specialization*, depending on the direction you look). A class that generalizes another class is its *superclass*, and the specialized one is the *subclass*.

Generalization should be used every time there is a set of classes X_1, \dots, X_n that have specific differences and common similarities, so that the similarities can be grouped in a superclass X (generalization of X_1, \dots, X_n) and the differences maintained in X_1, \dots, X_n .

The generalization has a semantic much different from the other associations of the

conceptual model. It exists only between classes, not between instances as normal associations do. For example, if a class *Customer* is associated to the class *Order* by a normal association, then instances of *Customer* may be linked to instances of *Order*: the normal association allows the links to exist. On the other hand if a class *BankAccount* is a generalization of *CheckAccount*, then every instance of *CheckAccount* is *also* an instance of *BankAccount* (Figure 6.34). There are not two instances to be linked, just one instance with the properties defined in both class. In the example, every instance of *CheckAccount* has three attributes: *number*, *overdraftLimit*, and *overdraftFee*. However, not every instance of *BankAccount* is an instance of *CheckAccount*: the generalization is anti-symmetric. An instance of *BankAccount* has only one attribute: *number*.

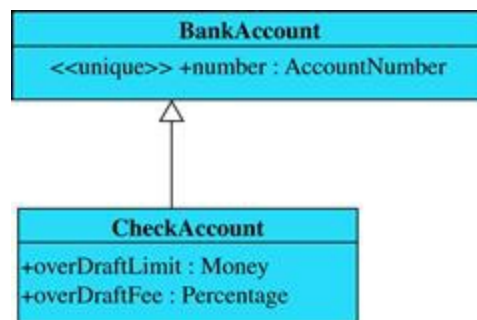


FIGURE 6.34 An example of generalization with inheritance.

Thus, one of the reasons to use inheritance is when there are some similar properties (attributes, associations, or methods) in different classes, which may be factorized in an abstract construct such as a superclass.

If the superclass may have its own instances, then it is a normal class. However, most of the time superclasses are defined not to allow direct instances: only their subclasses may have instances. In this case they are known as *abstract* superclasses. Abstract superclasses are represented in UML by a class with its name written in italic, or, more visibly, with the constraint `{abstract}` (Figure 6.35).

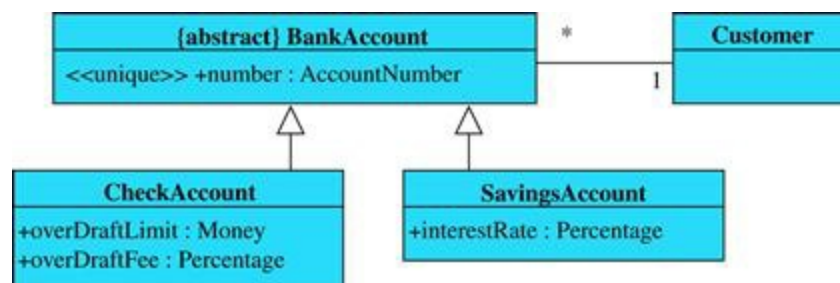


FIGURE 6.35 Two classes being generalized by an abstract class.

In Figure 6.35, the *BankAccount* class is abstract, and therefore cannot have direct instances; only its subclasses may have instances. Instances of *CheckAccount* have *number*, *overdraftLimit*, and *overdraftFee* as attributes; instances of *SavingsAccount* have *number* and *interestRate* as attributes. Both subclasses also inherit the association to the class

Customer, and all instances of both classes must, therefore, be linked to an instance of *Customer*.

Generalization is not advisable when the superclass has no properties, that is, no attributes, associations, or methods of its own (Figure 6.36).

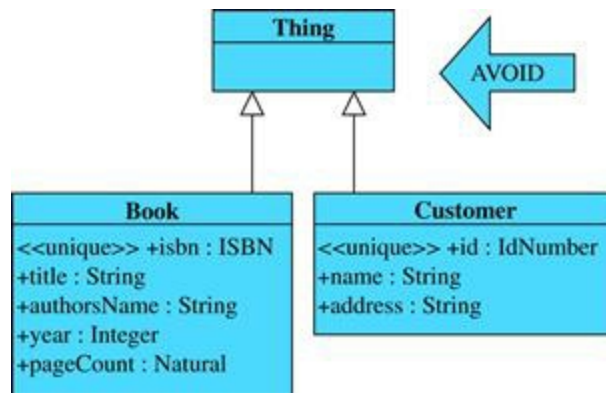


FIGURE 6.36 Situation where generalization is not recommended, because there are no generalized properties.

Generalization should also not be used when subclasses do not have properties (attributes, associations, or methods¹²) that differentiate them from each other as in Figure 6.37.

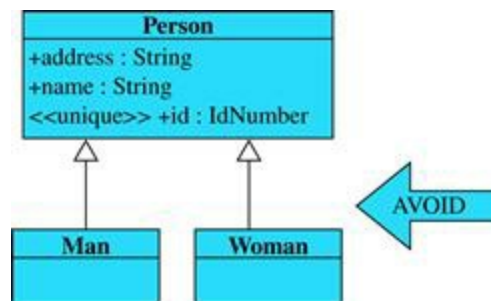


FIGURE 6.37 Situation where specialization is not recommended because there are no properties being specialized in subclasses.

In these cases, the ideal solution is to use a single attribute, possibly typed with an enumeration, to differentiate instance groups of objects that have the same properties (Figure 6.38).

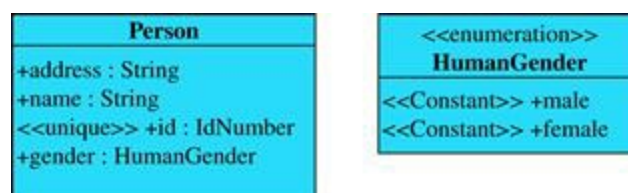


FIGURE 6.38 A more adequate model for the situation of Figure 6.37.

Besides these rules, before deciding on the use of inheritance, it is advisable to verify if

the generalization really represents a structural classification of the elements, and not an associative or temporal organization, as seen in the following sections.

6.6.2 Association classes

One issue frequently misunderstood in conceptual modeling is related to the definition of generalization among classes that are not really structural subtypes, but *roles*. For example, a bookstore may have two “types” of people: customers and workers. After discovering that both have a name, address, phone, etc., an analyst could assume that they are two concepts that should be generalized as *Person*. But that apparently simple solution (Figure 6.39) generates a complicated problem, because they are not different types of people, but different roles that people could play when relating to a company.

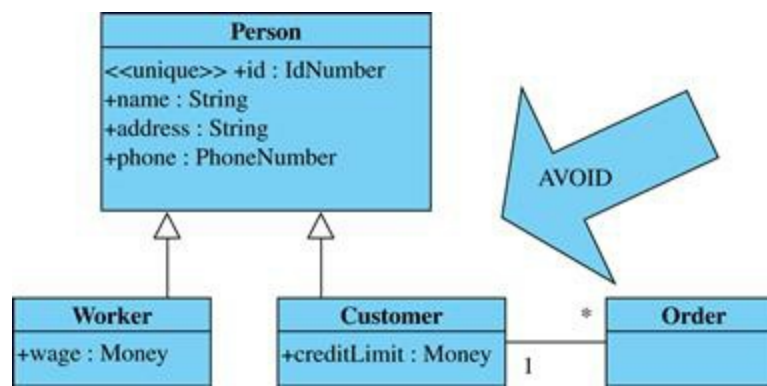


FIGURE 6.39 Unsuitable way to represent roles with generalization.

Why is this a problem? I imagine that a bookstore worker decides to buy books at her workplace. In this case the worker will be behaving like a customer. An unsuitable although very frequent solution to this is to create a second record for the worker as a customer, as if she was a different person. As a consequence, the same person would have two records in the system: one as a worker and another as a customer. This produces data redundancy and it is a source of data inconsistency, because, for example, if the worker has registered a change in address, the customer may still keep the old address. As they are the same person, this information is inconsistent.

A more suitable solution is to consider that there exists a *Person* that may relate to a *Company* in at least two ways: as a customer and as a worker. The specific properties of a customer (credit limit, for instance), and of a worker (salary, for instance), would be properties of the associations, and not of the person. To represent these association properties, an *association class* should be defined for each one, as shown in Figure 6.40.

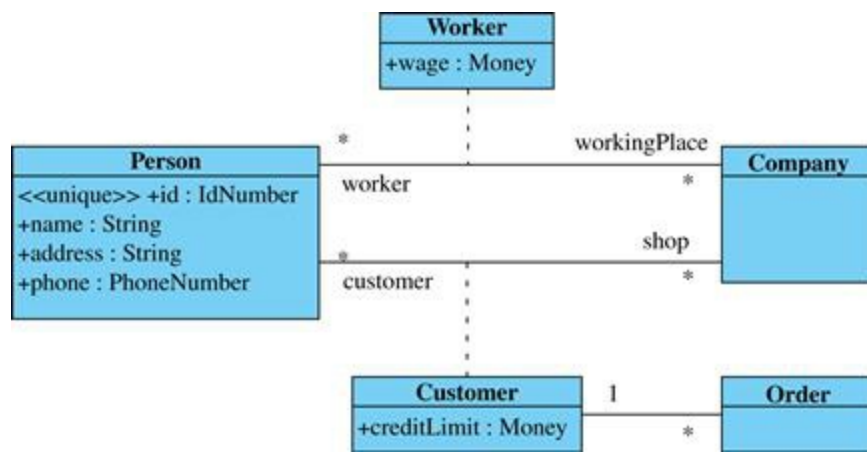


FIGURE 6.40 More suitable representation of roles as class associations.

Therefore, when the same object may play different roles related to other objects, these roles should not be represented as subclasses, but as *association classes*.

In order to decide which situation demands inheritance and which situation demands association classes, it may be verified if the “subtypes” depend on the existence of a third class to make sense. If the “subtype” depends on a third class, then the solution consists of using an association class. For example, nobody may be simply a student; if someone is a student, then she must be associated to a school or at least to a teacher.

Thus, it is unsuitable to create classes that represent kinds of people that in reality are not subclasses, but roles. Workers, teachers, students, principals, customers, etc., should not be subclasses of *Person*. Concepts like these only make sense if related to other concepts such as *Company*, *School*, *Department*, etc. A person may be worker *at* a company, student *at* a school, etc.

The difference between using an association class and an intermediary concept is subtle. [Figure 6.41](#) shows an example of a reservation being modeled as an intermediary concept, and [Figure 6.42](#) is a version of that reservation modeled as an association class.

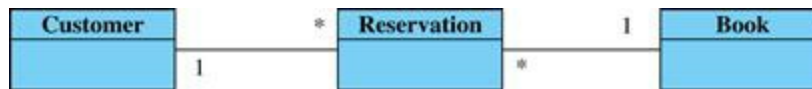


FIGURE 6.41 A reservation being modeled as an intermediary concept.



FIGURE 6.42 A reservation being modeled as an association class.

In the case of [Figure 6.41](#), a reservation associates a customer to a book. In the case of [Figure 6.42](#) the customer and the book are directly associated, and the reservation, as an

association class, is a consequence of that association. However, in [Figure 6.41](#), the same customer may have many reservations for the same book (nothing in the model prevents it), while in [Figure 6.42](#) one customer may have just one reservation for each book. The link between the customer and the book may exist or not; it is not a bag. However, if both roles of the association in [Figure 6.42](#) were marked with `{bag}`, then a user could have more than one reservation for the same book and the model in [Figure 6.42](#) would be equivalent to the one in [Figure 6.41](#).

The issue of having different records for the same person, as discussed above, frequently creates problems in information systems. For example, a student enters the university and is registered as such. Then, she receives a scholarship and a new record is created. Then, she works at a lab, and is registered again. Finally, she is hired as a professor and a new record is created. She appears many times in the university records, as if she was different people. Her old records become out of date; address and phone numbers usually are only updated in the most recent records. This happens because systems usually use inheritance ([Figure 6.43](#)), or completely separated records ([Figure 6.44](#)) in these cases.

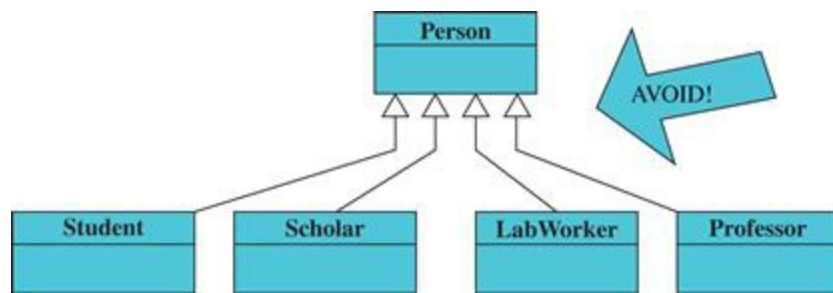


FIGURE 6.43 Unsuitable representation of many records for the same person with inheritance.

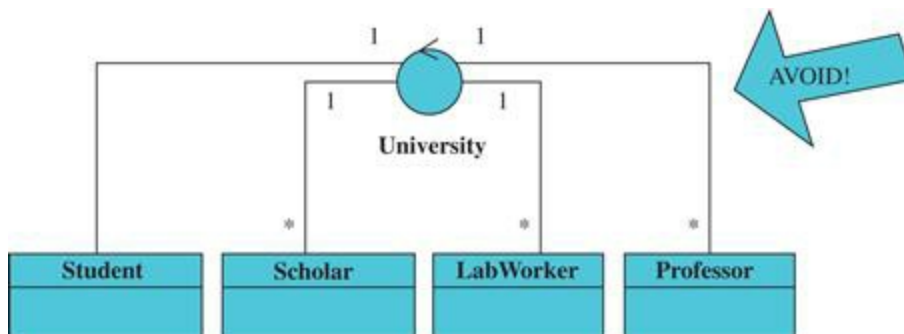


FIGURE 6.44 Unsuitable representation of many records for the same person as separated concepts.

To avoid this problem, as seen before, the solution is to recognize that a person is always the same. The person's roles at the institution change, but the person does not become a new person. Thus, the solution for this situation is more suitable when based on association classes, as shown in [Figure 6.45](#).

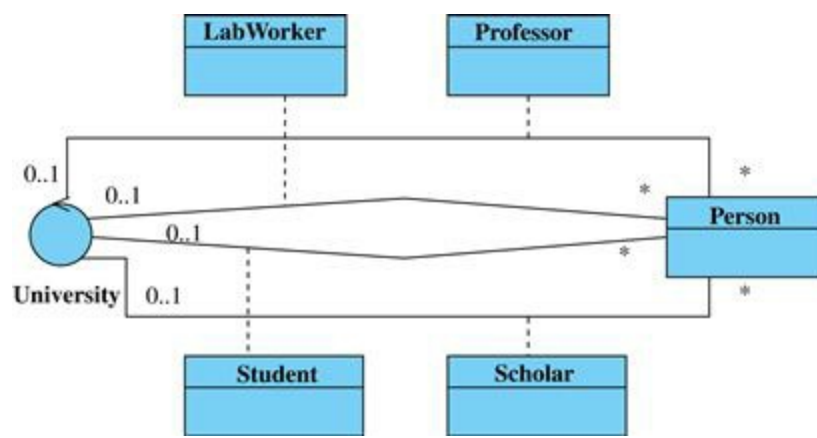


FIGURE 6.45 Representation of many records for the same person as association classes.

In [Figure 6.45](#), if a person can play the same role more than once, then the multiplicity at the left side of the association should be $*$.

A reasonable exception may be admitted if just one kind of person exists in the context of the system. For example, if the system only manages information about professors and that is the *only* role a person may play at the university in the context of that information system, then it would be easier just to create a class named *Professor*, and in the context of that system, *Professor* is a synonym of *Person* because there are no other kinds of people. The same thing happens in the Livir example, where it was decided to keep *Customer* as the only class representing people, instead of transforming it into an association class.

6.6.3 Modal classes

Modal classes are used to model concepts with instances that can change from one state to another during their existence, changing, possibly, their properties' structure, including attributes, associations, and behavior. Although a few programming languages allow instances to change their structure by changing their class, this is not assumed as a modeling principle because such changes may create unpredictable structural problems.

Even if an instance could change its class, redefining its attributes and associations remains a problem. However, there are modeling techniques that allow one to represent situations that would need objects that change class without using this feature. The idea is not to change the object class, but its *state*. When a TV is turned on it does not become a new type of TV: it just changes its *state*.

Three increasingly complex situations related to state modeling may be identified:

- *Stable transition*: The different states of an object do not affect its structure, but only its property values (attributes and links).
- *Monotonic increasing transition*: As the object changes state it may only keep its properties or acquire new ones.
- *Nonmonotonic transition*:¹³ As the object changes state it may keep, acquire, or lose properties.

Modeling solutions for the stable and monotonic increasing transitions are quite simple. However, for modeling the nonmonotonic transition, a design pattern named

State (Gamma, Helm, Johnson, & Vlissides, 1995) is necessary, as seen in the following subsections.

6.6.3.1 Stable transition

Frequently, different states of an object may be represented by a simple attribute. In fact, the set of all attributes and links of an object defines its state. For example, the state of an order could be modeled simply as a *state* attribute typed with an enumeration whose values may be “ongoing,” “concluded,” and “paid.”

Another example is the *suspended* attribute of the *Address* class (Figure 6.46). If a package sent to that address returns, then the address is marked as suspended (the attribute is changed to true). When the customer updates the address it is changed to false again. The default value for the *suspended* attribute is false.

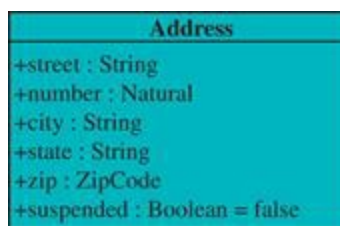


FIGURE 6.46 A concept with a stable state transition.

If the attribute *suspended* is true, then the address cannot be used for deliveries.

The transition is considered stable because just the value of the attribute changes. The inner structure of the object is kept the same. This does not happen with the two cases presented in the next subsections.

6.6.3.2 Monotonic increasing transition

The situation is a bit more complex when the concept may acquire new attributes or associations as it changes state. For example, invoices that are waiting for payment have only *dueDate* and *dueValue* as attributes. But an invoice that is paid additionally has *paymentDate* and *paidValue* (Figure 6.47).



FIGURE 6.47 A concept with a monotonic increasing transition.

It is said that the transition in Figure 6.47 is monotonic increasing because new attributes and associations may be created for it, but not eliminated. This implies that a paid invoice cannot go back to be a new invoice again.

It would not be adequate to model that situation with inheritance, as seen in [Figure 6.48](#), because in that case, an instance of *NewInvoice* should be transformed into an instance of *PaidInvoice*, or it has to be destroyed and created again from scratch, and all its references should be rebuilt.

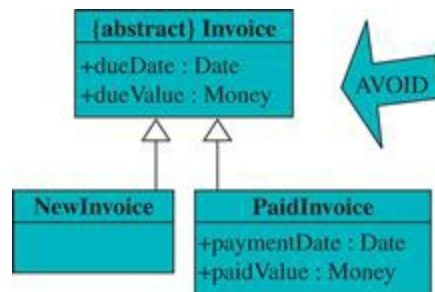


FIGURE 6.48 Unsuitable way to model a monotonic increasing transition with inheritance.

Another solution that is not very nice, but still very popular, consists of creating a single class *Invoice* and allowing certain attributes to be *null* until the class changes its state, as shown in [Figure 6.49](#). Usually, the verification of the instance consistency is done in the updating methods. But an *invariant* could also be used (as explained in [Section 6.7](#)) to guarantee that no instance reaches an invalid state, as, for example, with a defined *paymentDate* and null *paidValue*.

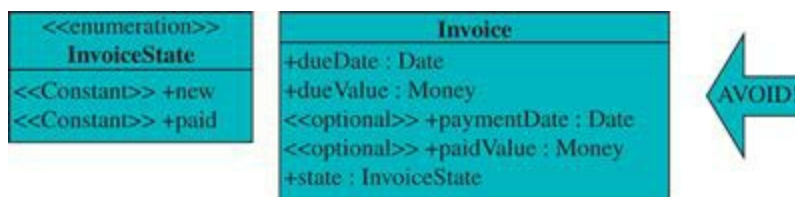


FIGURE 6.49 Unsuitable way to model a monotonic increasing transition with a single class and null attributes.

This model is still not good because it generates a class with low cohesion, which has, in consequence, complex consistency rules that should be checked frequently. This kind of class is highly susceptible to design and programming errors.

It would be better to model the invoice concept so that the consistency of the objects could be constrained by the structure of the model itself, and not by external rules. As it is a monotonic increasing transition, then it is possible to model this situation just by splitting the original concept in two: one that represents the invoice, and another that represents its payment with the properties that were added when the payment occurred. These two concepts are, then, linked by a 1 to 0..1 association ([Figure 6.50](#)).



FIGURE 6.50 Effective way to model a monotonic increasing transition.

With the model presented in [Figure 6.50](#) it is impossible for an invoice to have a defined payment date and undefined paid value, and this is guaranteed without performing any further constraint on its attributes or associations.

The state of an invoice is a derived attribute, which is calculated as follows: if there is no payment linked to the invoice, then the invoice is *new*; otherwise it is *paid*. In OCL the condition may be expressed as

```
Context Invoice::state
derive:
  if self.payment->isEmpty() then
    InvoiceState::new
  else
    InvoiceState::paid
  endif
```

The expression above introduces two new OCL features:

- *if-then-else-endif*, which is one of the OCL selection functions. If the condition after the *if* is true, then the function results in the evaluation of the expression after the *then*; otherwise it results in the evaluation of the expression after the *else*.
- *isEmpty()*, which is a function applied to a collection structure and returns *true* if the collection is empty or false otherwise. It is an easy way to check if there is some object linked to another object or not.

6.6.3.3 Nonmonotonic transition

With *nonmonotonic transition*, an object may both gain and lose properties as it changes its state.

Fortunately, it is not frequent that an information system requires any information to be *lost*. However, sometimes, that can be exactly what should be done.

There are many ways to conceive and model a reservation system for a hotel, for example. One of them consists of regarding the hosting as an entity that evolves from a reservation in three steps:

1. Initially, a potential guest makes a reservation indicating the days she intends to arrive and depart, the kind of accommodation, and the number of people. The hotel provides the fee.
2. When the guest checks in, the arrival date is registered (it may be the same date of the reservation or a different date). The hotel assigns a room that may be different from the one initially reserved and, if this is the case, it informs the guest of the new fee. The expected date of departure continues to exist, although it can be updated during the stay.

3. When the guest checks out, the expected departure date ceases to exist and the effective departure date is registered, and the bill has to be paid.

This set of states may be modeled during business analysis with a state machine diagram similar to the one in [Figure 6.51](#).



FIGURE 6.51 A state machine diagram for modeling a simple hosting process.

If a hosting just acquires new attributes and associations as it evolves through the states of [Figure 6.51](#), then it could be represented by a chain of concepts like the one shown in [Figure 6.52](#).



FIGURE 6.52 Possible model for reservations as monotonic increasing transitions.

However, it is possible to assume that some state transitions may eliminate some attributes or associations that have no more meaning in the new state, as, for instance, estimated dates, which are replaced by real dates, and the type of room, which will be replaced by a real room that already has a type.

In order to represent this kind of situation, it is necessary to use a design pattern known as *State* ([Gamma, Helm, Johnson, & Vlissides, 1995](#)). According to this pattern, a class should be separated from its states. Properties that are shared by all states are kept in the original class, and properties that belong only to one or a few states are moved to the respective states. These states are specializations of an abstract class, as shown in [Figure 6.53](#).

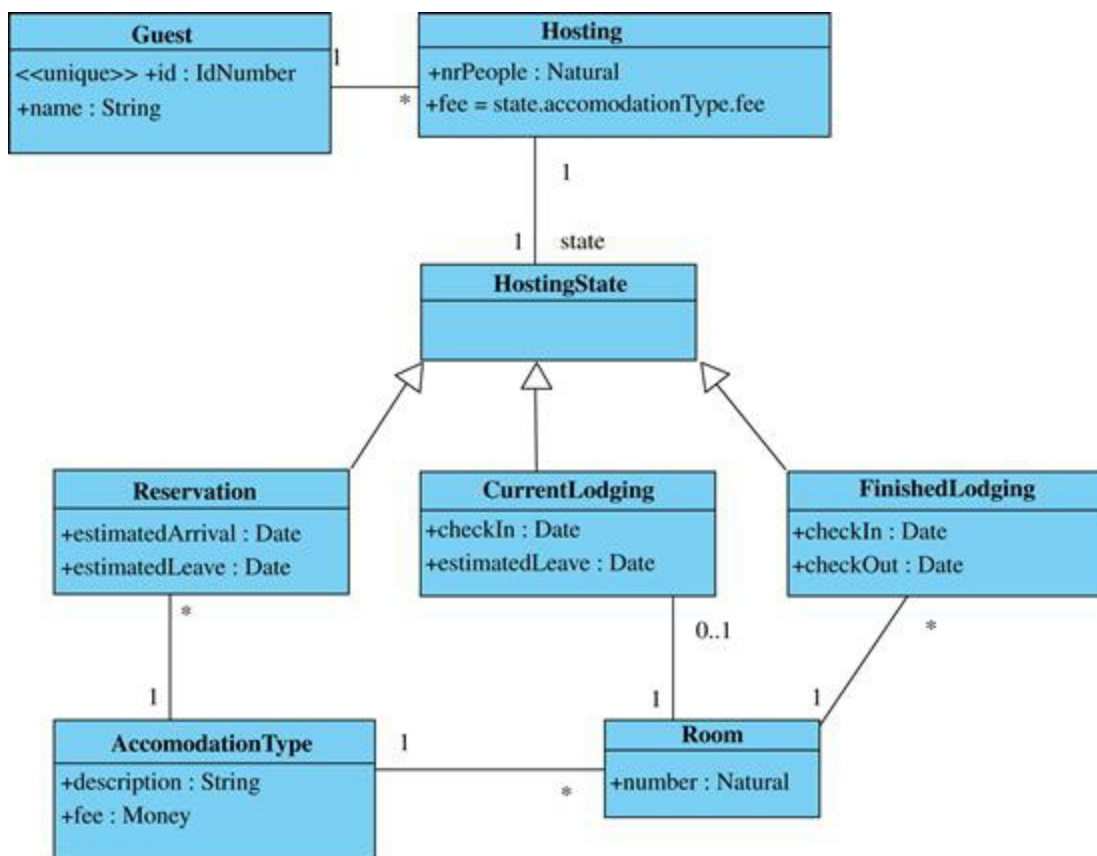


FIGURE 6.53 Nonmonotonic state transitions modeled with the *State* design pattern.

In [Figure 6.53](#), the attributes *nrPeople* and *fee* exist in every possible state of a hosting, and this is why they are represented in the *Hosting* class. The other properties are distributed among the three possible states:

- *Reservation*: A hosting, in addition to the number of people, has estimated arrival and leaving dates and the accommodation type (not yet a room), from which the initial value for the hosting fee is obtained.
- *CurrentLodging*: A hosting, in addition to the number of people and fee, has an effective check-in date, estimated leaving date, and an effective room assigned. The *estimatedArrival* attribute and the straight link to the accommodation type are not necessary anymore and are discarded.
- *FinishedLodging*: A hosting, in addition to the number of people and fee, has effective check-in and check-out dates, and an effective room that was assigned. The estimated leaving date is not necessary anymore and is discarded.

From the point of view of a *Room*, it can be linked to at most one *CurrentLodging* (0..1), but it can be associated to many *FinishedLodging* (*). This way, the temporal property of that association is already modeled too, because the association of a room to lodging may be at most one in the present, but many in the past.

6.7 Invariants

There are situations where the expressivity of the diagrams is not enough to represent rules that should be recorded in the conceptual model. In these cases, the analyst should use *invariants*.

Invariants are general constraints over instances of the classes. Some constraints are represented in the association roles; for example, a constraint that states that an order should have not more than five items can be represented as in [Figure 6.54](#).



FIGURE 6.54 A constraint that can be represented as a role multiplicity.

However, not every possible constraint may be represented in the association roles. Consider the model shown in [Figure 6.55](#).

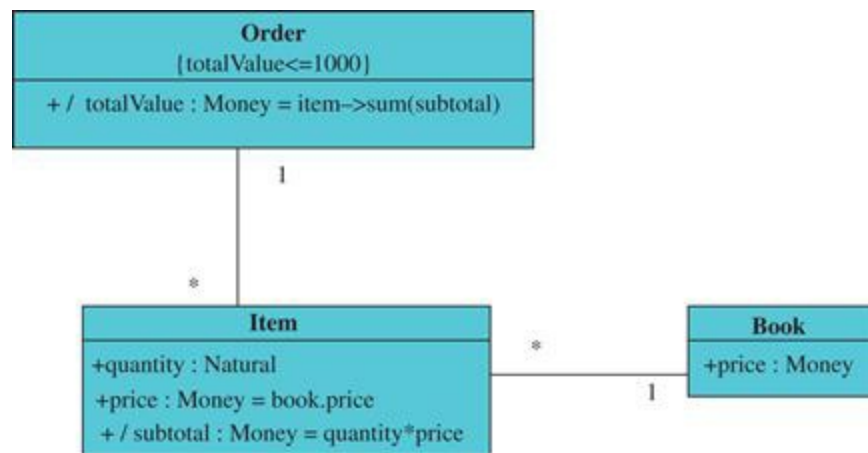


FIGURE 6.55 An example of a class with an invariant.

The *totalValue* of an order is a derived value in [Figure 6.55](#). The *sum* function in OCI returns the sum of the numerical elements that are produced for each of the elements of the set. The expression between parentheses after *sum* indicates how each individual numerical value is obtained from the elements of the set. The expression used to define the *totalValue* in [Figure 6.55](#) is an abbreviation of

```
Context Order::totalValue:Money
```

```
derive:
```

```
self.item->sum(anItem|anItem.subtotal)
```

Someone could also try to write the expression above as *self.item.subtotal->sum(aSubtotal|aSubtotal)* or *self.item.subtotal->sum()*. However, in that case, as the role *item* is a set and not a bag or sequence, the subtotals would be also a set. If two items have the same value for the subtotal, then it would appear just once in the set and the sum would not reflect the intentions of the modeler.

The subtotal in the *Item* class is also a derived value calculated by the following expression:

```
Context Item::subtotal:Money
```

```
derive:
```

```
self.quantity*self.price
```

The item price has an initial value defined as the price of the book that is linked to the item:

```
Context Item::price:Money
```

```
init:
```

```
self.book.price
```

Now, if there is a constraint that states that no order can have a total value higher than 1000 dollars, it would not be possible to represent it in the role multiplicity, as done in [Figure 6.54](#). However, the constraint could be represented by using an *invariant* like the following:

```
Context Order
```

```
inv:
```

```
self.totalValue<=1000
```

Invariants may appear in the diagram indicated as constraints associated to a class. The invariant mentioned above is represented just below the *Order* class name in [Figure 6.55](#).

The invariant declaration in OCL does not have a subcontext; it must be true for every instance of the class, not for a particular attribute or association role.

Many software developers, when faced with the need to use constraints like invariants, choose to incorporate these rules in methods that update the attributes and links of the objects. For example, considering [Figure 6.55](#), a test for the constraint could be added to the method that adds a new item to the order to check if the total value surpasses one thousand dollars; in that case, the operation would be prevented from completing.

The problem with that approach is that the verification would be made only on *that* method, but it would not be a general rule for every method. The developer knows the rule now, but what happens if another developer assumes the project later? What about maintenance? A developer that changes the system some years later would not necessarily know that that rule exists. She would probably not review the requirements document (assuming that it still exists and is updated), and she would potentially introduce a conceptual error in the system if she allows other methods to not follow that rule.

For instance, if the check was implemented only in the operation that adds a new item to an order, what if the item has its price changed? What if the item has its quantity changed?

Thus, *general conceptual rules that always apply should be explicit in the conceptual model, and not relegated to verifications in the programming code*. Inconsistent instances may be

prevented. If it is possible, such constraints may be represented in the diagram structure; otherwise, they should be represented by invariants.

Another example, which happens frequently, is the need to constrain two associations that would be independent otherwise. In [Figure 6.56](#), note that students are associated to a chosen career, each career has courses, and students take courses.

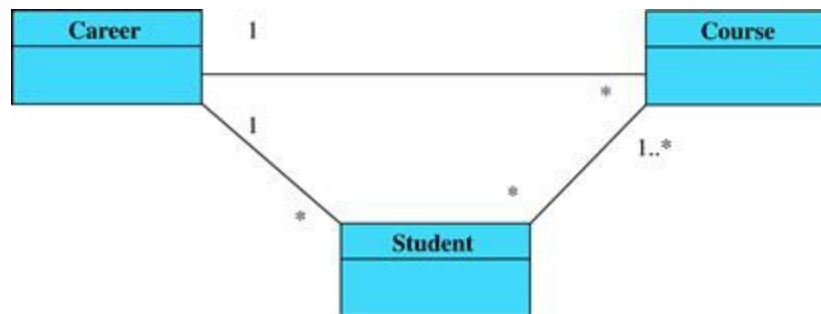


FIGURE 6.56 A model that needs a further constraint to be consistent.

However, the model depicted in the figure does not guarantee that students will choose courses associated to their career. According to the model, a student may take any course, and if that is not the case, a constraint should be added to the model. UML allows constraints to be added between associations (with the use of dashed lines). This notation is an alternate form for an invariant.

To guarantee that a student can only take courses related to her own career an invariant like the following should be provided:

Context Student

inv:

```
self.course.career= self.career
```

The expression uses a feature of sets (not allowing elements to be repeated) to verify that all courses taken by the student belong to the student's career. The expression *self.career* results in the student's career (a set with a single element because of the multiplicity of the role from student to career). On the other hand, the expression *self.course.career* results in the set of all careers associated to all courses taken by the student. If there are courses from different careers, then that set could not be equal to *self.career* (which has a single element). Otherwise, if all careers are the same then the set *self.course.career* will contain a single element (because sets do not repeat elements). If that element is the same as *self.career*, then the invariant will be true and valid.

6.8 Iterative construction of the conceptual model

This section shows how the conceptual model can be refined with information from the expanded use cases. Initially, we discuss how to find concepts and attributes from the use case texts. Next, we discuss how to identify associations of the model. Finally, we

present a complete example with some use cases from the first iterations of the Livir project, and their contribution to the conceptual model.

6.8.1 How to find concepts and attributes

The process of discovering the elements of the conceptual model may vary. However, one of the most useful techniques is to look at the text of the expanded use cases or system sequence diagrams. From those artifacts one can discover all textual elements that eventually refer to information to be managed.

Usually, these textual elements are composed by *nouns* such as “person,” “order,” “payment,” etc., or by expressions that denote nouns (known in linguistics as *noun phrases*), such as “payment authorization.” Moreover, sometimes even verbs may represent concepts, when verbs express an act that corresponds to a noun, as, for example, the verb “paying,” which corresponds to the noun “payment.”

The analyst should mind the project goals and scope when looking for the elements of the conceptual model. It is not advisable to represent in the conceptual model information that is irrelevant to the system. Thus, not every noun, adjective, or verb should be considered as a model element. The analyst is responsible for understanding what the real information needs of the system are, and for filtering the irrelevancies.

The process of identification of concepts and attributes consists of the following:

1. Identify in the expanded use case text or in the arguments of the operations in the system sequence diagram the words or phrases that correspond to concepts that are relevant to maintain the record.
2. Group words or phrases that refer to the same entity as, for example, “buy” and “acquisition,” “customer” and “client,” etc.
3. Decide which of the identified elements are complex concepts, and which are mere attributes. Usually, attributes are the elements that may be considered alphanumeric (names, numbers, codes, monetary values, Boolean values, etc.), enumerations, or primitive (dates, ISBNs, etc.).

Applying this technique to the use case of [Figure 5.10](#), the main elements related to the use case may be found. [Figure 6.57](#) shows these elements underlined.

Use case 01: *Order books*

1. The customer provides keywords for searching books.
 2. The system generates a list of books for sale that matches the keywords, including at least title, author, price, page count, publisher, ISBN, and cover image.
 3. The customer selects books from the list and indicates the desired quantity for each one.
 4. The system generates the order summary (title, author, quantity, unit price and subtotal for each book) and total value.
 5. The customer finishes the order.
-

Exception 3a: The quantity asked is higher than the quantity in stock for one or more books.

- 3a.1. The system informs the available quantities for each book that was ordered above stock
 - 3a.2. The user changes the desired quantities to meet the quantity in stock.
 - 3a.3. Advances to step 4.
-

Exception 5a: The customer is not identified yet.

- 5a.1. The customer provides valid identification.
 - 5a.2. Return to step 5.
-

Exception 5b: The customer does not have an account.

- 5b.1. The customer registers herself by providing username, password, and email address.
 - 5b.2. Return to step 5.
-

Exception 5c: No book was selected for buying.

- 5c.1. Return to step 1.
-

FIGURE 6.57 Use case with candidate concepts and attributes underlined.

Even if there is no preliminary conceptual model, this use case is enough to identify some concepts that belong to the system domain. If the preliminary conceptual model already exists, then the use case helps to refine the model, indicating new concepts, new attributes, or structure changes. For the example, we assume that there is already a preliminary conceptual model, which is depicted in [Figure 6.58](#).

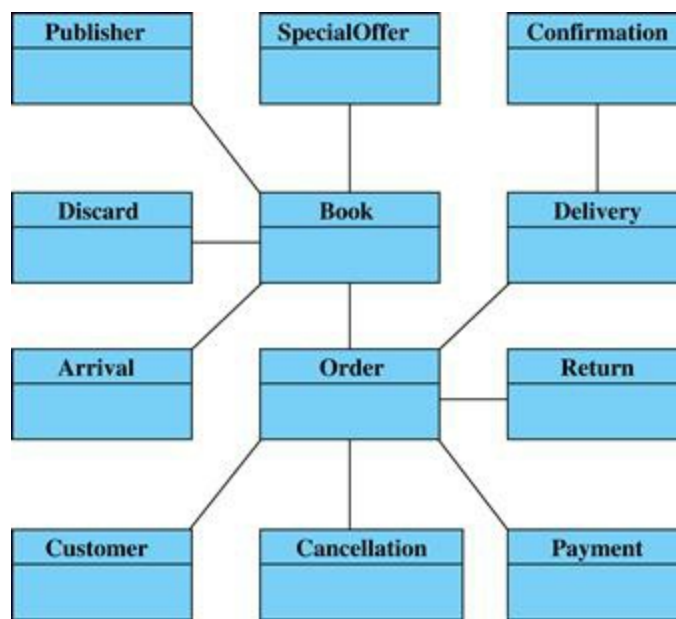


FIGURE 6.58 Preliminary conceptual model for the current example.

The terms identified in the use case text of [Figure 6.57](#) are, in order of appearance:

- *Customer*: A class that is already in the conceptual model.
- *Keyword*: Keywords are used for searching; they are not considered at this point attributes of complex concepts on their own (although they could be if search history or preferences have to be recorded).
- *Book*: A class that is already in the conceptual model.
- *Sale*: In the context of that use case and at this point of analysis “sale” and “order” could be considered synonyms. Thus, this is a class that is already in the conceptual model.
- *Title, author, price, and page count*: Attributes of *Book*. *Author* could still be considered a complex concept if the information about authors is more complex than a single name.
- *Publisher*: A class that is already in the model. The use case probably is referring to the publisher’s name.
- *ISBN and cover image*: Attributes of *Book*.
- *Desired quantity*: Cannot be considered an attribute of *Book* because it varies depending on the order. It cannot be considered an attribute of the order either because it varies depending on the book (there may be many books in an order). The desired quantity is, therefore, an attribute of the association between a book and an order, represented by an intermediary concept or by an association class called *Item*.
- *Order summary*: This is referring to an *Order*; it does not seem to be a new concept.
- *Title, author (or author’s name), quantity, unit price, and subtotal*: Except for *subtotal* all attributes have been already mentioned. *Subtotal* may be a derived attribute of an *Item*. *Unit price* is equivalent to *price*, which is already considered an attribute of *Book*. However, it may in fact also be an attribute of *Item* as well.
- *Total value*: Derived attribute of *Order*.
- *Finishes the order*: This indicates an action that represents a change in the state of an order. Maybe an attribute would be sufficient to represent this state, but more

analysis would be advisable in order to see if finishing an order is not a more complex operation (which is indeed it is).

- *Quantity requested*: Same thing as *quantity*.
- *Quantity in stock*: Probably a new attribute of *Book*.
- *Available quantities*: Synonym of *quantity in stock*.
- *Identified*: Could this be a state of a customer? Maybe not. It refers to whether the user is logged in or not.
- *Valid identification*: This belongs to the security domain. However, the conceptual model probably may have a way to identify customers. Usually ID numbers such as the Social Security number are used for that purpose.
- *Account*: This again has to do with login.
- *Username, password, and email address*: Depending on the kind of security framework that is used (or not) these may be attributes of a customer (or attributes of a *user* that is attached to a customer), or they may even be completely addressed by the security framework and ignored in the conceptual model. For now they are left as security issues, and not included in the conceptual model.

From the information collected above, the preliminary conceptual model may be refined, and it would become similar to the one shown in [Figure 6.59](#), where new classes and attributes are highlighted.

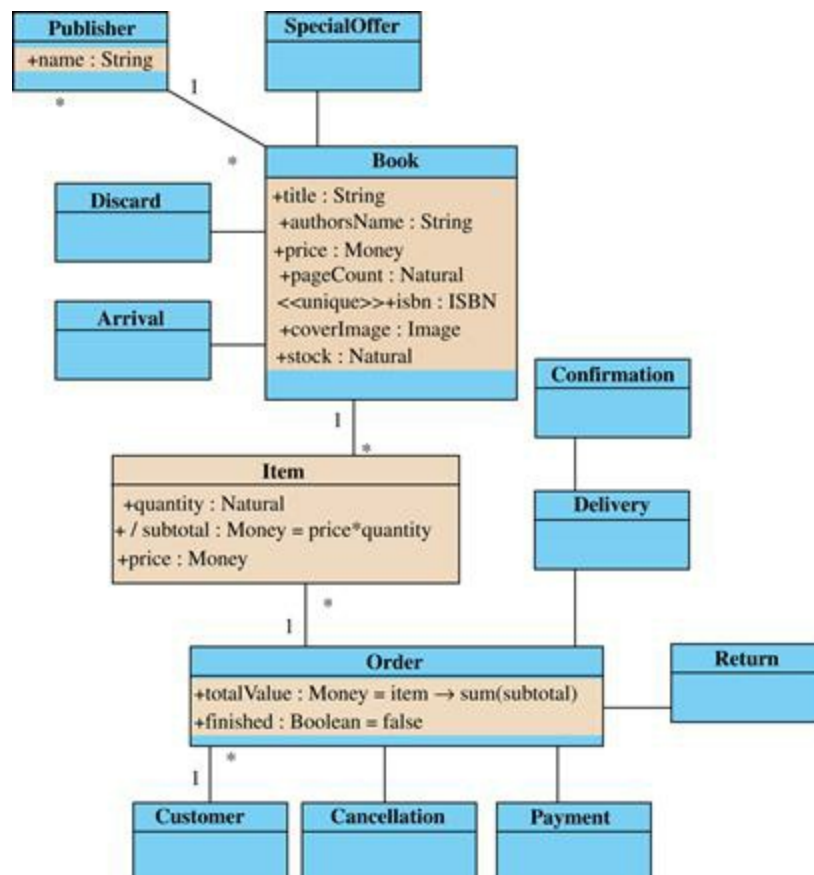


FIGURE 6.59 The second version of the conceptual model, refined after the analysis of use case 01: *Order books*.

Note that a single use case added significant new information to the model. This is no

coincidence, because that use case was considered the most important in the system and thus the first to be detailed in iterations. Its complexity has a lot to contribute to the comprehension of the information structure that the system should manage.

Not every association in [Figure 6.59](#) had its multiplicities defined. The UML default for undefined role multiplicity is 1, but the situation in [Figure 6.59](#) is that associations with no multiplicity defined were simply not analyzed up to this point and are considered “to be defined.”

6.8.2 Dependent and independent concepts

A concept is *dependent* from others if it needs to be associated to them in order to make sense, that is, to represent information that is minimally meaningful.¹⁴ Looking at [Figure 6.59](#), it can be seen that an order would not make sense unless it was associated to a customer and to a set of items. Without those associations, the concept of *Order* would not make sense. The order also has other associations in that figure, but they are optional.

A concept is *independent* if it has a meaning even without being associated to others.¹⁵ For example, the *Publisher* concept may be understood just from its attributes, without being associated to other concepts. The existing associations for a publisher are optional. Thus, the *Publisher* concept, in that sense, is independent.

But, what purpose does this distinction serve? It was discussed in [Chapter 5](#) that only the simplest concepts may be managed by CRUD use cases. Dependent concepts should not be considered simple in principle, while independent concepts may be simple in principle.

In [Figure 6.59](#), almost all the concepts are dependent. For example, a delivery depends on an order, a confirmation depends on a delivery, and so on. Only *Publisher*, *Book*, and *Customer* are independent concepts and candidates to be managed by CRUD use cases.

Concepts that are managed by CRUD use cases are usually those that can be accessed directly when information is being retrieved from a system. Although adding the façade-controller class to the model could wait for design modeling activities ([Chapter 9](#)), it can be added now to indicate which concepts are independent ones. In principle, only independent concepts have mandatory links to the controller class. Being linked to the controller class means that the instances of the class may be accessed directly. For example, an instance of *Book* may be accessed directly by its ISBN. However, an instance of *Item* is not accessible directly. To access an *Item* one has to navigate from a *Book* or an *Order*. Thus, the *Item* is not linked to the façade-control class.

Although *Book* is an independent class, it was not linked to the controller because it has a mandatory association to another independent class: the *Publisher*. In this case, *any* book may be accessed by looking at the set of books of the publishers. If later it is discovered that books must be accessed directly, independently from its publisher or that any other concept needs this kind of access, then (derived) associations to the controller may be created as needed, but they are not created by default for every concept, to avoid increasing coupling among classes.

There are, therefore, two main paths to reach information in the model of [Figure 6.60](#): from a publisher, and from a customer. All other concepts are reached by starting at one of those. Later, as mentioned, information needs may require defining other paths. In [Chapter 9](#), we see that an order must also be accessed directly, because a number of system operations need to obtain an order by its number or date independently of who the customer is or what books are in it.

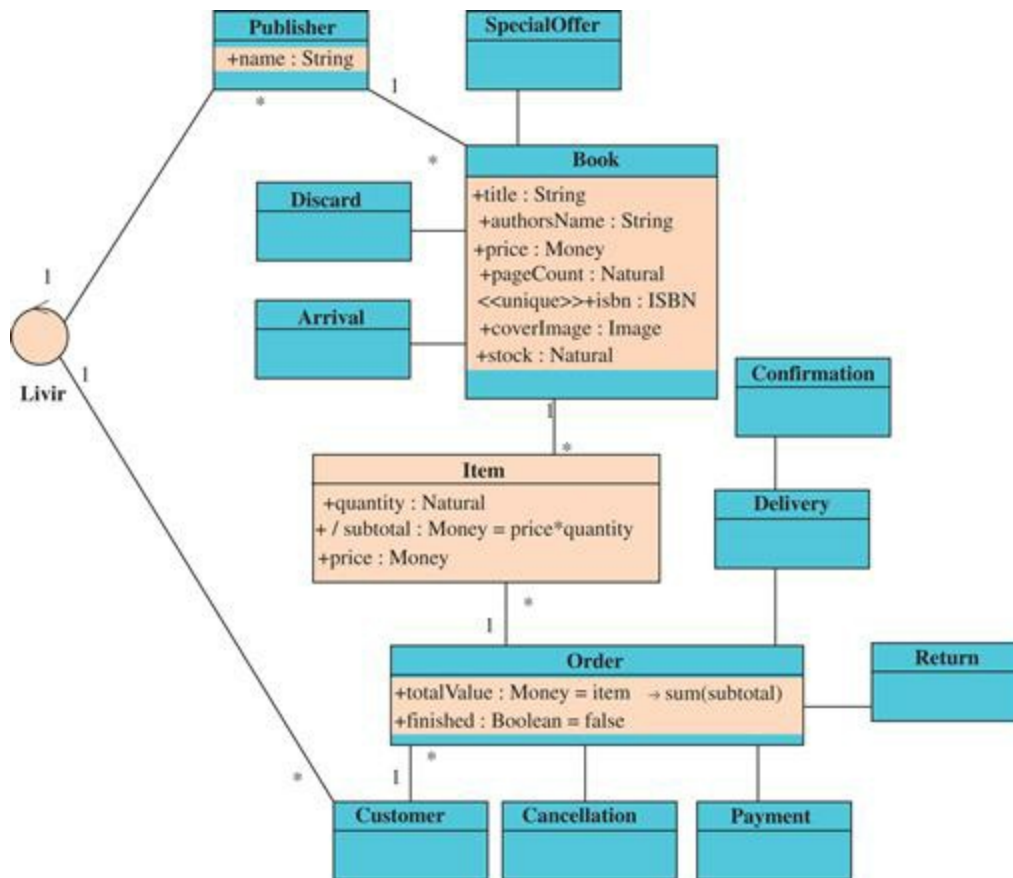


FIGURE 6.60 Refined conceptual model with a façade controller.

6.8.3 How to find associations

If the information related to concepts and attributes is easily found in the use case texts, the associations are not. As mentioned before, use cases indicate many operations that transform information, but one must seek for clues about associations.

So, how do we find associations between complex concepts? There are at least two rules that can be used:

- *Dependent concepts* (such as *Order*, *Payment*, *Cancellation*, etc.) should be necessarily associated to the concepts that complement them (which may be dependent or independent).
- *Associative information*, that is, associations that are not mandatory, but that complement the information in the model (such as “a person may own a car”), should be represented by associations.

Sometimes, associative information appears in the use case texts as concepts. For example, an order could be an association between a customer and books), but it is sufficiently complex to be represented as a concept. As an order is a dependent concept, there are associations between those concepts and their complements.

When one states that a person may own a car, that information should be represented by an association. Many professionals who work with programming but not modeling could produce models similar to the one presented in [Figure 6.61](#). However, this is unsuitable, because (first) attributes should not have concepts as type, and (second) the model is disguising a real association.

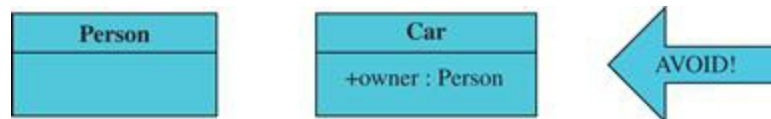


FIGURE 6.61 An attribute disguising an association.

It is also unsuitable to use foreign keys ([Date, 1982](#)), as shown in [Figure 6.62](#), because that also disguises associations. This mistake is commonly made by modelers used to design relational databases. Foreign keys must not be used in conceptual modeling because they are an implementation technique.



FIGURE 6.62 An association disguised as a foreign key attribute.

Thus, when two complex concepts are related to each other, the construct to be used is the association ([Figure 6.63](#)).



FIGURE 6.63 Suitable representation for the situation shown in [Figures 6.61](#) and [6.62](#).

The example of [Figure 6.63](#), contrary to the others, respects a *cohesion* rule that states that a concept should only contain *its own properties*. As the ID is an attribute of the owner, it should not be represented as an attribute of a car (as in [Figure 6.62](#)). Another reason to have visible nondisguised associations is more practical: it is easier to see which objects have references to others. Later ([Chapter 9](#)), we discuss how these *visibility* lines may be used for designing effective and good quality object-oriented code.

6.8.4 Example of iterative construction of the conceptual

model

It was mentioned before that the Unified Process is use-case driven and architecture centered. But, what does that mean in practice? We have seen already how to identify and detail use cases, and we have also seen that they can be used as a source of requirements for refining the conceptual model, which is the basis for the design class diagram, an important part of the system architecture. [Section 6.8.1](#) showed how the preliminary conceptual model could be refined with information from the first detailed use case (*Order books*).

Applying that process again, simulating a second iteration for use case *02 Pay order*, new concepts and attributes are discovered ([Figure 6.64](#)).

Use case 02: *Pay order*

1. The system generates the pending order summary (title, author, quantity, unit price and subtotal for each book), as well as the total price of the order, and a list of registered delivery addresses for the customer.
2. The customer selects a delivery address.
3. The system generates the delivery fee, and the date scheduled for arrival, and a list of credit cards already registered for the customer (flag and last four digits of credit card number).
4. The customer selects one of her credit cards for payment.
5. The system sends to the respective credit card operator the following data: card number, owner's name, validity, security code, purchase total value, code of the store.
6. The credit card operator approves the sale by sending an authorization code.
7. The system informs the customer that the sale was approved and the tracking number of the delivered package.

FIGURE 6.64 Main flow of use case *02 Pay for books* with candidate concepts and attributes identified.

The elements identified in this use case are:

- *Delivery addresses*: As a customer may have more than one delivery address and as addresses are used to calculate delivery fees, they may not be represented by a simple attribute, but instead are represented by a class.
- The second occurrence of *delivery address* refers to the specific address associated to the current order. Thus, a new association is found between an order and one address.
- *Delivery fee*: Attribute of the order. But from where is this value obtained? If requirements are further analyzed it would be discovered that different cities have different fees that depend fundamentally on the distance from the city to the company headquarters. Thus a new class is created (*City*), which is associated to *Address*, and the default fee for each city is defined as an attribute of that class.
- *Credit cards*: New concept. Should be associated to a customer.

- *Flag*: Probably not an attribute, but a complex concept associated to a credit card.
- *Card for payment*: An association between an order and a customer's credit card.
- *Credit card operator*: It is in fact an external actor; its data may be associated to the credit card flag.
- *Card number, owner's name, validity, security code*: Attributes of a credit card.
- *Purchase total value*: Same as *order total value*.
- *Code of the store*: Not an attribute of the bookstore, as might be thought. This code identifies the bookstore for each card operator. As different operators may provide different codes, this attribute should be represented in the credit card flag concept, unless a network of stores is being modeled.
- *Approval*: This happens when an authorization code is received (see next item).
- *Authorization code*: Can be thought of as an attribute of the order. But it may only exist when a credit card is already linked to the order. Thus, it would be better placed as an attribute of the association between credit card and order.
- *Tracking number of the delivered package*: May be an attribute of the delivery.

The new elements are highlighted in [Figure 6.65](#), which is the third version of the conceptual model for the *Livir* example. Notice that there are still some concepts with no attributes and some associations are left with undefined multiplicity. This is because the use cases analyzed to this point did not produce information to refine those elements. They would be refined later when other use cases are expanded and their information incorporated in the conceptual model.

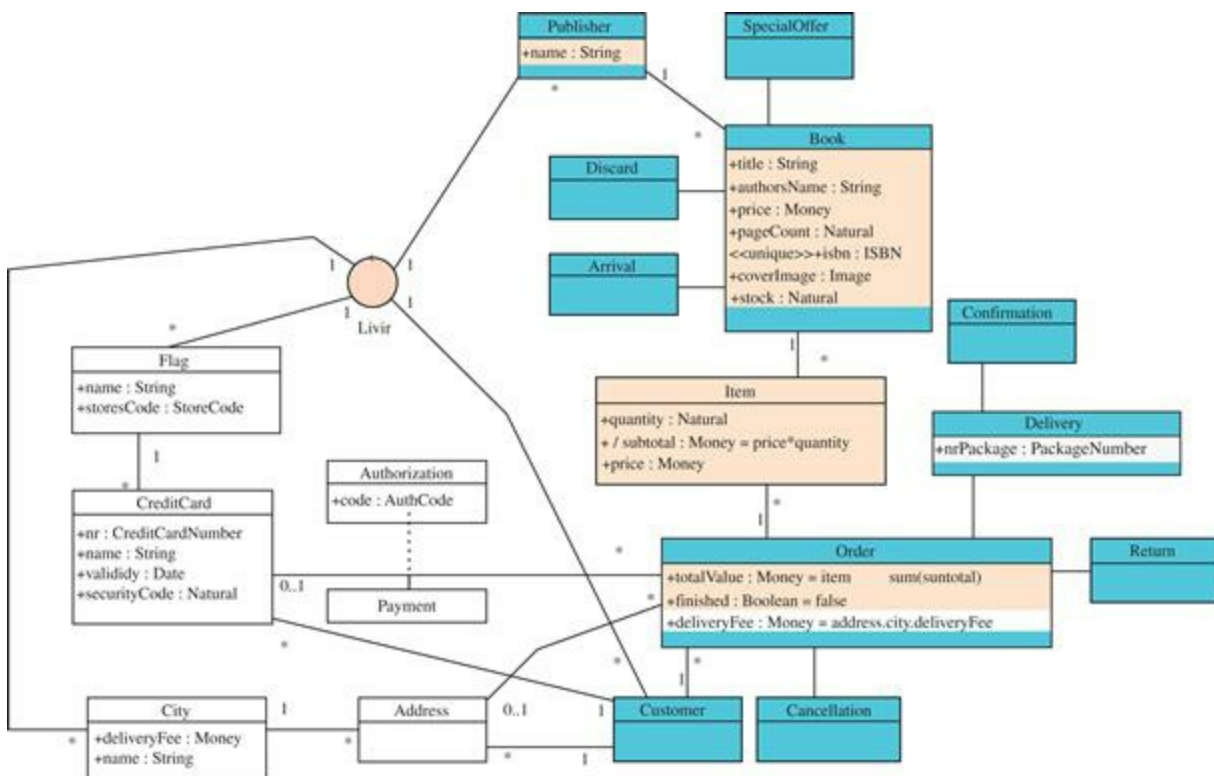


FIGURE 6.65 Conceptual model refined with information from the main flow of the use case of the second iteration: 02 *Pay order*.

Observe that just the *main flow* of this use case was used in the example. Alternate

flows of this use case still need to be analyzed and included. Recently, [Jacobson et al. \(2011\)](#) have proposed the *use case 2.0* technology, which specifically suggests that a use case should be addressed in *slices*, and not as an indivisible whole.

At least two invariants still should be added to the diagram of [Figure 6.65](#) in order to avoid the existence of inconsistent information: the address of the order must be one of the addresses of the order's customer and the credit card used to pay the order must be one of the credit cards of the customer. Those invariants may be defined in the *Order* class as

```
Context Order
```

```
inv:  
  
self.customer.address->includes(self.address)  
  
and  
  
self.customer.creditCard->includes(self.creditCard)
```

The OCL predicate *includes* verifies if the object received as an argument is included in the set. It is equivalent to the math operator " \supseteq ". For example, *self.customer.address->includes (self.address)* means *self.customer.address \supseteq self.address*, that is, *self.customer.address* contains *self.address*. It is assumed that if the element received as an argument is the empty set, then the expression is true. Thus, if the delivery address and credit card were not yet defined, then the invariant is not being violated.

The analysis of the second use case has incremented the conceptual model with a significant number of new classes and attributes. The idea is that the quantity of novelties decreases as the iterations advance, because, in the end, only CRUDs o elements already studied and reports with information that should already be in the system will be addressed. It is not expected that they will bring any new information to the conceptual model.

Observe that inverting that order would leave for the final iterations the insertion of many classes and attributes, and this would require a lot of rework in the design, and some architecture refactoring would possibly be needed in order to accommodate those needs. That is why it is so fundamental to initiate the project with the most complex use cases and only later incorporate the simpler ones into the architecture.

6.9 The process so far

	Inception	Elaboration
Business Modeling	<p>Build a general view of the system:</p> <ul style="list-style-type: none"> • Build a business use case diagram and determine the automation scope for the project. • Build preliminary activity diagrams for business use cases. • Build preliminary state machine diagrams for key business objects. 	
Requirements	<p>Prepare the system use case diagram (functional requirements):</p> <ul style="list-style-type: none"> • Identify the system actors from the business use case model. • Identify the system use cases from the business use case model, and the activity and state machine diagrams from business modeling. <p>Identify nonfunctional requirements as use case annotations:</p> <ul style="list-style-type: none"> • Identify the main business rules associated to use cases. • Identify the main quality issues associated to use cases. <p>Identify supplementary requirements.</p>	<p>Detail requirements by expanding use cases:</p> <ul style="list-style-type: none"> • Identify the main flow. • Identify alternate flows: variants and exception handlers.
Analysis and Design	<p>Prepare the preliminary conceptual model by observing system use cases and the concepts needed by them.</p>	<p>Elaborate the system sequence diagrams:</p> <ul style="list-style-type: none"> • Represent the main flow of a use case as a system sequence diagram. • Represent the system commands and queries using stateful or stateless strategies. • Complete the system sequence diagrams with alternate flows.

(Continued)

	Inception	Elaboration
Analysis and Design		Refine the conceptual model: <ul style="list-style-type: none"> • Identify concepts, attributes, and associations in the text of expanded use cases. • Detail attributes and associations with stereotypes, multiplicity, and constraints as needed. • Organize the model by using inheritance, association classes, and temporal specifications. • Add invariants as needed.
Implementation		
Test		
Project Management	Estimate total effort, ideal calendar time and average team size for the project. Estimate the duration and quantity of iterations for each phase. Prepare the phase plan and iteration plan for the first iteration.	

6.10 Questions

1. Try to imagine a real world situation where the right structure to be used is a *sequence*. Remember that a real sequence allows elements to be repeated and assigns position to elements. Try not to get fooled by natural language use of the word “list.” A “grocery list,” for example, is not a sequence, because elements do not repeat and their position in the list is irrelevant; a student roll call may be organized in alphabetical order, but again, repetition is not allowed and the order is again irrelevant although convenient. Thus, those are not real sequences.
2. What is the purpose of the controller class? What is its relation with dependent and independent concepts?
3. Why should an attribute not be typed with a class name? Why should an attribute not be used to reference another class?
4. Elaborate a list of concepts and subconcepts (subtypes of the original concept, as, for example, *dog* and *beagle*) from common sense. Then, try to identify the type of each relation: structural, associative, or temporal. Try to have at least one example of each type in your list.
5. What is the difference between a normal association, an aggregation, and a composition?
6. List some examples of data types that could be defined as primitive.

¹Not necessarily, because many analysts may chose natural language or other specification languages. However, as OCL is part of the UML package, it is advisable to use it.

²At this point, some readers that are also programmers could be worried about performance issues with the code that will be used to implement derived attributes. As they are by definition “calculated,” it seems a waste of processing time to

repeat those calculations if the original values of cost and price do not change. However, the readers can stay calm because code optimization mechanisms are available for derived attributes. For example, the value calculated for a derived attribute may be kept in the cache and recalculated only when one of its components is changed. For example, *profit* has to be recalculated only when the instance of *Book* changes the values of *price* or *cost*.

³The ISBN formation rule is explained in <http://www.isbn.org/standards/home/isbn/us/isbnqa.asp>.

⁴Some languages such as Smalltalk allow constants to change their value in runtime, but this is not usually the case and is not recommended for most applications.

⁵A design pattern that states that a class with a single instance may be accessible globally.

⁶See also an example in [Figure 6.18](#).

⁷In fact, in OCL, even a single object is considered a collection with a single element. This is equivalent to the natural notion of a set, but not to the mathematical notion, because in set theory, a set with one element is not the element. In OCL a set with one element and the element are the same thing.

⁸In the case of a set of people *p*, if someone wants to get the set of all ages, she could write *p.age*. However, the size of the set is not a property of the people; it is a property of the *set structure*. Thus, to obtain the size of the set the expression *p->size()* should be used. If it was written as *p.size*, then it would be representing the set of the sizes of each person, not the size of the set.

⁹An association may represent a composite aggregation (i.e., a whole/part relationship). Only binary associations can be aggregations. Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it. Note that a part can (where allowed) be removed from a composite before the composite is deleted, and thus not be deleted as part of the composite. Compositions may be linked in a directed acyclic graph with transitive deletion characteristics; that is, deleting an element in one part of the graph will also result in the deletion of all elements of the sub graph below that element" ([Object Management Group, 2011](#)).

¹⁰Just try to search Google for images with the keywords "UML" and "inheritance." You will see lots of examples of bad use of inheritance, as, for instance, *Person* as a generalization of *Student* and *Professor*.

¹¹More specifically, an object is also an instance of all of its superclasses ([Section 6.6.1](#)). The idea is that the set of superclasses should not change after the instance is created.

¹²Methods are only considered when design classes are being modeled. Remember that conceptual classes do not have methods.

¹³The monotonic decreasing case can be considered here too (a transition that can only remove properties from an instance).

¹⁴There is a parallel between dependent concepts and transitive verbs. The sentence "He opened" does not make sense, because the verb needs a complement, as for example, "He opened the can."

¹⁵A parallel exists between independent concepts and intransitive verbs. The sentence "He slept" makes sense because the verb does not require a complement. However, even intransitive verbs may have supplemental information (for example, "He slept for three hours"), as independent concepts do.

CHAPTER 7

Conceptual Modeling

Patterns

Building a conceptual model is much more than placing concepts, associations, and attributes together. Frequently the model does not work not for being wrong, but for being too complex to maintain. Analysis patterns consist of solutions already tested that can resolve families of recurrent problems. When adequately used these patterns may reduce significantly the complexity of an otherwise naïve conceptual model. This chapter presents and sometimes reinterprets some patterns such as high cohesion, quantity, strategy, account, temporal, and others.

Keywords

Design pattern; analysis pattern; high cohesion; temporal pattern; conceptual modeling

Key Topics in this Chapter

- High cohesion
- Specification classes
- Quantity
- Measure
- Strategy
- Composite
- Organizational hierarchy
- Account/transaction
- Range
- Temporal patterns

7.1 Introduction to conceptual model patterns

Class diagrams that represent conceptual models almost always become more complex and hard to maintain than analysts and other team members would like. There exist techniques that reduce the complexity of such diagrams, and at the same time, improve their expressiveness. Naïve models may be unnecessarily complex.

These techniques are called *analysis patterns* and may be understood as a special case of *design patterns* (Gamma, Helm, Johnson, & Vlissides, 1995) that are applied specifically to the conceptual model. Most patterns presented in this chapter were first described by Fowler (2003).

Analysis or design patterns are not rules that must be obeyed, but recommendations

based on previous experience. It is the responsibility of the analyst to decide when to apply a given pattern in the models.

7.2 High cohesion¹

High cohesion is so important for object-oriented modeling that it is considered more a principle or guideline axiom than a pattern itself. A concept with high cohesion is more stable and reusable than a concept with low cohesion, which can become rapidly confusing and hard to maintain. Most systems could have all their information represented in a single table, but that would be an extreme worst case of low cohesion, and it would not be practical.

Earlier, we mentioned that concepts should not have attributes that belong to other concepts (e.g., a car should not have the owner's ID as one of its attributes). Attributes also must not have class names or data structures (set, list, array, etc.) as data types because all those situations are evidences of low cohesion (a class with that kind of attribute is probably representing more than one single concept). For example, an instance of *Order* should not have an *itemsList* attribute because items have their own attributes and associations; they should appear related to *Order* by a 1 to * association.

In the example of [Figure 6.65](#), when it was discovered that a customer could have more than one address, instead of the naïve solution of creating attributes like *address1*, *address2*, etc., or even *address:Array of Strings*, we created a new concept (*Address*), which was associated to the customer. I imagine how hard it would be to deal with information such as *deliveryFee* if the addresses were modeled as an array of strings.

Concepts and attributes should have a simple structure with high cohesion elements. Evidence of low cohesion includes scenarios where some attributes of a class may be null depending on the value of other attributes. If complex constraints (invariants) are necessary to keep a concept consistent, this is equivalent to using duct tape to keep the pieces of a broken pot together.

In [Figure 7.1](#), the attributes *paidValue* and *paymentDate* are mutually dependent: both are null or both are not null. An invariant must be added to the class to avoid the situation where one of them is null while the other is not, in order to avoid inconsistent instances. Invariants are good to represent business rules that must always be valid. But when an invariant just relates two attributes of a class, it is a strong clue of lack of cohesion.

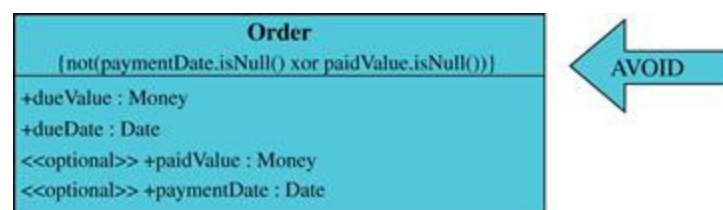


FIGURE 7.1 A class with low cohesion because attributes depend on each other.

A better way to model this situation is shown in [Figure 7.2](#), where the concepts *Order* and *Payment* appear separated but with high cohesion. In this case, there are no optional attributes depending on each other and *no need for constraints*.

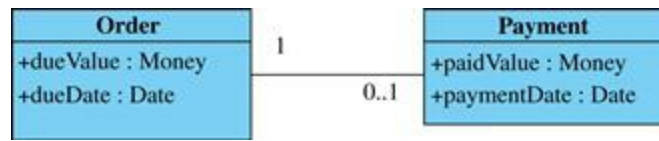


FIGURE 7.2 A modeling solution with high-cohesion classes.

Another potential problem related to low cohesion is the existence of groups of strongly correlated attributes, as seen in [Figure 7.3](#) where it can be observed that groups of attributes have stronger relations inside the group, such as *street*, *number*, *city*, and *state*, which compose an address, or *ddd* (distance direct dialing) and *phone*, which are part of a complete phone number, or *passportNumber* and *issuingCountry*, which are part of passport information. The relation that exists inside each group appears implicitly on the order they are presented in the class. This is bad, because if the attributes were sorted in alphabetical order, for example, the weak structure of the groups would be messed up: *city*, *ddd*, *issuingCountry*, *name*, *number*, *passportNumber*, *phone*, *state*, and *street*. The understanding of a concept should not depend on the ways its attributes are sorted.

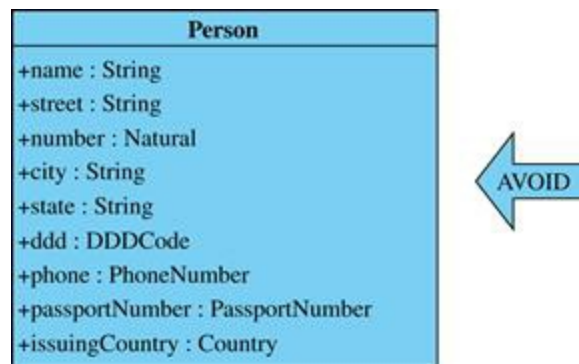


FIGURE 7.3 A class with low cohesion because of strongly related attributes.

A better solution to improve high cohesion in that model is shown in [Figure 7.4](#). It also opens a way to other modeling possibilities such as, for example, allowing a person to have more than one address, or more than one phone. In those cases, it is simply the case of changing the association multiplicity.

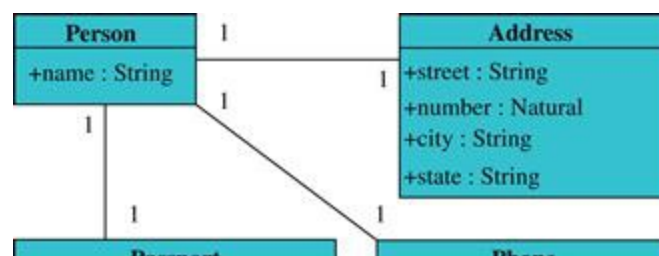




FIGURE 7.4 A solution to Figure 7.3 with high cohesion.

Yet another situation occurs when some attributes repeat their values for a group of instances. Figure 7.5 presents an example. In this figure, the attributes *customerName*, *customerId*, and *customerBirthDate* repeat the same values for different orders when the customer is the same.

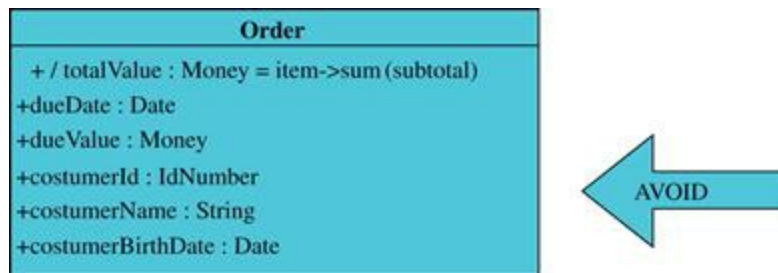


FIGURE 7.5 A class with low cohesion because it has attributes that repeat values in different instances.

This type of situation can be eliminated by splitting the concept into two associated concepts with better cohesion, as shown in Figure 7.6. Notice that even attributes' names are shortened with this choice of model.

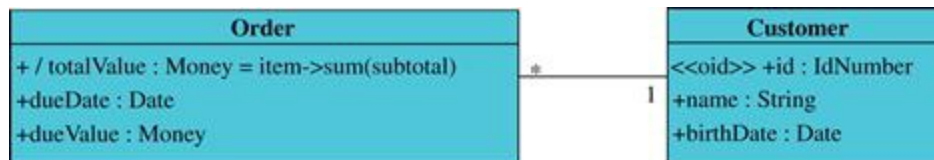


FIGURE 7.6 A model for the example of Figure 7.5 with high-cohesion classes.

7.3 Specification classes

A special case of low cohesion happens when an object is confused with its specification. That situation is very frequent: sometimes, products or physical items share a common specification, while their exemplars may have some differences among them.

One example of specification already discussed in the Livir example is the difference between the concept of a book as a *published title*, and a book as a *physical copy* of a published title. If both book concepts are not distinguished, lots of trouble will occur. In past examples the class *Book* refers to the published title, while *Item* stands for a set of physical copies. Thus, an instance of *Book* is a *specification* for a set of undistinguishable instances of *Item*.

Another possibility is to consider that each item is a single copy of a book. That would eliminate the need to include the *quantity* attribute in class *Item*, but it would create more

instances and associations for an order if the order includes more than one copy of the same book.

Depending on the information needs, it would be necessary to differentiate one copy from another. For example, if used books are to be sold, the conservation state should be kept for each copy. In this case, instead of a set of items, the *Book* class could be used as a specification for the class *Copy*, whose instances are single copies of a book.

The specification class *Book* should contain the attributes and associations that do not vary for different copies. For example, different copies of the same book have the same title, author, ISBN, base price, etc. However, conservation state and discount may be distinct for each copy. Thus, these attributes (and associations, if any) are placed in the *Copy* class, as shown in [Figure 7.7](#).



FIGURE 7.7 A class and its specification class.

It is possible for a class to have more than one specification. For example, the discount of the book could be predetermined depending on the conservation state of the book (an enumeration with values such as *new*, *used*, *damaged*, etc.). Thus, a copy, besides being specified by the published title, may also be specified by its state of conservation. Thus, *Conservation* would be a class with some instances that may specify physical copies of books, and define their discounts as shown in [Figure 7.8](#).

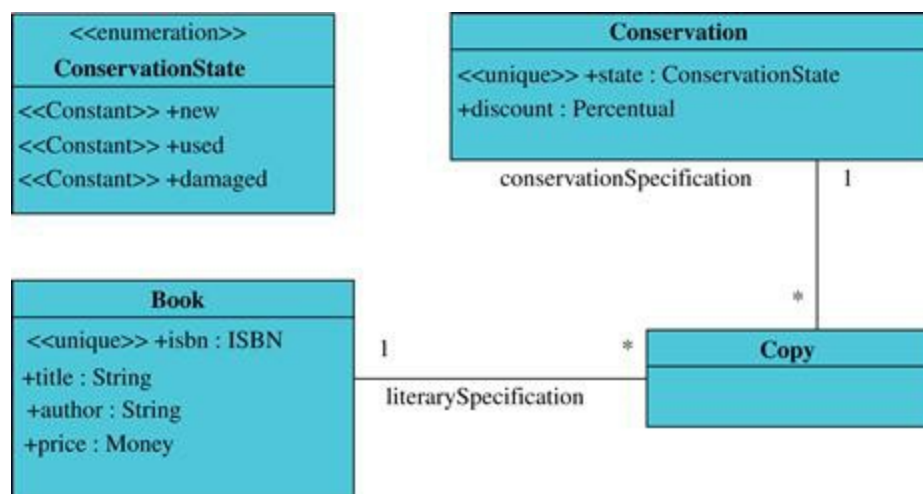


FIGURE 7.8 A class with two specification classes.

The instances of class *Conservation* have two attributes. The first is the conservation state, which is a value obtained from the enumeration *ConservationState*. There are

initially only three possible values: *new*, *used*, and *damaged*. The second attribute is the discount percentage that would be applied to a copy depending on its conservation state. As it is expected that books with the same conservation state have the same discount, then the attribute *state* in class *Conservation* must be unique and therefore marked with `<<unique>>`. Otherwise, different discounts could be associated to the same conservation state.

7.4 Quantity

Frequently the team faces the need to model quantities that are not merely numbers. For example, the weight of a book could be defined as 400. But 400 what? Grams? Pounds? Kilos? One solution is to define a specific type for the weight (*Pounds*, for example), and then keep using it consistently. The attribute then could be declared as *weight:Pounds*. But that demands that all books have their weight expressed in pounds.

In some cases, the system must be configurable to support different weight units. In some countries, grams and kilograms are used, while others use pounds, or even other units. If the attribute has a type that refers to a specific unit, then the system must suffer refactoring to accommodate new weight units.

However, the *quantity* pattern allows different unit systems to coexist and to be easily exchangeable. The pattern consists of creating a new primitive type *Quantity*, which has two attributes, as shown in [Figure 7.9](#).

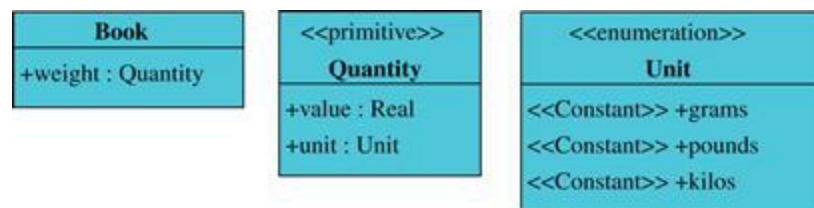


FIGURE 7.9 Definition and use of *Quantity*.

This way, the weight of each book is specified as a primitive quantity formed by a numeric *value* and a *unit*, whose type is an enumeration with possible values defined as *grams*, *pounds*, and *kilos*.

If a unit conversion is needed, then an option is to transform the enumeration *Unit* into a normal class, and associate a new class *Ratio* to two units, *origin* and *destination*, as shown in [Figure 7.10](#). When a quantity in the origin unit has to be converted to a quantity in the destination unit, then the value of the quantity associated to the origin unit is divided by the conversion ratio.

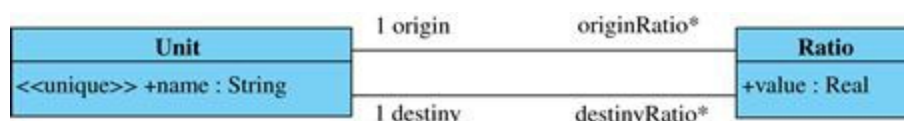


FIGURE 7.10 Unit with conversion ratio.

Thus, for example, an instance of *Unit* whose name is “grams” may be linked to an instance of *Ratio* as *origin*, which may be linked to another instance of *Unit*, named “kilos.” The *value* of that ratio should be 1000, because in order to convert grams into kilos, the original value in grams must be divided by 1000.

7.5 Measure

An evolution of the *quantity* pattern is the *measure* pattern, which can be used when a number of different measures over an object must be taken possibly at different times. For example, a person in observation at a hospital may have many measures taken from time to time: body temperature, blood pressure, blood glucose level, etc. Thousands of different measures could be taken, but usually just a few are actually taken for each patient. Therefore, to avoid creating a concept with thousands of attributes with most of them likely being null, a better option is to use the *measure* pattern as shown in [Figure 7.11](#).

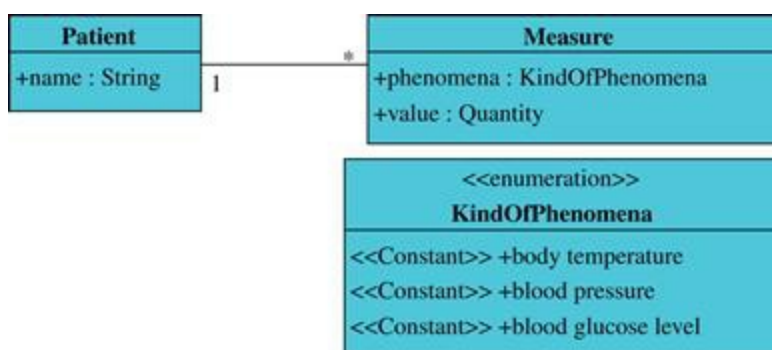


FIGURE 7.11 Definition and use of the *measure* pattern.

Thus, a patient may have a series of measures taken, each one assessing a different phenomena and presenting a value that corresponds to a quantity (following the *quantity* pattern).

It is still possible to make these patterns a little more sophisticated by adding attributes in the *Measure* class to indicate the instant of time at which the measure was taken, and also the validity of the measure. For example, the fact that a patient had a fever a couple of hours ago does not mean that she still has it. This evolution is explained in [Section 7.12.1](#).

Another evolution for this pattern is to add an attribute to define the *precision* or *accuracy* of the measure taken. For example, body temperature usually is measured with a precision of 0.1 degrees (Celsius or Fahrenheit).

7.6 Strategy

It was mentioned that one of the great requirements challenges is to manage their change. Transitory requirements must especially be accommodated in the system design

so that when they change the impact to the system, is minimized.

Some cases are relatively easy to address. For example, if it is already decided that the system must operate in different countries, the *quantity* pattern could be used to deal with currency and other measures that may vary from country to country.

But there are much more complicated situations. For example, the tax calculation procedure may vary a lot from country to country and even from state to state in some countries. There are some taxes that are calculated in relation to the profit, others with regard to the selling price, and so on. Methods for calculating taxes change from time to time, and in some countries new taxes are created every year. Systems must be prepared to cope with that, but the changes are completely unpredictable. Just configuring parameters usually is not enough.

Another example is the bookstore's discount policy. Current policy could be applying 10% off on orders above 100 dollars, for example. However, after some time, creative and unpredictable policies may be defined by the sales department, such as:

- Give a free book with a value up to 50 dollars for orders above 300 dollars.
- Give 20% off for up to two books on the customer's birthday.
- Give 5% off on horror books on Friday the 13th.
- The customer spins a wheel and it defines the discount she receives, from 1% to 10%.

Moreover, it would be possible to combine policies, if applicable, or choose the one that gives the higher discount.

The *strategy* pattern suggests that in these cases, the procedure (e.g., calculate taxes or discount) must be separated from the data to which it applies. In other words, if a discount is applied to an order, then the discount should not merely be a method implemented in the *Order* class. The discount should be defined as something easier to change. The solution proposed by the *strategy* pattern is to create an abstract class associated to the order. This abstract class may have concrete subclasses that represent concrete discount policies, as shown in [Figure 7.12](#).

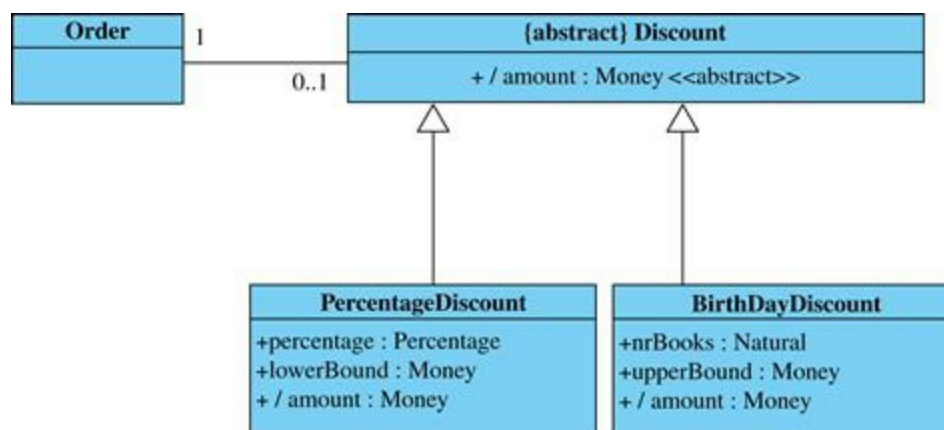


FIGURE 7.12 Strategy pattern.

Thus, each instance of *Order* is associated to an instance of one of the subclasses of the *Discount* strategy.

UML does not allow an attribute to be declared *{abstract}*; only classes and methods

may be declared so. However, as derived attributes will be implemented by methods eventually, the idea of an *abstract derived attribute* stereotyped with `<<abstract>>` as shown in [Figure 7.12](#) is coherent to the UML spirit.

Thus, the abstract class *Discount* implements an abstract derived attribute *amount*, which is implemented in a concrete way in each of its subclasses. If some of the strategies need data from the customer or from the order to calculate the discount, that data can be accessed through the associations from the discount abstract class to the classes that own the information.

This pattern minimizes two problems related to requirements change. First, it keeps the discount strategy applied when the order was issued, even if strategies change in future. Second, if new strategies are created later, it is sufficient to implement one or more subclasses for *Discount*. This does not affect the old strategies, or the old orders.

7.7 Composite²

As mentioned in [Section 7.6](#), sometimes different strategies must be grouped and a combination or selection of them must be applied. The combination of different discount strategies can be obtained, for example, by the sum of the values obtained for the derived attribute *amount* for each individual strategy, or by choosing the highest or lowest values among them. The choice of one or another combination produces different aggregated strategies.

In [Figure 7.13](#) there are two composite strategies shown. *SumDiscount* defines *amount* as the sum of the values returned by each component discount:

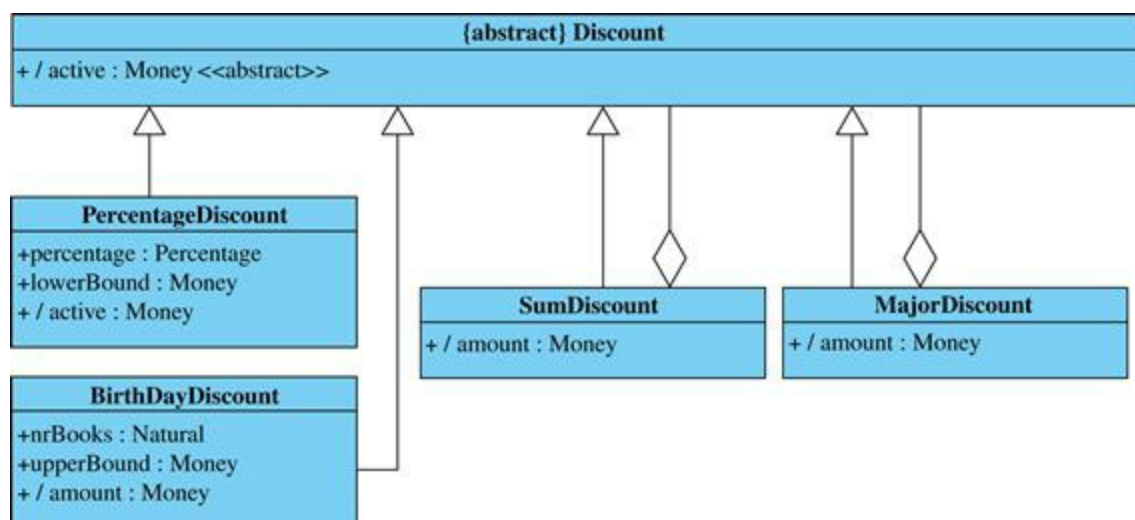


FIGURE 7.13 Example of the *composite* pattern.

Context `SumDiscount::amount:Money`

```
derive:
```

```
self.discount->sum(aDiscount|aDiscount.amount)
```

MajorDiscount selects the highest discount among each aggregated subclass:

```
Context MajorDiscount::amount:Money
```

```
derive:  
self.discount->maxElement(aDiscount|aDiscount.amount)
```

The current version of OCL does not define a function *maxElement* to take the highest element of a collection, but it could be defined as a new operation. Another solution is defining the expression by using functions already provided by the language, such as

```
Context MajorDiscount::discountAmount:Money
```

```
derive:  
self.discount->sortedBy(aDiscount|aDiscount.amount)->last()
```

The reader could find that it is not necessary to produce a sorted version of an entire list just to obtain its highest element. But remember that this is specification, not implementation; it is the final result that counts. Anyway, defining a new operation in this case, if possible, would be better because the meaning of the specification would be clearer.

7.8 Organizational hierarchy

Another common situation consists of the need to represent organizational hierarchies. It is commonplace, for example, to represent the administrative organization of a company as a composition hierarchy as shown in [Figure 7.14](#).

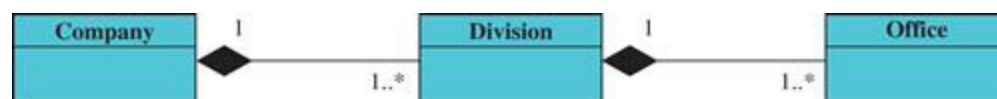


FIGURE 7.14 Straight representation of a company's organizational structure by using classes for the different levels of composition.

However, hierarchies like this usually do not behave well. First of all, this kind of organization is not followed by every company: other companies may have different levels or different compositions among them. Second, the company structure may change over time. New divisions or offices may be created, some may be joined, and others split. Last, but not least, different views of an organization may exist at the same time; for example, the finance department may have a different view of the same organization.

How to deal with all that complexity in a conceptual model? By using the *organizational hierarchy* pattern.

The solution consists of not considering the different levels of an organization as concepts, but as instances of a single concept, as shown in [Figure 7.15](#).



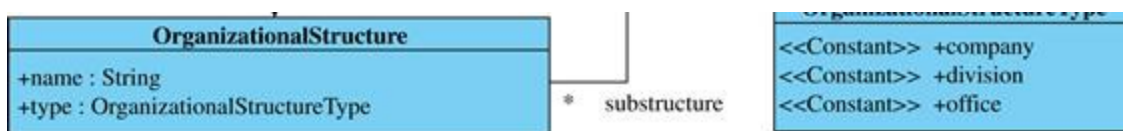


FIGURE 7.15 Application of the organizational hierarchy pattern.

In this way, flexibility is gained related to dealing simultaneously with organizational structures from different companies. Also, eventual changes in the organizational structure, such as adding more levels, or changing dependencies, may be easier to accomplish.

This pattern has some variations that may be used if concurrent hierarchies, superseding structures, or equivalent structures should also be represented. Those variations are discussed in the following sections.

7.9 Object joining

One of the team's assumptions that usually fails is that users will do everything right. That is not always the case. User failure (and sometimes sabotage) is still frequent, despite the efforts to build failsafe interfaces. A user could, for example, register a new publisher in the system, and later discover that it was already registered. Entering an incorrect code or the impossibility of having a unique identification for a real world object may cause this situation. The result is that two objects are registered in the system, both representing the same publisher, and each of them may possibly have its own set of books associated, some repeated, some not.

The solution when this kind of error happens is to join the objects, usually copying one over the other. This operation can be performed directly within the database, but in some situations it may be so common that it would be necessary that the system be prepared to allow this possibility as a user feature.

Besides, it is not always an *error* that causes the need for a join. In some situations, there are objects considered equivalent by one group of people and not by another, and both situations must be represented simultaneously. In other cases, as in multiple view organizational hierarchies, it may be necessary to indicate that two structure elements in different views are equivalent, or that one succeeds another in time.

The following subsections present the main strategies to deal with these kinds of situations.

7.9.1 Copy and replace

The first strategy that comes to mind when it is necessary to join two objects consists of copying the data of one object over the second (*copy and replace*). The copy/replace command could be defined by a contract (see [Chapter 8](#)), and the team should define, for each attribute and each association, what must happen during the copy/replace command. Rules should be defined to decide if an attribute is going to be copied over the other, or if their values are to be added, or if the newest or highest of them must prevail,

etc. Regarding the associations, the team must decide what happens: if one replaces the other, or if their properties are added, and so on.

The registry of the date of the last inclusion or change of a concept may be useful for deciding which attribute must be kept in the case of a conflict. For example, if a registered customer makes a new record, and it is going to be resolved, then the most recent address must be kept, not the oldest one. On the other hand, all of her orders must be added to the resulting instance.

After performing the copy/replace command the instance that was copied must be destroyed, and any references to it must be redirected to the instance that received the data.

7.9.2 Superseding

Superseding is a technique that can be used when the original object must be kept and not destroyed. Superseding is applicable, for example, in the case of organizational structures that are succeeded in time. Suppose that the departments of marketing and sales are joined into a single department of *customer contact*; the original departments must be marked as no longer active, and a new department must be added as their successor. The superseding strategy may be implemented by a reflexive association, as shown in [Figure 7.16](#).

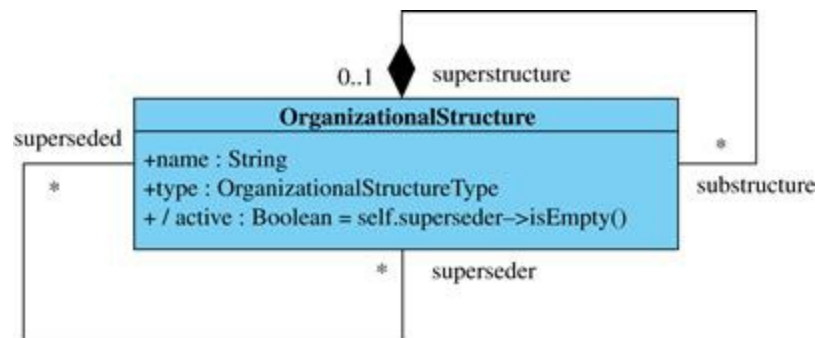


FIGURE 7.16 An example of the superseding strategy.

The derived attribute *active* indicates if the structure is active or if it has been superseded. It is true if the set of superseders is empty and false otherwise.

To maintain the original organizational structure even if it is not active anymore may be important for registry purposes. Someday someone could need to know how much a given department that does not exist anymore spent on toothpicks.

It would be useful to add an association class to the *superseded/supersede* association; its attributes could indicate, for example, the date when the superseding event taken place, or the context or view in which it must be considered. If that kind of information is necessary, the team should consider applying one of the temporal patterns ([Section 7.12](#)).

7.9.3 Essence/Appearance

Another situation that can still often happen is the existence of objects that are considered equivalent but must be kept as separated objects. This is not a registry error and it is not an object that supersedes another: this is *object equivalence*.

Object equivalence may be modeled by the use of an essence object that is associated to a set of equivalent objects. As opposed to *copy/replace*, the original objects are kept, and as opposed to *superseding* there is no active or superseded object: all associated objects are equivalent. [Figure 7.17](#) shows an example of a class (*Book*) that accepts that its members share a common essence.

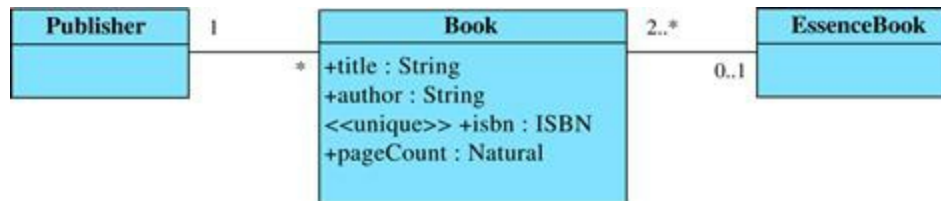


FIGURE 7.17 *Essence/Appearance* technique.

In the example, we consider that the same book in essence, especially a classic work, may be published by different publishers. Each publication is distinct, with a different ISBN and number of pages. Even the title and author's name may vary (proper name may be translated or abbreviated sometimes). But the essence of the text is the same. Some users may be interested in ordering Plato's *The Republic* from a specific publisher, and others may be interested in the book independently of the edition or publisher. Thus, the users interested in a specific publication are looking for the appearance (a given instance of *Book*), and the users interested in the text are looking for the book in essence – the publisher does not matter. The second group of users, when viewing any edition of *The Republic* must also be allowed to view other instances of *Book* linked to the same *EssenceBook* if they exist. In other words, if you look for `book` and if `book.essenceBook->notEmpty()`, then you will see `book.essenceBook.books`, which is a set of books with the same essence.

Not every *Book* must be linked to an *EssenceBook*, only those that participate in an equivalence class. Also, there must be at least two equivalent books to create an *EssenceBook*. This is why the association role from *EssenceBook* to *Book* is marked with `2..*`.

Objects are considered equivalent if they are linked to the same essence object. The essence object exists only to establish that equivalence; usually it will not have any other properties. Otherwise, maybe the *specification class* pattern could be used instead.

7.9.4 Undoing a join

Just when you think that the real world cannot be more complex, it becomes so. Therefore, if there is a possibility of joining objects, it is also possible that joined objects must be separated again. Once again, these commands may be performed directly in the database, or through well-planned operations available at the system's user interface.

To allow the joins of the *copy/replace* technique to be undone, a backup of the original objects must be kept, because the technique destroys one of the objects and disfigures the other. The *superseding* technique allows the join to be undone just by removing the association. In the case of *essence/appearance*, it is necessary to remove the association (and the essence object if less than two links to it remain).

However, in every case it is important to decide how to deal with eventual changes within the object that occurred while it was joined with the others.

A way to implement the possibility of undoing joins and any other operation is to use a variation of the temporal pattern (Section 7.12.3), which keeps track of former values of objects attributes and associations.

7.10 Account/Transaction

The *account/transaction* pattern is closely related to business, but it has wide applicability. It was mentioned before that books may be cataloged, ordered, received, delivered, discarded, etc. Such movements, as well as the financial transactions involved with them, give origin to concepts such as *Order*, *Delivery*, *Discard*, *Return*, etc., each one with its own attributes and associations.

However, it is possible to identify a common core for all these concepts and many more, which is constituted by a single and powerful pattern.

An *Account* is a concept that bears quantities of *something* (such as items, products, or money). An *Account* has a balance that usually consists of the sum of every deposit or withdrawal.

On the other hand, deposits and withdrawals are usually just entries of goods or money from an account to another. Thus, a *Transaction* consists of two entries: a deposit in one account and a withdrawal of the same value in another account. Figure 7.18 illustrates those classes.

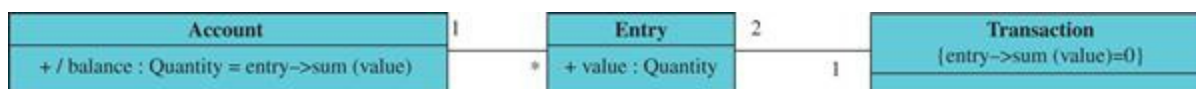


FIGURE 7.18 Classes for the account/transaction pattern.

A consistent transaction with two entries such as the one in Figure 7.18 needs two entries with the same absolute value but opposite signs. In other words, if a transaction takes 5 dollars from an account, it must necessarily deposit 5 dollars into another account; the withdrawal has a negative sign and the deposit a positive sign. Thus, the *Transaction* class needs the following invariant:

Context Transaction

```

inv:
  entry->sum (value)=0
  
```

This means that for each instance of *Transaction* the sum of the value of the two

associated entries must be zero.

The derived attribute *balance* of class *Account* is defined as the sum of every entry linked to that specific *Account*:

```
Context Account::balance
```

```
derive:
```

```
entry->sum(value)
```

Many situations related to the bookstore example could be modeled from a set of instances of class *Account*, such as:

- For each *publisher* there is an instance of *Account* from which books are “withdrawn,” that is, it is an *entry account*, and its balance becomes more negative as books are ordered from the supplier.
- There is an account for *orders expected*, which contains the books that were ordered from the supplier but that had not yet arrived.
- There is an account for *stock* containing the available books.
- There is an account for *sold books* containing books sold but not yet delivered.
- There is an account for *delivered books* containing books delivered but not yet confirmed by the customer.
- There is an account for *confirmed delivery* containing books delivered and confirmed by the customer. Its balance represents the whole set of books that the customer has bought from the bookstore.

In parallel to book transactions there are concomitant money transactions. There are receivable accounts, payable accounts, received accounts, paid accounts, investment, debts, values saved for paying taxes, etc.

Thus, many transactions of the bookstore example could be modeled as an instance of *Transaction*. For example:

- An *order* is a transaction that takes books from a *publisher* account, and deposits them into an *orders expected* account.
- The *arrival* of the books is a transaction that takes from the *orders expected* account and adds to the *stock* account.
- A *sale* is a transaction that takes from the *stock* account and adds to the *sold books* account.
- A *delivery* is a transaction that takes from the *sold books* account and adds to the *delivered books* account.
- A *return* is a transaction that takes from the *delivered books* account and adds to the *stock* account (if the books are in good shape).
- A *confirmation of delivery* is a transaction that takes from the *delivery* account and adds to the *confirmed delivery* exit account.

As an example, imagine a company that just started, where five instances³ of the *Account* class were created: *supplier*, *pendingOrders*, *stock*, *sold*, and *delivered*. [Figure 7.19](#) shows the initial state for those objects, when all accounts have a zero balance.

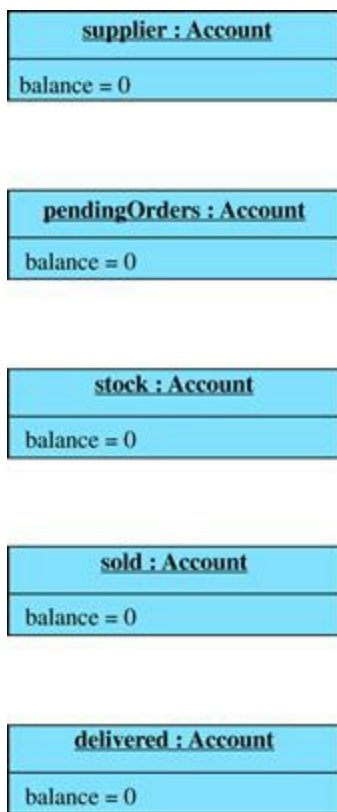


FIGURE 7.19 Initial state of an accounting system for products.

Figure 7.20 shows the resulting set of objects when a *purchase order* (instance of *Transaction*) is created. The order was issued for 50 books.

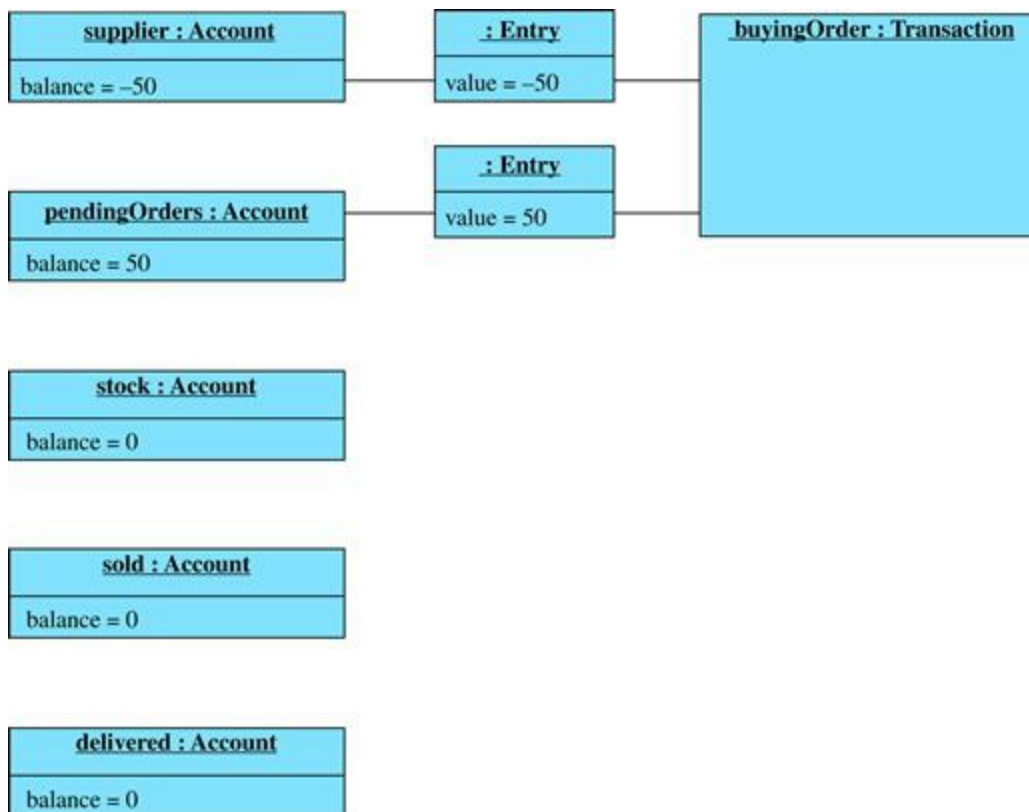


FIGURE 7.20 State of the accounts after a purchase order of 50 books is issued.

Figure 7.21 shows the state of the accounts if only 40 of the 50 ordered books arrive.

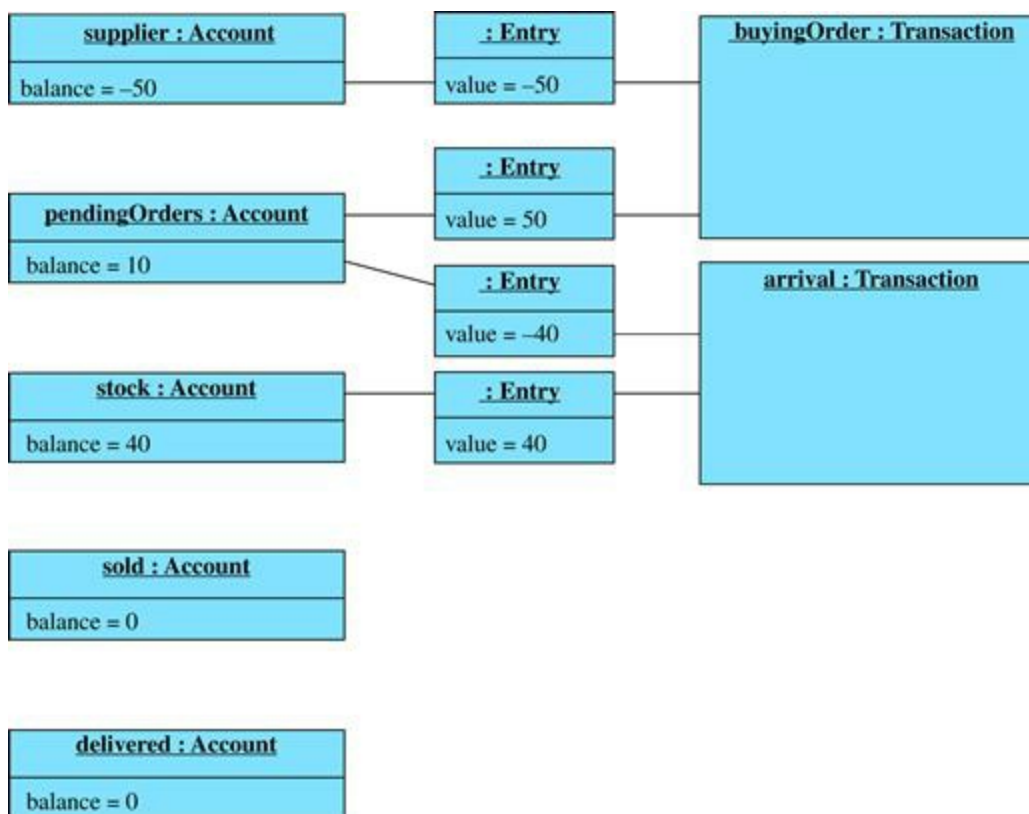
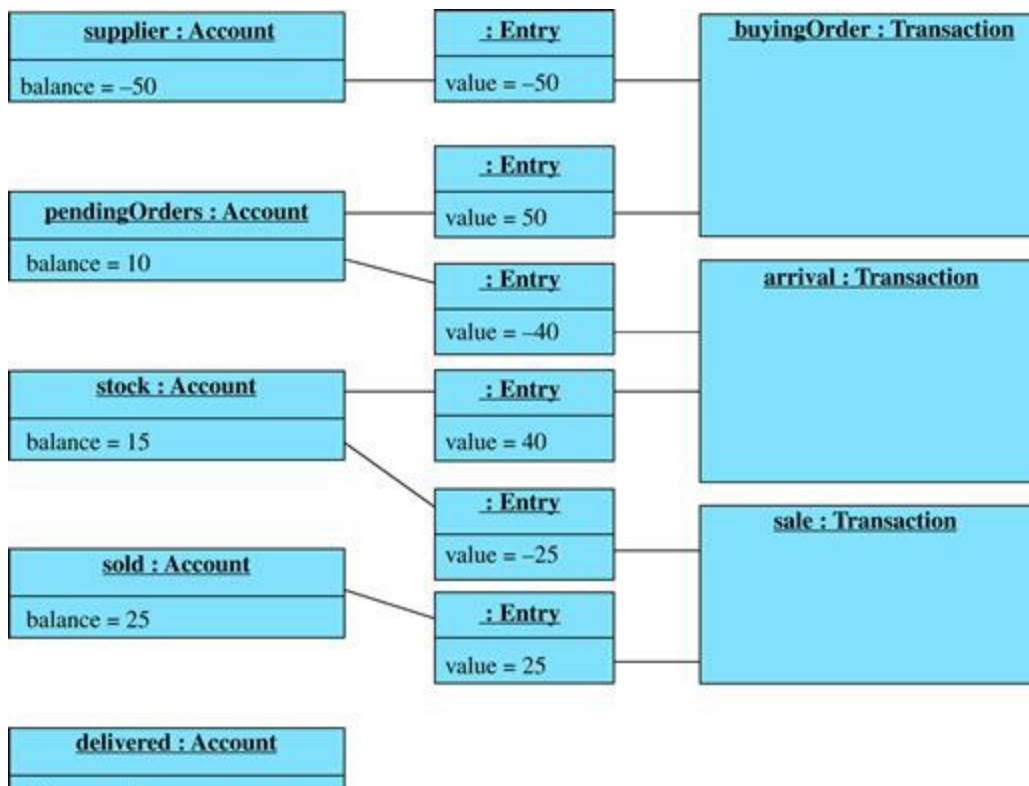


FIGURE 7.21 State of the accounts after 40 books arrive at the bookstore.

Figure 7.22 shows the state of the accounts after 25 books are sold.



balance = 0

FIGURE 7.22 State of the accounts after 25 books are sold.

And finally, [Figure 7.23](#) shows how the accounts look like after the 25 books sold are delivered.

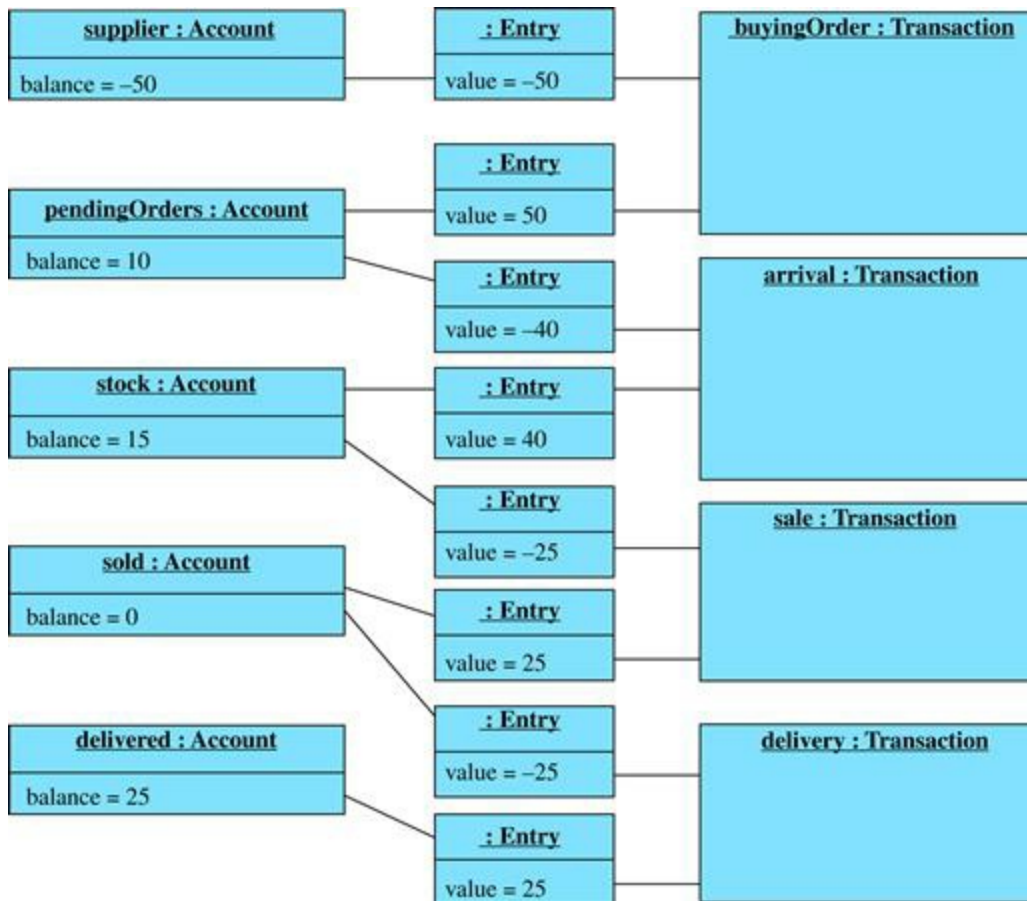


FIGURE 7.23 State of the accounts after the delivery of 25 books.

New transactions and accounts may be created dynamically (i.e., without having to recompile the system). For example, a new transaction may be created for returning books, which moves books from the *delivered* account back to the *stock* account. Also, a new transaction for discarding books could be created moving books from the *stock* or *sold* accounts to a new account for *discarded* books. Parallel financial transactions could be created as well by using *Account* and *Transaction*.

This pattern is very interesting as an example of how a simple powerful idea can deal with so many different situations. It has many variations and sophistications. For example, *multi-legged transactions* may be created if necessary, for moving from more than one account to a single account, or from a single account to more than one account, or even from more than one account to more than one account at the same time. To accomplish this variation, the multiplicity role from *Transaction* to *Entry* in [Figure 7.18](#) should be changed to 2..*.

Another important variation is the use of *memo entries*. For example, when money is received, taxes must be paid, but not necessarily immediately. Thus, when a transaction

moves money from a customer account to the bookstore balance account, a quantity of money equivalent to the taxes due could be registered into a tax memo account. When it is time to pay taxes, the balance of the memo account registers the amount to be paid.

Memo accounts and normal accounts could be distinguished. However, as they share the same definition, they were kept as instances of a single *Account* class in [Figure 7.24](#). On the other hand, memo entries are distinguished. A transaction must have at least two normal entries, but any number of memo entries (zero in most cases). Furthermore, only normal entries are subjected to the invariant of the *Transaction* class. Memo entries are ignored because they do not really move goods or money.

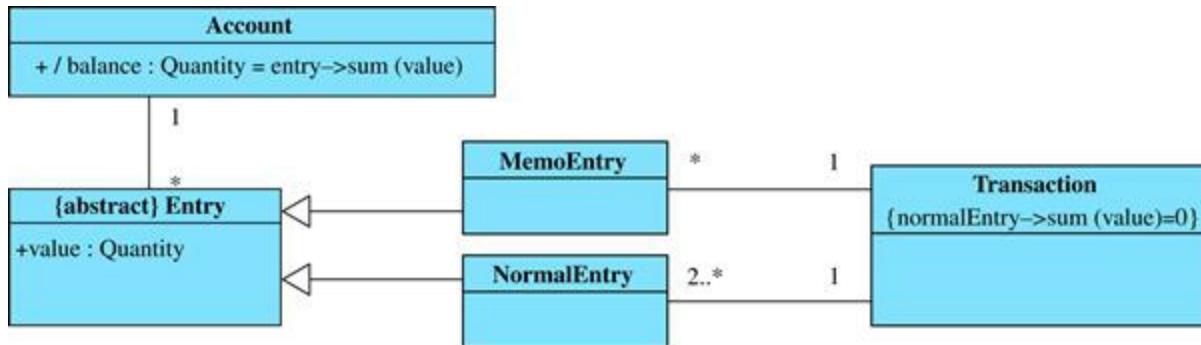


FIGURE 7.24 Evolution of Account/Transaction with memo entries.

Despite normal and memo accounts both having the same structure and behavior, it would be wise to clearly differentiate both kinds in order to avoid the system linking memo entries into normal accounts and vice versa.

7.11 Range

When an object has a pair of attributes indicating the beginning and end of some phenomena, such as initial and final date, instead of representing it as two separated attributes, it is advisable to declare a single attribute type such as *Range* (or *Interval*), as shown in [Figure 7.25](#).

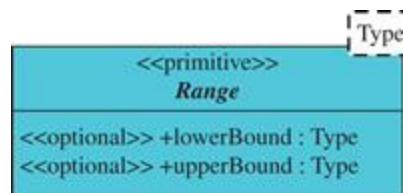


FIGURE 7.25 A generic primitive type *Range*.

The primitive *Range* is parameterized, meaning that when an instance is created it must indicate the *Type* to be used to define its attributes. For example, a range over *Date* could be referred to as *Range[Date]*, and a range over a quantity defined as *Distance* could be referred to as *Range[Distance]*.

There are two reasons for using a range instead of two attributes:

- If the lower and upper bounds are separated attributes in a conceptual class, they would have a strong relation between them. This goes against the high cohesion principle explained in [Section 7.2](#).
- Operations that are specific to ranges (for example, checking if a value is inside a range) will be needed. If there is a specific *Range* primitive type, these operations may be implemented once just in the primitive. If two attributes are used instead, each class where they are declared should implement its own version of the operations on ranges.

It is much more reasonable to implement operations just once in a high-cohesion class, than implementing them many times in low-cohesion classes.

As seen in [Figure 7.25](#), both attributes of the *Range* class are optional, which means that ranges may be undefined in one or both ends. For example, a range that starts at January 1st, 2013 and has no upper bound could be defined with *lowerBound*="01/01/2013" and *upperBound*=null.

Evolutions of this pattern could take into account the mathematical properties of intervals. For example, there are *open* and *closed* intervals: [0..10] is an interval that includes 0 and 10, while (0..10) is an interval that excludes 0 and 10. Furthermore, (0..10] excludes 0 and includes 10 and [0..10) includes 0 and excludes 10. This evolution could be implemented simply by adding two Boolean attributes to the class: *lowerBoundOpen* and *upperBoundOpen*.

7.12 Temporal patterns

Frequently the analyst faces the need to deal with time. For example, it may be necessary to represent the relations between people and their jobs. But, if the association between people and jobs represents only the present time, then the information of former jobs will be lost when someone quits the job. This section discusses some patterns that deal with that important notion.⁴

7.12.1 Effectivity

When an object is valid for some time (for example, a temperature measure, which has a time it was taken and a time it is considered valid), then the *effectivity* pattern may be used. It consists of declaring an *effective* attribute in the class, which is typed with a range, as seen in [Figure 7.26](#).

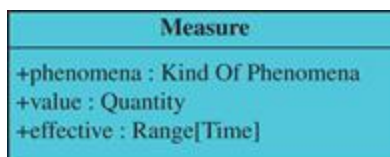


FIGURE 7.26 Use of the *effectivity* pattern.

7.12.2 History

If historical information has to be stored about the past states of an association, then the *history association* pattern may be used. This kind of association may be identified by the stereotype `<<history>>`, as seen in [Figure 7.27](#).



FIGURE 7.27 An example of the history association pattern.

The stereotype of the association of [Figure 7.27](#) means that a person must have just zero or one job currently, but she may have had other jobs in the past. These past jobs may be recovered as items in a list. In other words, it is possible to recover the last job, the job before the last, and so on.

When it comes to design, each class with an association with a navigable role implements a method for accessing the elements linked. In the example of [Figure 7.27](#), the design class for *Person* would have a getter method `getJob()` that returns the company of the current job of a person or the empty set if there is no current job. If the association is stereotyped with `<<history>>`, in addition to that standard method there would exist another method with a parameter, `getJob(index:Natural)`, where `getJob(1)` returns the current job, `getJob(2)` returns the last job, `getJob(3)` returns the job before the last, and so on. If there is no job for the index given, then the method returns the empty set.

In practice, this pattern can be modeled by two associations, as shown in [Figure 7.28](#). The stereotype is, then, a way to abbreviate that complex structure, replacing it with a simpler one.

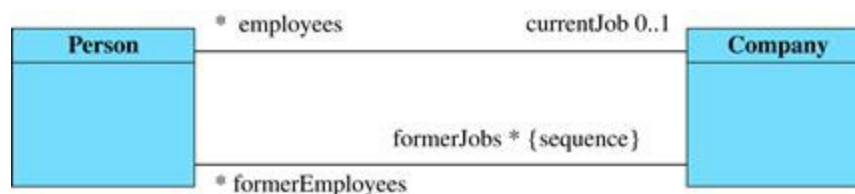


FIGURE 7.28 Concrete model for the `<<history>>` stereotype.

7.12.3 Temporal

The history association pattern is not capable of answering what the job of a person was at a given date, or if a person was unemployed in the past. In order to represent that kind of information, an evolution of that pattern may be used: an association that in addition to sequential memory also has time memory, as shown in [Figure 7.29](#).



FIGURE 7.29 A temporal association.

A temporal association has, in addition to the aforementioned two *get* methods, a method for obtaining the value of the association at a given time. In the example, that method could be *getJob(date:Date)*. Thus, the *Person* class presented in Figure 7.29 would have at least three getters for the temporal *job* role:

- *getJob()*: Returns the current job of a person or the empty set if the person is currently unemployed.
- *getJob(index:Natural)*: Returns the current or former job of a person depending on the index argument, or the empty set if there is no job at that index.
- *getJob(date:Date)*: Returns the job a person had at a given date or the empty set if the person was unemployed at that date.

If it was required to know the job of a person at a given time of the day, then instead of the *Date* the primitive type *Time* could be used, which refers to year, month, day, hour, minute, and second. For some applications, milliseconds, microseconds, and so on could be used instead.

Figure 7.30 presents a model for implementing the *<<temporal>>* stereotype.



FIGURE 7.30 A possible implementation of the *<<temporal>>* stereotype.

The *Job* class does not have a pair of attributes such as *initialDate* and *endDate*, but a single attribute *interval* with type *Range[Date]*, representing an interval between two dates.

Not only can associations be temporal, but attributes as well. If an attribute must keep a record of its former values, then the *<<temporal>>* stereotype can be used to keep track of them.

For example, some countries allow people to change their names after marriage, divorce, or decision of a judge. In Figure 7.31, the ID and birthdate of a person are not supposed to change or, if they change, old values are not kept. However, former names must be kept.

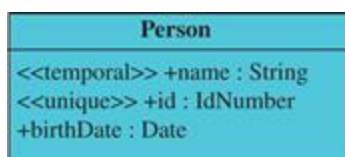


FIGURE 7.31 Temporal pattern applied to an attribute.

Another variation of this pattern is to define a temporal class. In the case of a class stereotyped with `<<temporal>>`, all of its attributes and association roles are temporal. The class also could provide a method for producing a version of its instances at a given instant of time; for example, if x is an instance of a temporal class, one could obtain a previous state of x by using a function such as $x.atInstant(aTimeInstant)$. Each time one of the attribute's values or links is changed, a new version of the object is created.

7.12.4 Bitemporal

Not only in science fiction, but also in many current information system applications, time may be considered two-dimensional. There is a dimension of time in which events occur and another dimension where we acknowledge that an event occurred.

Look back at [Figure 7.29](#), and imagine that Mary is currently working for Company X Today (August 11th, 2013) we discover that in fact Mary changed jobs on March 1st, 2013. We update Mary's job, and the date she changed jobs is recorded as March 1st, 2013. However, we only knew about the change on August 11th, 2013. Any actions the company could have performed between March 1st and August 11th were taken with the knowledge that Mary still worked at Company X. Sometimes, especially for accountability or legal procedures, knowing when a record that was made is very important. The bitemporal pattern is an elegant way to keep track not only of the line of events but also the line of our knowledge about events ([Figure 7.32](#)).



FIGURE 7.32 Bitemporal pattern.

Now a new getter may be introduced for the role *job*: `getJob(date, knowledgeDate: Date)`. The first argument is the instant of time we are interested in and the second argument is the instance of time of our knowledge about it. For example, if we want to answer the question “On June 15th, 2013, where did we believe that Mary worked on June 1st, 2013?” we could use `getJob("06/01/2013", "06/15/2013")` to get the answer, which would be “Company X.”

If we want to know where Mary worked on June 1st, 2013 based on our current knowledge, we could use the getter mentioned for the temporal pattern ([Section 7.12.3](#)): `getJob("06/01/2013")`, which is short for `getJob("06/01/2013", today)`.

[Figure 7.33](#) presents a possible model for implementing that stereotype.

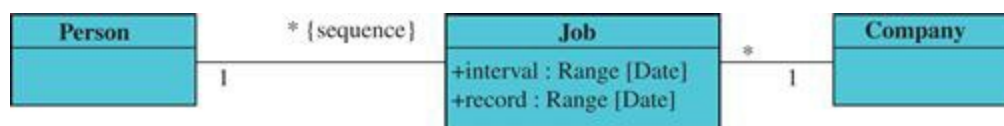


FIGURE 7.33 A model that implements the bitemporal pattern.

Now there are two date ranges in the class *Job*, one for registering the time the job was effective and the other to represent the range of time when the registered time was recorded.

Each time our knowledge about the job of a person is changed a new instance of *Job* is created. The *interval* attribute establishes when the job was effective, and the *record* attribute establishes the point from when we believed that. The upper bound of the record range will be kept undefined while it represents our current knowledge, that is, while that information was not overridden by newer (eventually contradictory) information.

The bitemporal pattern can also be applied to attributes or entire classes, just like the temporal pattern.

7.13 Discussion

A good conceptual model produces an organized structure that is suitable to generate an already normalized database. It incorporates structural rules that prevent information from being represented in an inconsistent way; it also simplifies the code that will be generated because it minimizes and organizes consistency checks, and several verifications that are guaranteed by the model itself do not need to be performed by the code.

The use of suitable design patterns in situations where they are necessary simplifies the conceptual model and gives it more flexibility and quality. It is, therefore, a powerful tool. Many other patterns exist, and analysts may create their own new patterns. It just has to be kept in mind that the creation of a pattern is justified only when its benefits pay for the effort expended in creating it.

7.14 The process so far

	Inception	Elaboration
Business Modeling	<p>Build a general view of the system:</p> <ul style="list-style-type: none"> • Build a business use case diagram and determine the automation scope for the project. • Build preliminary activity diagrams for business use cases. • Build preliminary state machine diagrams for key business objects. 	
Requirements	<p>Prepare the system use case diagram (functional requirements):</p> <ul style="list-style-type: none"> • Identify the system actors from the business use case model. • Identify the system use cases from business use case model, and the activity and state machine diagrams from business modeling. <p>Identify nonfunctional requirements as use case annotations:</p> <ul style="list-style-type: none"> • Identify the main business rules associated to use cases. • Identify the main quality issues associated to use cases. • Identify supplementary requirements. 	<p>Detail requirements by expanding use cases:</p> <ul style="list-style-type: none"> • Identify the main flow. • Identify alternate flows: variants and exception handlers.

(Continued)

	Inception	Elaboration
Analysis and Design	Prepare the preliminary conceptual model by observing system use cases and the concepts needed by them.	<p>Elaborate the system sequence diagrams:</p> <ul style="list-style-type: none"> • Represent the main flow of a use case as a system sequence diagram. • Represent the system commands and queries using stateful or stateless strategies. • Complete system sequence diagrams with alternate flows. <p>Refine the conceptual model:</p> <ul style="list-style-type: none"> • Identify concepts, attributes, and associations in the text of expanded use cases. • Detail attributes and associations with stereotypes, multiplicity, and constraints as needed. • Organize the model by using inheritance, association classes, and temporal specifications. • Add invariants as needed. • Improve the conceptual model with the application of analysis patterns.
Implementation		
Test		
Project Management	<p>Estimate total effort, ideal calendar time, and average team size for the project.</p> <p>Estimate the duration and quantity of iterations for each phase.</p> <p>Prepare the phase plan and iteration plan for the first iteration.</p>	

7.15 Questions

1. In [Figure 7.18](#), the *Transaction* class needs another invariant. A transaction cannot be linked to entries that are linked to the same account. Elaborate that invariant in OCL.
2. Apply the *account/transaction* pattern to the parallel financial transactions of the example shown in [Figures 7.19](#) to [7.23](#). As the bills may be paid forward it may be necessary to use memo entries. Define the set of instances of *Account* and *Transaction* that are necessary and describe them.
3. Use Google to find examples of conceptual models including classes with attributes. Analyze those classes and check if they present cohesion problems. Propose alternative solutions to them if necessary.
4. A system for registering genealogical trees allows different users to check if they have coincident people in their own trees. If this is the case, the shared person is marked in each tree as equivalent to the other. Thus, two cousins, for example, who have the same grandfather, may have a coincident registry in their trees. Which of the three strategies of object joining is best for dealing with this case? Why? Elaborate a model

for representing people in the genealogical trees using that pattern.

5. Complete the diagram of [Figure 7.12](#) with information about orders, items, books, and customers. Then use OCL to define the derived attribute *amount* for both subclasses of *Discount*.

¹Usually, high cohesion is mentioned in conjunction with low coupling. However, as coupling problems appear to emerge when object collaborations are designed, discussion on low coupling is left for [Section 9.6](#).

²*Composite* is the classic name for this pattern. However, if basic elements may be shared by different composite elements, it is the *shared aggregation* association that is used instead of the *composite aggregation*. This happens when strategies are composed.

³The example deals with only five instances for simplification. Many others could be included.

⁴Martin Fowler presents a nice summary on many temporal patterns at <http://martinfowler.com/eaDev/timeNarrative.html>.

CHAPTER 8

Functional Modeling with OCL Contracts

Commands and queries identified in the system sequence diagram must be implemented. Contracts are high-level specifications for those operations. They may be written in natural language or in a formal language such as OCL. This chapter presents a writing discipline for OCL contracts that avoids ambiguous interpretations and that is relatively easy to read and write. Contracts are composed of preconditions, postconditions, exceptions, and return values. Contracts for CRUDs are presented as well as contracts for other operations found in system sequence diagrams.

Keywords

System operation contract; precondition; postcondition; exception; formal specification; OCL; Object Constraint Language

Key Topics in this Chapter

- Preconditions
- Query return
- Postconditions
- Exceptions
- System operation contracts

8.1 Introduction to functional modeling

Before functional modeling, the analysis process in the Elaboration phase produces two important artifacts:

- The *refined conceptual model* ([Chapters 6 and 7](#)), which represents statically the information managed by the system.
- The *expanded use cases* or *system sequence diagrams* ([Chapter 5](#)), which show how potential users exchange information with the system, without mentioning how that information is processed internally by the system and without mentioning the interface technology.

When system sequence diagrams are built, the system commands and queries that must be implemented are identified. Each system command and query implies a user intention or goal; for example, the user may be providing information on which books she intends to buy, or she could be asking how much a book costs. That intention is captured by *system command contracts* and *system query contracts*, generically referred to as *system operation contracts*, which incorporate the functional model of the system.

A *system command contract* may have three sections:

- *Preconditions* (optional): States what is assumed to be true by the command and

therefore should not be checked by it.

- *Postconditions* (mandatory): States how the command changes existing information if it is successfully executed.
- *Exceptions* (optional): States what conditions that could prevent the command from succeeding will be checked by it.

On the other hand, a *system query contract* may have the following sections:

- *Preconditions* (optional): States what is assumed to be true by the query and therefore should not be checked by it.
- *Results* (mandatory): Defines the information that the query returns.
- *Exceptions* (optional): States the conditions that were added by the designer to prevent the system from performing the query in some situations.

Preconditions exist in both types of contracts. They define constraints that are complementary to the ones already existing in the conceptual model. *Preconditions* are assumed to be true by the operation. This means that *preconditions* are *not* tested by the operation that is defined by the contract; they must be tested and assured *before* the operation is performed.

Postconditions exist only in system command contracts because they specify *how the information kept by the system is changed*. Special care must be taken to not confuse *postconditions* with *query results*. By the Command-Query Separation principle (Meyer, 1988) it is not appropriate for a command to return any result (except in special justified cases). Queries, on the other hand, must return some result, but cannot produce any change in the existing information. Therefore, system queries must have *results* and system commands must have *postconditions*.

As opposed to *preconditions* that must be true before the operation is called, *exceptions* are situations that usually cannot or should not be tested before; *exceptions* are tested by the operation itself. *Exceptions* are events that, if they occur, prevent the operation from succeeding.

Exceptions in command contracts and *exceptions* in query contracts may have distinct purposes. In command contracts, *exceptions* are used when arguments or existing information do not satisfy a given business rule (for example, trying to register a customer that is already registered).

However, in query contracts, *exceptions* have a different meaning. As a query does not change information, there is no invalid argument that could possibly prevent a well-defined query from being performed. If a user queries a system about a customer that does not exist, the query is not prevented from being performed: it would answer that there is no customer for the query; this is the right answer, not an *exception*. *Exceptions* in queries would be used only if by any chance the team decides that for a given condition the query *must not* be performed. Usually, those conditions are related to security issues, or other technological concerns such as multi-window interface, concurrency, communication, etc. As these issues are usually not addressed by functional modeling, *exceptions* in queries contracts are not frequent.

Normally, in an operation contract, be it a command or query, an *exception* can be transformed into a *precondition* and vice versa. The decision on designing an issue as a

precondition or exception determines which part of the system is responsible for testing that issue: the *calling operation* or the *called operation*. Usually, in large multiuser systems, *race conditions*¹ may lead to choosing to avoid preconditions; however, if the system is not subject to complex concurrency or a concurrency control mechanism that prevents data from changing during transactions exists, then a design based on preconditions may be the better choice.

For the examples in this chapter a partial refinement of the conceptual model presented in former chapters is used (Figure 8.1), as well as a stateless version of the system sequence diagram for use case 01: *Order books* (Figure 8.2).

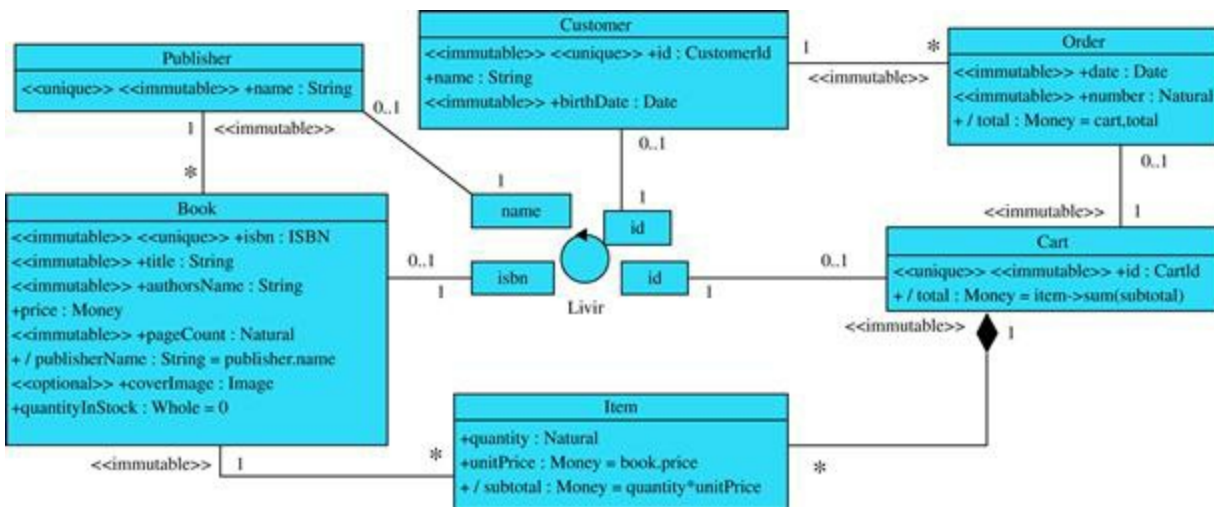


FIGURE 8.1 Reference conceptual model.

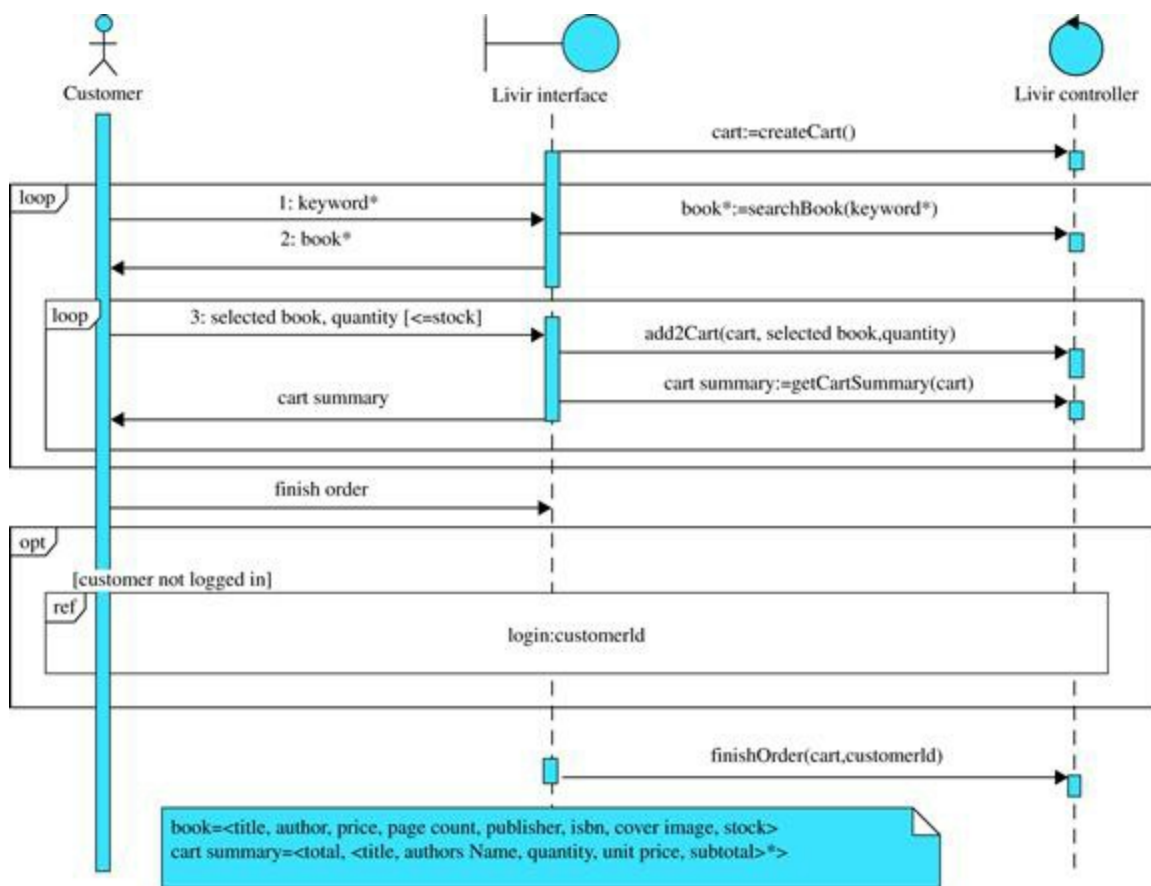


FIGURE 8.2 Reference system sequence diagram.

The `<<immutable>>` stereotype used for some attributes and roles in [Figure 8.1](#) is explained in [Section 8.7.2](#).

In the partial conceptual model of [Figure 8.1](#), the concept of a shopping cart was added because a user may start shopping even if she is not yet identified. Thus, the shopping cart has an identification code (because as the stateless strategy is being adopted it must be identified in some way). Also, despite the fact that the shopping cart is not an independent concept, it should be connected to the controller because it may exist without a customer and even without items (a shopping cart may be empty).

In class *Book*, the attribute *quantityInStock* could probably be a derived attribute if stock was modeled using the *account/transaction* pattern ([Chapter 7](#)). But here it is kept as a simple attribute.²

The first command in the system sequence diagram creates a shopping cart and returns its identifier. This is an acceptable exception for the Command-Query Separator principle, because the interface needs a reference to the shopping cart in order to send further messages to it. If the stateful strategy was used instead of the stateless strategy, this return would not be necessary because the controller would remember which order was just created.

If the customer is not logged in, a fragment named *login* is called (which could be specified by a reusable separated sequence diagram). The fragment returns a customer identifier and replaces the two fragments that were mentioned in [Figure 5.40](#): if the user has an account, she may log in; otherwise she may create a new account and log in.

The operations (commands and queries) found in [Figure 8.2](#) are the following (the

argument passing shown in the diagram was adapted to a style that is more common in programming languages):

- *createCart():CartId* – This is a command that creates a new empty instance of *Cart* and assigns an identifier to it. It returns the identifier.
- *searchBook(keywords:Set[String]):OrderedSet[Tuple]* – This is a query that returns an ordered set of tuples that contain information about the books that are relevant to the search. Each tuple has *title*, *author's name*, *price*, *page count*, *publisher name*, *isbn*, *cover image*, and *quantity in stock* for a book.
- *add2Cart(aCartId:CartId, anIsbn:Isbn, aQuantity:Whole)* – This is a command that creates a new instance of *Item* and links it to the cart and book identified by the parameters.
- *getCartSummary(aCartId:CartId):OrderedSet[Tuple]* – This is a query that returns an ordered set of tuples with information on the cart identified by the cart ID. The information returned, as stated in the note of [Figure 8.2](#), is *title*, *author's name*, *quantity*, *unit price*, and *total*.
- *finishOrder(aCartId:CartId, aCustomerId:CustomerId)* – This is a command that creates a new instance of *Order* and links it to the customer and the cart identified by the parameters.

8.2 Preconditions

Preconditions establish what must be true when a system query or command is performed. For example, considering the reference conceptual model of [Figure 8.1](#), the operation *add2Cart(aCartId:CartId, anIsbn:Isbn, aQuantity:Natural)* may assume that the cart ID is valid because it was obtained by the previous operation *createCart():CartId*. Thus, a precondition that states that this is true could be written:

```
Context Livir:: add2Cart(aCartId:CartId, anIsbn:Isbn, aQuantity:Natural)
```

```
pre:
```

```
cart[aCartId]->notEmpty()
```

The OCL expression above states that in the context of the method *add2Cart*, which is a system command implemented by the controller *Livir*, there is a precondition that establishes that there is indeed an instance of *Cart* that is identified by the argument *aCartId*.

If the association between *Livir* and *Cart* in [Figure 8.1](#) was not qualified, then the precondition could be written by applying *select* to the set of carts:

```
Context Livir:: add2Cart(aCartId:CartId, anIsbn:Isbn, aQuantity:Natural)
```

```
pre:
```

```
cart->select(id=aCartId)->notEmpty()
```

However, when an association is qualified as in [Figure 8.1](#), a key value should be used

to directly access the element mapped by the association. Thus, as the *add2Cart* command has an argument *aCartId*, the expression *cart[aCartId]* produces the same result as the expression *cart->select(id=aCartId)*. Remember that the former expression is also an abbreviation of *self.cart->select(aCart|aCart.id=aCartId)*.

In order to be useful for software development, preconditions must be expressed in a careful way. They must reflect facts that can be identified in the previously developed conceptual model, or the model has to be updated to reflect the new information just discovered. This justifies the use of formal languages such as OCL for writing contracts (Warmer & Keppe, 1998) instead of natural language.

Two families of preconditions may be identified:

- *Parameter guarantee*: Preconditions that assure that the arguments of a command or query correspond to valid objects (for example, “there exists a shopping cart with the ID passed as an argument”).
- *Complementary constraint*: Preconditions that further constrain the conceptual model when performing a given command or query, in order to assure that the information is in a given state when the operation is being executed (for example, “the cart has at least one item”).

8.2.1 Parameter guarantee

There is a systematic way to know if a given operation must have a parameter guarantee precondition (or exception). This technique is also related to the operation’s functional test, as seen in Chapter 13. The team must look at each parameter of the operation and ask if there are valid and invalid values considering the type of the parameter.

For example, if the parameter type is *Isbn*, and the command is *add2Cart(..., anIsbn:Isbn, ...)*, then, considering the business rules, there is a group of valid values (ISBNs that belong to a registered book), and a group of invalid values (ISBNs that are valid but not assigned to any book in the system).

On the other hand, if the type of the parameter is *String* and the command is *add2Cart(..., anIsbn:String, ...)*, then three classes should be considered: one valid (the same as before: ISBNs for books that are registered), and two invalid classes consisting of the ISBNs that are valid but do not belong to any registered books (as before), and incorrectly formed ISBNs (because the type *String* admits *any* string, not only ISBNs). That distinction between incorrectly formed and nonexistent ISBNs can be useful to prevent, for example, misunderstandings related to typos when the user is entering ISBN as a search parameter.

Thus, considering the type of the parameter and identifying the groups of invalid parameters, the rule for finding parameter guarantee preconditions is the following:

For each group of invalid values for a system operation parameter, a precondition or exception must be defined.

Regarding parameter guarantee preconditions, special care must be taken not to

confuse *semantic* preconditions with simple *syntactic* verifications. Semantic verification requires that the database is consulted: for example, verifying if there is a book with a given ISBN. However, syntactic verification may be done by mechanisms other than preconditions. If a parameter has a type *Natural*, no database query is needed to verify if a given argument belongs to the type or not. Then, if a parameter is declared as $x:Natural$ it is not necessary to write any preconditions to assure that x is a positive number: the type already states that.

Another example is the ISBN, which has a formation rule, and can be checked syntactically. Instead of writing preconditions to verify if a parameter is a well-formed ISBN, the team should simply create a primitive type *Iisbn* with a verification method, and use it.

What about *PrimeNumber*? Could it be a type? The answer is *yes*, because in order to know if a given number is a prime number, a syntactic verification is sufficient.

Types must be defined in the command signature. If the type does not exist, a primitive type can be defined by the team. For example, [Figure 8.1](#) presents primitive types such as *CustomerId* and *OrderId* that may incorporate not only identification and checksum mechanisms but also security mechanisms, because those identifiers could be encrypted to avoid unauthorized access. In both cases, primitive types could be defined to deal with the complexity of the verification mechanism.

8.2.2 Complementary constraints

A complementary constraint consists of assuring that certain constraints stronger than those of the conceptual model are obtained while a given command is performed. Thus, if the conceptual model states that a given role has multiplicity 0..1, for example, a complementary constraint could just say that when performing a given command that role is filled (1) or not (0). For example, in general a *Cart* may have an *Order* associated with it or not (0..1). But the command that finishes an *Order* must have a precondition that states that the *Cart* was not yet linked to any order (the role from *Cart* to *Order* must not be filled: 0).

Thus, a precondition can never contradict the conceptual model. If the conceptual model states 0..1, then no precondition could state a multiplicity of 2 or more for the same role. But if the conceptual model states 0..2, then the precondition can further restrict it, stating, for example, 0..1 or 1..2.

Complementary constraints are common in designs that use the stateful strategy, because in that case each command will need information that was left in transient memory by other commands, such as, for example, the identification of the client, which is passed by one command and used by others. Thus, those other commands would have established as a precondition that the client was adequately identified (possibly by a transient association).

It is possible to identify at least three kinds of complementary constraints:

- A *specific statement* about an instance or about a set of instances.
- An *existential statement* about a set of instances.

- A *universal statement* about a set of instances.

One example of a *specific statement* about an instance, considering again the model of [Figure 8.1](#), could be to state that a customer with ID 123 has *abirthDate* equal to July 14th, 1970:

```
Context Livir::someCommand()
```

```
pre:
    customer[123].birthDate=Date.getDate(1970,7,14)
```

As OCL does not define any literals for the *Date* type, a predefined operation *Date.getDate* may be used to define a given date.

An example of an *existential statement* would be to say that there is at least one customer born on July 14th, 1970 (does not matter who):

```
Context Livir::someCommand()
```

```
pre:
    customer->exists(aCustomer |
        aCustomer.birthDate=Date.getDate(1970,7,14))
```

An example of a *universal statement* would be to say that every customer was born on July 14th, 1970:

```
Customer Livir::someCommand()
```

```
pre:
    customer->forAll(aCustomer |
        aCustomer.birthDate=Date.getDate(1970,7,14))
```

Both the expressions *exists* and *forAll* may be written in simplified form as *exists(birthDate=Date.getDate(1970,7,14))* and *forAll(birthDate=Date.getDate(1970,7,14))*, keeping the same meaning stated above.

8.2.3 Precondition assurance

As the preconditions are not tested by the command that declares them, an external mechanism should exist to assure that they will be guaranteed before the command is called.

This guarantee can be made by explicitly calling a query that verifies if the condition is true before calling the command, or by interface mechanisms that prevent the command from being called with invalid data.

For example, instead of allowing the user to type any string in an ISBN field, the use could be led to select from a list of valid ISBNs. That way, the parameter would be validated before performing the command.

Usually the flow of messages expressed in the system sequence diagram is a way to verify if preconditions are viable instead of exceptions. For example, in [Figure 8.2](#) observe that the cart ID received by the operation *add2Cart* is valid because it was returned by the command *createCart*; the ISBN of the book is also valid because it was collected from a list returned by the *searchBook* query. Finally, the quantity may be accepted as valid because the interface can compare it to the quantity available in stock that was returned with other information about the book.³

8.2.4 Preconditions and exceptions versus invariants

Invariants ([Section 6.7](#)) are used in the conceptual model to represent rules that are always valid, independent of any command or query. Preconditions are used for rules that are valid only when a given command or query is being performed.

When an invariant already exists for a given situation, it is equivalent to the existence of an exception for any operation. For example, if there is an invariant that states that a student may enroll only in courses related to her career, then the invariant functions as an exception for any command. Thus, commands for enrolling a student may consider that the exception already exists in the form of the invariant and if the student tries to enroll in a course that does not belong to her career an exception would be raised.

However, an invariant does not work as a precondition. The existence of an invariant does not assure that the design does not violate it. However, a command may add a precondition that states that the invariant must be assured before it is executed.

Test mechanisms must verify if the invariants and preconditions are being respected during the system test activities. If those conditions are violated at any moment, they should raise exceptions. However, in those cases, the designer must revise the design to correct that error so that it will not happen again. When the system is delivered to the final user, it must be guaranteed that no invariant and precondition is violated at any time.

8.3 Transient associations

When the stateful strategy is used in the system sequence diagram, it is necessary for the controller to keep in memory certain information that is not persistent, but that must be stored during the execution of the use case.

Some objects, attributes, or associations that are not persistent may be defined to indicate information that is stored only in main memory during a use case, and discarded after that. Those elements may be stereotyped with `<<transient>>`.

For example, a transient association may be used to indicate that the controller should store information about who is the current customer in the refined conceptual model, as shown in [Figure 8.3](#).

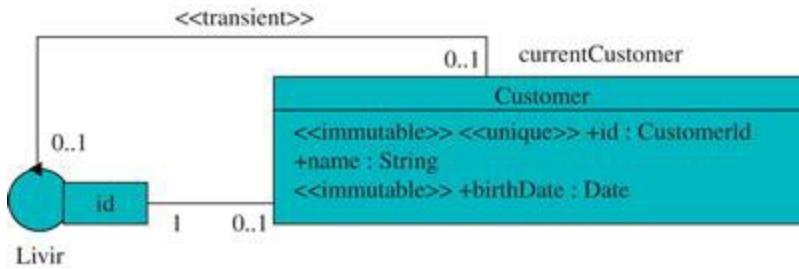


FIGURE 8.3 A transient association.

Thus, a precondition of a command, for example, could state that a current customer already exists:

```

Context Livir::someCommand()
pre:
  currentCustomer->notEmpty()
  
```

Transient attributes or classes, if needed, may be defined with this stereotype as well. In any case it means that the stereotyped information piece is not persistent.

8.4 Query return

As mentioned before, system commands change data while queries just return data to the user. The contracts for system queries must define what is returned, and this can be done in OCL by using the *body* clause.

Expressions that represent preconditions are all Boolean, but expressions that represent the return of a query may have other types. They can return strings, numbers, lists, tuples, or even more complex structures. The following examples are based on the model of [Figure 8.1](#). Initially, a query is defined that returns the *total* of a given cart:

```

Context Livir::getCartTotal(aCartId:CartId):Money
body:
  cart[aCartId].total
  
```

System queries as well as system commands always have the *Controller* as their context. Therefore, in the example above, *cart* is a property of the controller Livir: an association role from it to the class *Cart*.

The following query returns the name and birthdate of a given customer:

```

Context
Livir::getCustomerNameBirthDate(aCustomerId:CustomerId):Tuple
body:
  Tuple {
    name=customer[aCustomerId].name,
  
```



```
    birthDate=customer[aCustomerId].birthDate
}
```

The *Tuple* constructor is one of the ways to represent *DTOs*⁴ in OCL; the tuple works as a Pascal record in the example with two fields: *name* and *birthDate*. The values of the fields are defined by the expressions after the “=”.

To avoid repeating expressions like *customer[aCustomerId]* or even more complex expressions, the *def* clause can be used in order to define a constant that refers to the original expression. Using the *def* clause, the contract above would look like the following:

Context

```
Livir::getCustomerNameBirthDate(aCustomerId:CustomerId):Tuple
```

def:

```
    aCustomer=customer[aCustomerId]
```

body:

```
Tuple {
```

```
    name=aCustomer.name,
```

```
    birthDate=aCustomer.birthDate
```

```
}
```

The expression inside the *def* clause indicates that the identifier *aCustomer* is referring to the customer whose ID is the parameter *aCustomerId* received from the query.

The following expression makes a *projection* in the set of customers returning the names of all customers:

```
Context Livir::listCustomerNames():Set
```

body:

```
    customer.name
```

The next expression applies a filter and a projection, returning the names of all customers that are less than 25 years old:

```
Context Livir::listYoungCustomers():Set
```

body:

```
    customer->select(birthDate.addYear(25)>Date.getCurrent()).name
```

The idea in the example above is to add 25 years to the birthdate of the customer and compare it with the current date: if the customer reaches her 25th birthday after the current date then she is younger than 25 years. OCL also has operations to add days and months to dates: *addDay* and *addMonth*, respectively. The expression *Date.currentDate*

returns the system's current date.

The last example is a query that returns the cart summary, as shown in [Figure 8.2](#).

```
Context Livir::getCartSummary(aCartId:CartId):Tuple

def: aCart=cart[aCartId]

body:

Tuple {

  total=aCart.total,

  items=aCart.item->collect(anItem|

    Tuple {

      title=anItem.book.title,

      authorsName=anItem.book.authorsName,

      quantity=anItem.quantity,

      unitPrice=anItem.unitPrice,

      subtotal=anItem.subtotal

    }

  )

}
```

The *collect* expression is a way to obtain a set whose elements are properties or transformations on properties of another set. The dot notation (“.”) itself is an abbreviated form of *collect*. For example, *customer.name* is equivalent to *customer->collect(aCustomer|aCustomer.name)* or *customer->collect(name)*.

When possible, the dot notation is preferred because it is shorter. But in the *getCartSummary* example, the need to create a tuple instead of accessing a property of the elements of a set prevents the use of the dot, because the *Tuple* operator is prefixed. Thus, the *collect* expression has to be explicitly used in that case.

In the example, two tuple structures were built: the outer one contains the total of the cart and the ordered set of its items; the inner one contains for each item the information that is needed as mentioned in [Figure 8.2](#): *book title, author's name, etc.*

8.5 Postconditions

Postconditions establish what changes in the information managed by the system after some command is performed. Postconditions also must be carefully specified in terms that can be immediately recognized on the conceptual model structures. Thus, although contracts can be written in natural language, they should be carefully written so they can be immediately translated into OCL expressions.

An OCL post condition is written in the context of a (system) command by using the *post* clause:

```
Context Livir::someCommand()  
  
post:  
  
  <OCL expression>
```

If there is more than one postcondition, then they can be combined by the use of the *and* operator:⁵

```
Context Livir::someCommand()  
  
post:  
  
  <OCL expression> and  
  
  <OCL expression> and  
  
  ...  
  
  <OCL expression>
```

In order to classify the types of postconditions that can be used in a contract, we must take into account that the conceptual model has three basic elements, which are concepts, attributes, and associations. Thus, considering that instances of classes can be created and destroyed, association links can be added and removed, and attribute values can be changed, only five types of postconditions are possible:

- Change an attribute value.
- Create a class instance.
- Add an association link.
- Destroy a class instance.
- Remove an association link.

These commands are the most fundamental or basic commands that may be defined for an object-oriented system. They may not be decomposed any further and all other commands are defined as combinations of them.

Thus, a *basic command* is a command that performs one of the five postconditions listed above. The meaning and behavior of a basic command is predefined.

Unfortunately, programming languages still do not provide a standard implementation for such basic commands. Implementing these commands sometimes is left as a difficult manual task for programmers, especially in the case of commands that add and destroy links.

The basic commands, as defined in the following subsections, are effectively basic in the sense that they do the task without performing certain consistency verifications. For example, a command that adds a link would not verify if the upper bounds for both roles of the association were reached. That kind of consistency verification is preferably made for a whole set of commands belonging to the same contract.

The discussion about keeping objects consistent in contracts is continued in [Section 8.5.6](#).

8.5.1 Changing an attribute value

One of the postcondition types consists of indicating that the value of an attribute was changed. That is usually written in OCL as

```
object.attribute=value
```

But, as OCL is a declarative language, this expression must not be understood as an assignment. It has, in fact, three possible ways to become true:

- *object.attribute* may be changed to become equal to *value*. This is usually the default interpretation.
- *value* may be changed to become equal to *object.attribute*.
- Both *value* and *object.attribute* may be changed to a third value and become equal.

Therefore, in terms of code generation, this kind of expression may produce ambiguous interpretations. [Cabot \(2007\)](#) proposes a default semantic for interpreting such expressions so that ambiguity is reduced. However, another approach is possible, which does not require changing the original semantics of OCL. The idea is to use the “^” (caret) predicate instead of the “=” (equals sign) predicate, as explained below.

We can indicate without ambiguity that an attribute was changed just by declaring that a basic message concerning the change was sent to the object. The basic message is predefined, and its name should start with the prefix “set” followed by the name of the attribute. The new value is passed as an argument.

The message *setAttribute* may be sent to an instance of a class that contains the attribute. For example, if we are interested in changing the price of a given book, the following OCL expression could be used:

```
Context Livir::updateBookPrice(anIsbn:Isbn;newPrice:Money)
```

```
post:
```

```
book[anIsbn].price=newPrice
```

But we have already seen that it can be interpreted in ambiguous ways. The expression below expresses the same intention but without any ambiguity:

```
Context Livir::updateBookPrice(anIsbn:Isbn;newPrice:Money)
```

```
post:
```

```
book[anIsbn]^setPrice(newPrice)
```

The caret notation means that the message at the right *was sent* to the object or collection of objects at the left. Thus, it is a Boolean predicate and the result of the expression above is Boolean.

The expression *book[anIsbn].price=newPrice* means that two values are equal no matter

how that result was accomplished. But `book[anIsbn]^setPrice(newPrice)` states unambiguously that the attribute *price* of object `book[anIsbn]` was changed to *newPrice*.

This form of writing is still considered a declarative specification, not imperative. The expression `book[anIsbn]^setPrice(newPrice)` just states that an object has received a message; it does not say who sent that message. The decision about which object is going to send that message to `book[anIsbn]` is left for dynamic modeling ([Chapter 9](#)).

8.5.2 Creating an instance

The command that creates an instance must just do that: create a class instance. Although OCL is not an imperative language, it has a constructor to indicate that a new instance of a class was created. This is expressed as a property of an object, and is referenced as follows:

```
object.oclIsNew()
```

Its meaning, when used in a postcondition, is that the *object* was created during the command being specified by the postcondition. This property may be used in conjunction with another that declares the class of the object:

```
object.isTypeOf(class)
```

However, as it would be a bit tedious to keep declaring the creation of an instance using two expressions, the definition of a single predicate that states that an object was created as an instance of a given class is suggested. The expression `object.isNewInstanceOf(class)` is therefore defined as being equivalent to

```
object.oclIsNew() and object.oclIsTypeOf(class)
```

Thus, a postcondition that declares that a new instance of *Book*, referred to as *newBook*, was created can be defined as

```
newBook.newInstanceOf(Book)
```

In this case, the caret notation is not used because it is not a message sent to an instance of *Book*. It is a declaration that an instance *newBook* of class *Book* was created.

The message `newInstanceOf` does not say anything about the instance's attributes and links, even the mandatory ones. As before, it can be assumed that other expressions will initialize mandatory attributes and links, and that the new instance's consistency will be checked for the contract as a whole and not for each individual basic command.

Attributes with initial values are considered automatically initialized when the instance is created. Thus, it would be redundant to specify postconditions to update the attributes with initial values, unless values different from the default are desired.

Derived attributes, on the other hand, are calculated, and cannot be directly modified. Thus, they are not initialized.

Attributes stereotyped with `<<optional>>` may be kept undefined at creation time; defining a value for them is not mandatory.

But what happens with other (normal and mandatory) attributes at the moment an instance is created? The basic command represented by the predicate *newInstanceOf* simply produces the instance. It does not initialize attributes and links except for those that are defined with default initial values. If mandatory attributes and links are not initialized the instance may be inconsistent; thus, the contract should specify further postconditions that assure that all mandatory attributes and links are initialized when an object is created.

We have seen so far postconditions to create instances and change attributes. This allows us to combine them to create a contract for a CRUD command that creates a new instance of *Book*. The system command *createBook* may have the following as an *initial draft* of a contract:

```
Context Livir::createBook(anIsbn:ISBN;
  aTitle,anAuthorsName:String; aPrice:Money;
  aPageCount:Natural; aCoverImage:Image)
post:
  newBook.newInstanceOf(Book) and
  newBook^setIsbn(anIsbn) and
  newBook^setTitle(aTitle) and
  newBook^setAuthorsName(anAuthorsName) and
  newBook^setPrice(aPrice) and
  newBook^setPageCount(aPageCount) and
  newBook^setCoverImage(aCoverImage)
```

Note that *quantityInStock* is defined with an initial value and does not need to be initialized in this contract.

The *coverImage* attribute is an optional attribute and for that reason it is the only parameter that could accept the null value; alternatively, it could be suppressed from the parameter list of *createBook*.

The *publisherName* attribute is derived, and as it is read only it cannot be defined in the contract (instead, the *derive* clause in the class defines its value for every instance). However, its value depends on a link between the book and its publisher and the first version of this contract does not define how that link is added. This is discussed in the next section.

There is one more concern here: if we consider that all information is accessible by navigating links starting at the controller instance, then it would not be advisable to declare that an instance was created if it is not linked to any other instance with a path to the façade-controller. Unless another core search mechanism allows one to find instances that are not linked (directly or not) to the controller, the newly created instance is

unreachable. Therefore, the creation of an instance should always happen in conjunction with the adding of a link, which will be explained in the next section.

8.5.3 Adding a link

As mentioned earlier in the chapter, another kind of basic command is one that states that a link was added between two objects. Although link additions are bounded by the multiplicity of the roles, this is only checked for the contract as a whole and not for each individual basic command.

Usually, an OCL postcondition to indicate that a link was created would look like the following:

```
object.role->includes(anotherObject)
```

But, again, this expression may be ambiguously interpreted when at the code generation stage, because there are at least four ways for it to become true:

- Include *anotherObject* in *object.role*.
- Assign to *anotherObject* an object that is already in *object.role*.
- Assign a third object to *anotherObject* and include it in *object.role*.
- Assign the empty set to *anotherObject*, because independently of the contents of *object.role* it includes the empty set.

Usually, the first option is the intended meaning. But again we suggest a notation that does not cause ambiguous interpretations and at the same time is more familiar to software developers. Once again, the “^” notation and a basic command are used to indicate that a link was created.

There are many dialects for naming commands that modify association roles. Here, we use the prefix *add* followed by the name of the role. Another common option would be to use the *set* prefix, as in the case of attributes. However, attributes have their values changed or replaced, while links are added; thus, different goals should have different names.

Considering the association between the *Book* and *Publisher* classes, and considering two instances, *aBook* and *aPublisher*, respectively, a link between the instances may be created in two different ways. First, from the point of view of a book:

```
aBook^addPublisher(aPublisher)
```

Second, from the point of view of a publisher:

```
aPublisher^addBook(aBook)
```

Both expressions are symmetric and produce exactly the same result (a link based on the association is created between the instances).

Associations with mandatory roles, such as the one from *Book* to *Publisher*, usually require that their links are created in the same contract that creates the object that must have the role filled. Thus, a link from *Book* to *Publisher* will be added when an instance of *Book* is created, as shown below:

```
Context Livir::createBook(anIsbn:ISBN; aTitle,anAuthorsName:String;
```

```
aPrice:Money; aPageCount:Natural; aPublisherName:String; aCoverImage:Image)
```

```
post:
```

```
newBook.newInstanceOf(Book) and  
newBook^setIsbn(anIsbn) and  
newBook^setTitle(aTitle) and  
newBook^setAuthorsName(anAuthorsName) and  
newBook^setPrice(aPrice) and  
newBook^setPageCount(aPageCount) and  
newBook^setCoverImage(aCoverImage) and  
newBook^addPublisher(publisher[aPublisherName]) and  
self^addBook(newBook)
```

In the contract above, two mandatory links were added to the new book: one from it to its publisher, and another from it to the controller *Livir*. Remember that as *Livir* is the context of the contract, *self* stands for a *Livir* instance.

8.5.4 Destroying an instance

Although destroying objects is rare in object-oriented systems, it may sometimes occur. Usually out-of-print books and deceased customers are not deleted from the records: they are simply marked as inactive or moved to an alternate place. The information must not be lost if trustable records of the company activities in the past are needed.

However, if a publisher or any other class instance was inserted in the system record twice by mistake, the instances may be merged if both have already been changed or, if no activity was registered for the second instance, it may simply be deleted from the records. In this case, an object must be destroyed.

There are two approaches to indicate that an instance was destroyed:

- *Explicit*: It is declared that an object was destroyed by sending an explicit destruction message to it.
- *Implicit*: All associations to the object are removed so that it becomes inaccessible. In programming languages it is possible to implement *garbage collection* to remove from memory objects that are not accessible anymore.

In this book, the explicit approach is chosen because it makes the modeler's actual intention clearer. An object that has to be destroyed, then, must receive a basic message such as the following:

```
object^destroy()
```

The meaning of this expression in a system command postcondition is that the referred object was destroyed during the execution of the command.

It is assumed that all links to the destroyed object are removed as well. If any object formerly linked to it became inconsistent, then the contract must specify what happens to that object (if it is removed as well or if the role is replaced by a link to another object).

8.5.5 Removing a link

The removal of a link between two objects is made by a basic command prefixed by the word *remove*,⁶ followed by the role name and receiving as a parameter the object member of the role that must be removed. For example, for removing the last item of a given cart, the following contract could be used:

```
Context Livir::removeLastItem(aCartId:CartId)

def: aCart=cart[aCartId]

def: lastItem=aCart.item->last()

post:

    aCart^removeItem(lastItem)
```

When the multiplicity of the destination role of the link to be removed is 1 or 0..1, it is optional to inform the parameter, because there is only one possible link to be removed. If instead of removing the link from the point of view of the cart it was done from the point of view of the item, the final result would be the same, but the basic command would not need a parameter:

```
Context Livir::removeLastItem(aCartId:CartId)

def: aCart=cart[aCartId]

def: anItem=aCart.item->last()

post:

    anItem^removeCart()
```

However, both contracts above leave the item inconsistent, because it has a mandatory association to *Cart* that does not exist anymore if the contract succeeds. The contract must consider that as the role from *Item* to *Cart* is mandatory (1), the removal of this link implies the destruction of the *Item* or the linking of it to another *Cart*, which has to be completed in the same contract.

If the choice is to delete the item, then only that basic command should be used in the contract because with it, all links to the item are considered automatically removed as well:

```
Context Livir::removeLastItem(aCartId:CartId)

def: aCart=cart[aCartId]

def: lastItem=aCart.item->last()
```

post:

```
lastItem^destroy()
```

Trying to remove a nonexistent link is a design error and cannot be declared in well-formed contracts, which are discussed in the next section.

8.5.6 Well-formed postconditions

If we assume that the basic commands do not check if the objects and links are left in a consistent state after they are performed, then the designer must do the verification when she is specifying the system command contracts.

The verifications that must be done can be summarized as follows:

- A command that creates an instance must also have postconditions stating that all the instance attributes were initialized, except for (1) derived attributes (which are calculated), (2) attributes with an initial value (which are predefined by an *init* clause, unless a different value is desired), and (3) optional attributes, which can be null (in that case, initialization is optional).
- All mandatory links for a newly created instance must be added.
- Even if it has no links or associations, a newly created instance must be linked to at least one instance that has a link path to the controller or be linked to the controller itself.
- All created links must stay within the lower and upper bounds defined by their association roles.
- All invariants affected by changes in attributes, links, or instances must stay true.

The invariant-checking mechanism does not assure that the designer is writing a contract that keeps all invariants true. It is the responsibility of the designer to be aware of the invariants and avoid raising exceptions. If an invariant happens to raise an exception that is not handled by the system, then there is certainly a design problem.

Another issue involves attributes declared with initial values. If the definition of the initial value of an attribute uses information that may be obtained only by navigating its links, how can it be calculated before the links are created? We must consider that the initialization of attributes with initial values will be one of the last tasks in the programming code that implements a contract, that is, attributes with initial values may be defined only after all links are added and the instance is consistent. For example, the *unitPrice* of an item in [Figure 8.1](#) may only be initialized after a link between the item and the book is added, because the initial value depends on the link. The contract designer does not need to worry about this because contracts are declarative, but the programmer must be concerned with it when producing code.

8.5.7 Combinations of Postconditions

Usually, a system command will have many postconditions that can be joined by the *and* operator, as mentioned before. But it is also possible to use the *or* operator, which

establishes that one of the postconditions was obtained, but not necessarily the other:

```
post:  
  
<postcondition1> or  
  
<postcondition2>
```

The main problem with this operator is that it is *inherently ambiguous* in terms of code generation, and thus, its use should be avoided if deterministic expressions are possible.

Another operator that can be used to connect OCL expressions is the *implies* operator, which has the same meaning of the implication operator of Boolean logic. *Implies* is used when a postcondition is to be obtained only if a certain condition is true. For example, the following contract adds a book to a cart only if it is not there yet:

```
Context Livir::add2Cart(aCartId:CartId, anIsbn:Isbn,aQuantity:Natural)  
  
def:aCart=cart[aCartId]  
  
def:aBook=book[anIsbn]  
  
def:anItem=aCart.item->select(book.isbn=anIsbn)  
  
pre:  
  
aCart->notEmpty() and  
  
aBook->notEmpty()  
  
post:  
  
anItem->isEmpty() implies  
  
newItem.isNewInstanceOf(Item) and  
  
aCart^addItem(newItem) and  
  
newItem^setQuantity(aQuantity) and  
  
newItem^addBook(aBook) and  
  
newItem^setUnitPrice(aBook.price)
```

The *implies* operator can also be replaced by an *if-then-endif* expression. This expression is especially useful if an alternate postcondition must be obtained when the condition is false; in this case the form *if-then-else-endif* should be used instead. In the case of the last example, if the book is already in the order, the quantity ordered may be added to the quantity of the existing item. The contract below shows how to use this operator:

```
Context Livir::add2Cart(aCartId:CartId, anIsbn:Isbn,aQuantity:Natural)  
  
def:aCart=cart[aCartId]  
  
def:aBook=book[anIsbn]  
  
def:anItem=aCart.item->select(book.isbn=anIsbn)
```

```
pre:
  aCart->notEmpty() and
  aBook->notEmpty()
```

```
post:
```

```
if anItem->isEmpty() then
  newItem.isInstanceOf(Item) and
  aCart^addItem(newItem) and
  newItem^setQuantity(aQuantity) and
  newItem^addBook(aBook) and
  newItem^setUnitPrice(aBook.price)
else
  anItem^setQuantity(anItem.quantity@pre + aQuantity)
endif
```

The meaning of the *@pre* operator is explained in the next section.

The advantage of *implies* is that it is shorter. The advantage of *if-then-endif* or *if-then-else-endif* is that it is more familiar to programmers, and includes a way to specify what must be obtained for both cases of a Boolean expression that is shorter than using two *implies* structures.

8.5.8 Former values

Sometimes a postcondition is built upon values that attributes and roles had before the command is performed. Those former values are referred to by the function *@pre* to indicate that they do not correspond to the value of the attribute *after* the command is performed (which could be different). For example, a postcondition that establishes that a quantity in stock for a given book has increased may be expressed as

```
Context Livir::add2Stock(anIsbn:Isbn; aQuantity:Natural)
```

```
def:aBook=book[anIsbn]
```

```
post:
```

```
aBook^setQuantityInStock(
  aBook.quantityInStock@pre + aQuantity
)
```

So, why was *@pre* used above? Because after the command is performed, the quantity in stock would already be incremented, and the argument for the *setQuantityInStock* basic

command has to be defined as the *previous* quantity plus the new quantity.

Remember that OCL is declarative, just as arithmetic is. $x=x+1$ is not a true statement in mathematical notation, because it cannot be true for any value of x . But when you write $x :=x+1$ in a programming language it means the same as $x=x@pre+1$.

The operator *@pre* may also be used over the links of a given role. For example, to indicate the set of items that *were* linked to an order before the system command was performed, the following expression could be used:

```
def:anOrder=...

post:

  anOrder.item@pre ...
```

Another example with a more intense use of this operator is an expression that obtains the total value derived from an order before the system command is performed. Not only may the quantities and item prices have changed, but the list of items itself as well. Thus it is necessary to use the *@pre* operator three times:

```
def:anOrder=...

post:

  anOrder.item@pre->sum(unitPrice@pre*quantity@pre) ...
```

If, on the other hand, the analyst believes that the values of the attributes and links are not changed by the system command, then the *@pre* expression may be omitted, as in the following example:

```
def:anOrder= ...

post:

  anOrder.item->sum(unitPrice*quantity) ...
```

8.5.9 Postconditions covering collections of objects

It is possible with a single OCL expression to state that a whole set of objects was changed. For example, an expression that states that the price of every book was raised by $x\%$ may be written using the *forAll* expression:

```
Context Livir::raiseBookPrice(x:Percent)

post:

  self.book->forAll(aBook|

    aBook^setPrice(aBook.price@pre*(1+x))

  )
```

If there is no need to use the former values in the equation, that is, if every object is

going to be updated with the same price, then it is possible to abbreviate this notation. For example, if the price of all books has to be set to x dollars, then the following expression could be used instead:

```
Context Livir::setBookPrice(x:Money)

post:

  self.book->forall(aBook|

    aBook^setPrice(x)

  )
```

In this case, there is an even more abbreviated form:

```
Context Livir::setBookPrice(x:Money)

post:

  book^setPrice(x)
```

Remember that in our examples, *book* is a role of an association from the controller to the class *Book*, and therefore it is equivalent to *self.book*, that is, it represents the set of all books that are registered in the system. As in the case of the “.” notation, the “^” notation can be used with sets as well.

8.5.10 Postconditions and real-world events

The process of creating contracts is intimately related to the expansion of the use case and the conceptual model. The conceptual model should be used as a reference at every moment because it is the source of information with which assertions can be made about the information.

Contracts should be always written with expressions that can be interpreted in terms of elements of the conceptual model. Thus, assertions like “the books were shipped” would hardly be postconditions because they do not represent information contained in the conceptual model; they describe real-world events. If the team wishes to express this in a contract, they should create a conceptual structure (attribute, role, or concept) to represent it. For example, the fact that the books were shipped could be represented by creating an instance of the *Delivery* class and linking it to the current instance of *Order*.

8.6 Exceptions

Exceptions in contracts are fail conditions that the designer cannot or does not wish to avoid before performing the command itself.

Sometimes, conditions identified as exceptions may be converted into preconditions. If it is possible to transform an exception into a precondition, it is generally advisable to do so, because it is preferable to avoid problems as soon as possible. It is preferable to keep

the user from performing a mistaken action rather than allowing her to do it and reporting the error later.

OCL does not present a specific clause to manage exceptions, but they can be simulated in contracts by conditional postconditions such as

```
post:
    if <condition> then
        <raise exception>
    else
        <all other post conditions, including other exceptions>
    endif
```

As it would not be easy to represent many exceptions using this nested approach, we use a new stereotyped clause, *exception*, which is a shorter version of the expression above:

```
exception:
    <condition> implies <raise exception>
```

Now, for a contract to indicate an exception it is sufficient to identify the condition that produces the exception and name the exception.

A contract just indicates that exceptions may occur, but their handling must be defined only during the design of the interface or application tier, because if an exception is raised by the façade-controller, it should be handled by the tier immediately above: the application or interface tier.

The following example shows a contract with an exception based on [Figures 8.1 and 8.2](#):

```
Context Livir::finishOrder(aCartId:CartId, aCustomerId:CustomerId)
def: aCustomer=customer[aCustomerId]
def: aCart=cart[aCartId]
pre:
    aCustomer->notEmpty() and
    aCart->notEmpty()
post:
    newOrder.isNewInstanceOf(Order) and
    newOrder^addCart(aCart) and
    newOrder^addCustomer(aCustomer) and
```

```
newOrder^setDate(Date.getCurrent()) and
```

```
newOrder^setNumber(OrderNumberGenerator.getNew())
```

exception:

```
aCart.item->isEmpty() implies Exception.throw('Cannot finish an empty order')
```

Here, a basic *throw* message is used to signal the exception to a class named *Exception*, which is not a basic OCL class. When the code for that command is generated, the exception must be signaled to the object that called the system command (possibly an object from the application or interface tier), and we assume none of the postconditions are obtained.

A system command exception should be a condition that cannot be resolved internally by the implementation of the system command, and that requires the execution flow control to be returned to the interface tier so that the user can be asked to try other options.

Internal software exceptions that a system command can handle internally are not raised to the interface; therefore, they are simply not mentioned in contracts: they will be addressed when internal control mechanisms are designed and programmed.

It is also assumed that when an exception is raised the command cannot be executed and none of its postconditions are obtained, because the data must be kept in a consistent state even when an exception occurs.

As mentioned a bit earlier, some exceptions may be converted into preconditions if the designer devises a way to assure the condition before calling the command. Thus, the contract with an exception shown above could be converted into the following:

```
Context Livir::finishOrder(aCartId:CartId, aCustomerId:CustomerId)
```

```
def: aCustomer=customer[aCustomerId]
```

```
def: aCart=cart[aCartId]
```

```
pre:
```

```
aCustomer->notEmpty() and
```

```
aCart->notEmpty() and
```

```
aCart.item->notEmpty()
```

```
post:
```

```
newOrder.isNewInstanceOf(Order) and
```

```
newOrder^addCart(aCart) and
```

```
newOrder^addCustomer(aCustomer) and
```

```
newOrder^setDate(Date.getCurrent()) and
```

```
newOrder^setNumber(OrderNumberGenerator.getNew())
```


In this case, the command assumes that the previous operations in the system sequence diagram have checked and assured that at this point the cart is not empty; therefore, the command will not check it again. This prevents the *finishOrder* command from performing unnecessary verifications when called, and therefore it simplifies its code.

Only conceptual elements (concepts, attributes, and associations) may appear in the OCL expressions of the contracts. These elements are necessarily related to the business rules of the system being analyzed. The exceptions mentioned here must be exceptions related to the business rules, and not exceptions related to hardware or communication problems. Exceptions that may occur in physical storage, communication, or external devices must be addressed by specific mechanisms in the tiers where those elements belong. Usually, users are not even aware of them.

8.7 Pattern contracts for CRUD

The following subsections present models for contracts for typical CRUD commands. There are three system command contracts and one system query contract. The commands and query are performed within the *Book* class, defined following the model in [Figure 8.1](#). Contracts for CRUD operations are similar and may be considered patterns.

8.7.1 Create contract

The contract for a *create* command usually should include the instantiation of a new object, initialization of its mandatory attributes, and addition of its mandatory links. As an example, the following is a complete contract for creating a new book:

```
Context Livir::createBook(anIsbn:ISBN;aTitle,anAuthorsName:String;aPrice:Money;
  aPageCount:Natural;aPublisherName:String;aCoverImage:Image)
pre:
  publisher[aPublisherName]->notEmpty()
post:
  newBook.newInstanceOf(Book) and
  newBook^setIsbn(anIsbn) and
  newBook^setTitle(aTitle) and
  newBook^setAuthorsName(anAuthorsName) and
  newBook^setPrice(aPrice) and
  newBook^setPageCount(aPageCount) and
  newBook^setCoverImage(aCoverImage) and
```

```
newBook^addPublisher(publisher[aPublisherName]) and
```

```
self^addBook(newBook)
```

This contract assumes that the publisher's name was already checked and is valid; otherwise that condition should be an exception instead of a precondition.

As the *isbn* attribute is already stereotyped with `<<unique>>`, it is not necessary to establish an exception related to the issue of creating a book with an existing ISBN because that exception would be raised by the class invariant defined by the `<<unique>>` stereotype. The designer might want to make it explicit in the contract that such an exception exists; however, it is redundant to declare an exception that is already defined as an invariant, and as redundancy is a possible source of problems it should be avoided.

However, if the team wants to assure that the argument *anIsbn* passed to the command is really an ISBN that is not registered yet, they can do it by adding a precondition to the contract. This does not invalidate the exception that could be raised by the invariant, but it assures that this specific operation is not going to raise that exception:

```
Context Livir::createBook(anIsbn:ISBN;aTitle,anAuthorsName:String;aPrice:Money;
```

```
aPageCount:Natural;aPublisherName:String;aCoverImage:Image)
```

```
pre:
```

```
publisher[aPublisherName]->notEmpty() and
```

```
book[anIsbn]->isEmpty()
```

```
post:
```

```
newBook.newInstanceOf(Book) and
```

```
newBook^setIsbn(anIsbn) and
```

```
newBook^setTitle(aTitle) and
```

```
newBook^setAuthorsName(anAuthorsName) and
```

```
newBook^setPrice(aPrice) and
```

```
newBook^setPageCount(aPageCount) and
```

```
newBook^setCoverImage(aCoverImage) and
```

```
newBook^addPublisher(publisher[aPublisherName]) and
```

```
self^addBook(newBook)
```

8.7.2 Update contract

The *update* command usually involves just changing the value of some attributes, although sometimes it may involve adding or removing links. Before defining a contract for updating an object however, the designer must decide which attributes and links are

immutable. The `<<immutable>>` stereotype added to some attributes and association roles in [Figure 8.1](#) declares that those properties cannot change after an object is created. It is only meaningful when one begins to consider if objects can change or not, and the time for writing contracts is just the time to take that feature into consideration. In the case of an immutable attribute, its value can be defined when an instance is created, but it cannot change after that. An example is the ISBN of a book. An immutable role refers to a link that is usually mandatory and added when the object is created. The object at that role cannot be changed. For example, an item is linked to a cart when created and cannot change to a different cart after that.

The update command cannot change immutable attributes and links. In the case of a *Book* in [Figure 8.1](#) only the following attributes may be updated: *price*, *coverImage*, and *quantityInStock*.

The update contract in this case might look like the following:

```
Context Livir::updateBook(anIsbn:ISBN; aPrice:Money; aCoverImage:Image;
aQuantity: Natural)
def: aBook=book[anIsbn]
pre:
  aBook->notEmpty()
post:
  aBook^setPrice(aPrice) and
  aBook^setCoverImage(aCoverImage) and
  aBook^setQuantityInStock(aQuantity)
```

The ISBN is only used to find the book. But as it is immutable, it cannot be updated. If it *could* be updated, then two ISBN values should be passed as parameters: the old ISBN used to locate the book, and the new ISBN that would replace the old value.

Remember that *unique attributes are not primary keys*, as explained in [Chapter 6](#); they are just attributes with values that do not repeat in different instances. They may or may not be immutable depending on the nature of the concept.

8.7.3 Delete contract

The command that *deletes* an object must consider structural rules of the conceptual model before deciding if an object can or cannot be destroyed.

In the case of [Figure 8.1](#), for example, an instance of *Book* cannot be deleted without a decision on what happens to possible instances of *Item* eventually linked to it, because, from the point of view of an *Item*, a link to a *Book* is mandatory.

In order to perform deletion without violating any structural rules, one must choose one of the three approaches below:

- Assure by *precondition* that the object being deleted would not leave other objects inconsistent regarding the lower bound of their associations. With this approach, only books with no associated items may be selected for deletion. This means that only books that were never sold may be deleted.
- Use an *exception* to abort the delete command if it is attempted for an object that will leave other objects inconsistent. If a user tries to delete book that has items linked to it, an exception would be raised.
- Assure by *postcondition* that all objects that would be left inconsistent after deletion are deleted as well. In this case, items linked to a book to be deleted would be deleted as well.

The third approach is used when we want to propagate the delete command to every object that has mandatory associations to the object being deleted. This propagation is recursive: deletion will continue until no more objects with mandatory links to objects that were deleted are found.⁷ This approach may *not* always be used; there are situations in which propagating deletion is against business rules. In the case of books, for example, deleting items linked to a book that is being deleted would create problems with old orders that are already registered; it is not acceptable. However, deleting items when a cart is being deleted is acceptable, because if a cart is deleted it is expected by business rules that all information related to its items are deleted as well.

A possible contract that uses the *precondition* approach would look like the following:

```
Context Livir::deleteBook(anIsbn:ISBN)

pre:

  book[anIsbn]->notEmpty() and

  book[anIsbn].item->isEmpty()

post:

  book[anIsbn]^destroy()
```

As indicated earlier, the *destroy* message removes the instance of *Book* as well as its eventual links, which do not need to be removed one by one. The preconditions assure that the book exists and that the book does not have any items linked to it. If a given book has items, then the command cannot be called.

A possible contract that uses the *exception* approach would be something like the following:

```
Context Livir::deleteBook(anIsbn:ISBN)

pre:

  book[anIsbn]->notEmpty()

  post:

  book[anIsbn]^destroy()
```

exception:

```
book[anIsbn].item->notEmpty() implies
```

```
self^throw('The book cannot be deleted because copies of it have already been
```

A possible contract using the postcondition approach that propagates deletion to all items linked to a cart would look like the following:

```
Context Livir::deleteCart(aCartId:CartId)
```

```
pre:
```

```
cart[aCartId]->notEmpty()
```

```
post:
```

```
cart[aCartId].item^destroy() and
```

```
cart[aCartId]^destroy()
```

The postcondition above states that besides the cart, every item linked to it is deleted as well. As the *Item* class does not have associations with mandatory links coming from other classes, the destruction process may stop there (the books linked to the items, for example, are not destroyed). Otherwise it would be necessary to destroy other objects, and that should be established by the contract.

The postcondition approach must be chosen when it is desirable to propagate deletion to all objects that depend on the object that is being deleted. Is it also useful when the composite aggregation is deleted; by definition, all components must be deleted as well, unless previously removed from the composite.

However, as mentioned earlier, there are situations in which this propagation is not desirable. For example, removing a book from the catalog is not possible if there are already sales registered for that book. In this case, the precondition or exception approaches must be chosen. The first requires that a book that cannot be deleted is prevented from having its ISBN passed as an argument to that command. For example, the list of books available for deletion at the interface could contain only those books that could really be deleted. If this assurance is not possible or desirable, the exception approach should be chosen. In this case, the interface lets the user try to delete, and if it is not possible, an exception is raised.

Usually, information systems do not allow information to be deleted. A book that is removed from a catalog, for example, is not deleted, but marked as *inactive*; if the information about the book is removed, then the information about former sales would be left inconsistent with reality. For implementing this approach, one simple solution is to define a new Boolean *active* attribute for the class whose members may be inactivated instead of deleted. The default value for this attribute is *true*, because the object usually is active when it is created. In the example, for the *Book* class, the solution is shown in [Figure 8.4](#).

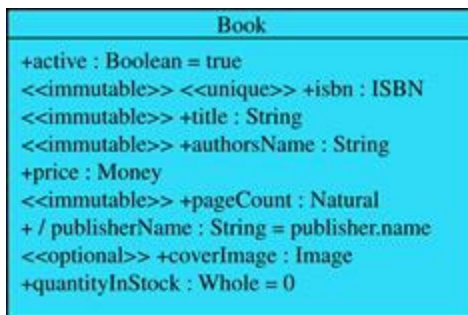


FIGURE 8.4 A variation of the *Book* class whose instances cannot be deleted, but instead are marked as inactive.

In this case, a contract to inactivate a book would be

```

Context Livir::inactivateBook(anIsbn:ISBN)

pre:

  book[anIsbn]->notEmpty()

post:

  book[anIsbn]^setActive(false)
  
```

There is no need to check if the book has items in this case because none will be actually deleted.

Another approach is to move inactive objects to a different space, by for example using a new association to the controller that keeps inactive objects separated from active ones. [Figure 8.5](#) shows an example of this approach for the *Book* class.

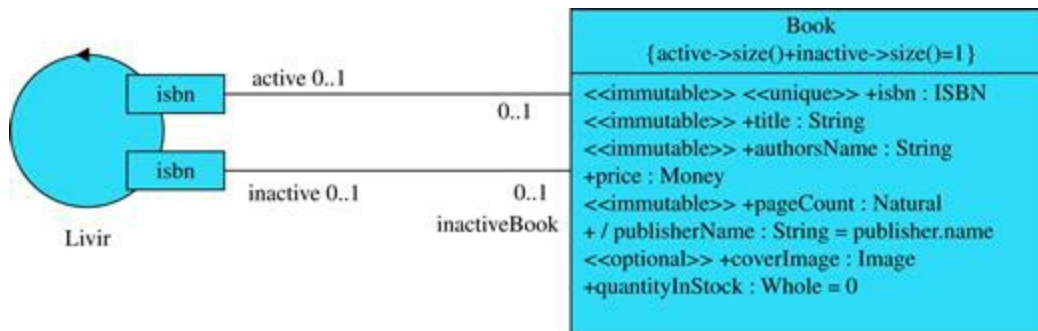


FIGURE 8.5 Alternate way to represent inactive books with two associations.

In [Figure 8.5](#) there are two sets of books: (active) *book* and *inactiveBook*. The constraint in the *Book* class assures that a book belongs only to one or the other set.

Still another approach is to use the *state* design pattern to model the two states of an object, as shown in [Figure 8.6](#). Again, there are two sets of books; the usual set of (active) books may be accessed directly by a derived association while the set of inactive books in the figure must be accessed by consulting the set of books linked to the instances of *Inactive* (however, nothing prevents the creation of another derived association to access inactive books if desired).

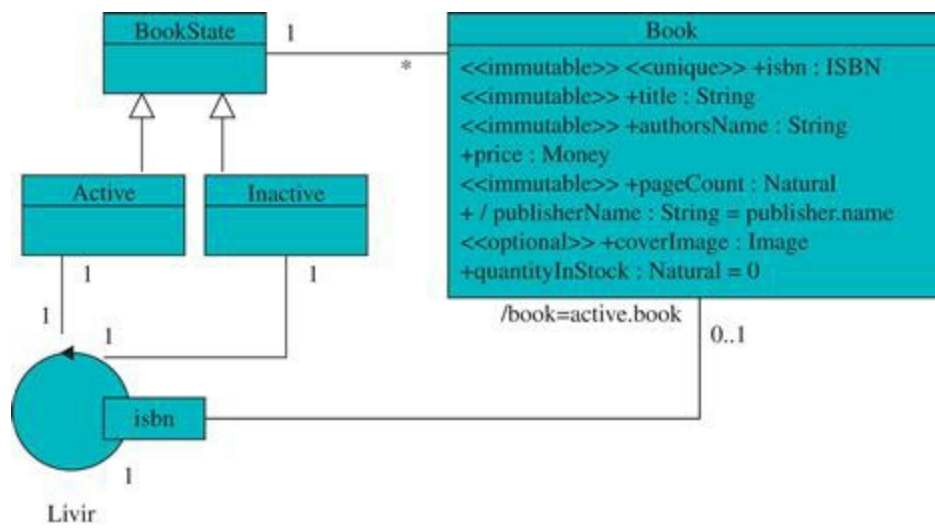


FIGURE 8.6 An alternative way to represent inactive objects using the *state* design pattern.

This model is a bit more complex visually but it avoids the need for a constraint. It also could be transformed into a pattern and a stereotype could be created for it.

8.7.4 Retrieve contract

The simple `CRUD` *retrieve* query returns available data about an instance of a given concept that is identified by one of its unique identifiers. These queries do not return data that is not explicitly in the class definition. Queries that return data calculated from one or more objects are reports (use cases stereotyped with `<<report>>`). Retrieve operations do not perform calculations.

A simple retrieve query for the class *Book* of [Figure 8.1](#) could be defined as follows:

```
Context Livir::retrieveBook(anIsbn:ISBN):Tuple
```

```
body:
```

```
Tuple {
  isbn=anIsbn,
  title=book[anIsbn].title,
  authorsName=book[anIsbn].authorsName,
  price=book[anIsbn].price,
  pageCount=book[anIsbn].pageCount,
  publisherName=book[anIsbn].publisherName,
  coverImage=book[anIsbn].coverImage,
  quantityInStock=book[anIsbn].quantityInStock,
}
```

The retrieve query returns a tuple or record with the data from the attributes of the

object selected by the query parameter.

8.8 Pattern contracts for listing objects

Frequently it is necessary to list objects where one or more attributes are presented. A first example of this kind of contract is listing the *title* of all books available in the bookstore:

```
Context Livir::listBook():Set
```

```
body:  
    book.title
```

As the *book* role from the controller is a strict set in [Figure 8.1](#), the result of the expression *book.title* will be a set as well. Thus, if there are two books with the same title, they will be listed just once, even if the books have a different ISBN.

If a list with multiple columns is desired (for example, author and title), then it is necessary to return a collection of tuples:

```
Context Livir::listBook():Set
```

```
body:  
  
book->collect(aBook|  
    Tuple {  
        authorsName=aBook.authorsName,  
        title=aBook.title  
    }  
)
```

Sometimes, it is necessary to apply a filter to the list, for example returning only the author and title of books that do not have any items linked to them (books that were never sold). In this case, *select* may be applied to the set before forming the tuples:

```
Context Livir::listBookNotSold():Set
```

```
body:  
  
book->select(  
    item->isEmpty()  
)->collect(aBook|  
    Tuple {  
        authorsName=aBook.authorsName,  
        title=aBook.title
```


}

)

Most listing queries likely have only these two constructors: *select* (filter) followed by *collect* (projection). But more complex queries may be created. In those cases, they do not necessarily fit the list pattern (`<<list>>`) and are considered more complex reports (`<<report>>`).

8.9 Contracts related to use cases

An expanded use case is a chain of commands and queries that will be performed in a given sequence. Usually, each operation will supply information or assure preconditions for other operations. The best approach to write contracts for this sequence of operations is to follow the use case flow that is depicted in the system sequence diagram. In this sequence, the designer must ask:

- What is the goal of each operation?
- What do they produce in terms of information?
- What do they expect their predecessors have produced?
- Which exceptions could happen during execution?
- Can their exceptions be reconfigured as preconditions?
- Do their parameters include groups of values that are invalid?
- Do the operations follow a known pattern?

When answering these questions, the designer will be building contracts that allow the commands and queries to be performed in a consistent way in the context of the use case transaction. If it is necessary to add new queries or commands to the sequence diagram in order to assure certain preconditions, then this is the right moment to do that.

This chapter shows many examples related to the system sequence diagram of [Figure 8.2](#). The reader is encouraged to review that figure and ask the questions above for each operation to understand how they have been answered during the chapter.

8.10 The process so far

	Inception	Elaboration
Business Modeling	<p>Build a general view of the system:</p> <ul style="list-style-type: none"> • Build a business use case diagram and determine the automation scope for the project. • Build preliminary activity diagrams for business use cases. • Build preliminary state machine diagrams for key business objects. 	
Requirements	<p>Prepare the system use case diagram (functional requirements):</p> <ul style="list-style-type: none"> • Identify the system actors from the business use case model. • Identify the system use cases from the business use case model, and the activity and state machine diagrams from business modeling. <p>Identify nonfunctional requirements as use case annotations:</p> <ul style="list-style-type: none"> • Identify the main business rules associated to use cases. • Identify the main quality issues associated to use cases. <p>Identify supplementary requirements.</p>	<p>Detail requirements by expanding use cases:</p> <ul style="list-style-type: none"> • Identify the main flow. • Identify alternate flows: variants and exception handlers.
Analysis and Design	<p>Prepare the preliminary conceptual model by observing system use cases and the concepts needed by them.</p>	<p>Elaborate the system sequence diagrams:</p> <ul style="list-style-type: none"> • Represent the main flow of a use case as a system sequence diagram. • Represent the system commands and queries using stateful or stateless strategies.

(Continued)

	Inception	Elaboration
		<ul style="list-style-type: none"> Complete system sequence diagrams with alternate flows. <p>Refine the conceptual model:</p> <ul style="list-style-type: none"> Identify concepts, attributes, and associations in the text of expanded use cases. Detail attributes and associations with stereotypes, multiplicity, and constraints as needed. Organize the model by using inheritance, association classes, and temporal specifications. Add invariants as needed. Improve the conceptual model with the application of analysis patterns. <p>Write system operation contracts for commands and queries in system sequence diagrams:</p> <ul style="list-style-type: none"> Identify preconditions and exceptions based on invalid parameters and complementary constraints. Identify postconditions for commands and returns for queries.
Implementation		
Test		
Project Management	<p>Estimate total effort, ideal calendar time, and average team size for the project.</p> <p>Estimate the duration and quantity of iterations for each phase.</p> <p>Prepare the phase plan and iteration plan for the first iteration.</p>	

8.11 Questions

1. Explain and present an example on how an exception may be transformed into a precondition and vice versa.
2. What is the difference between exceptions in command contracts and in query contracts?
3. What are the five possible postconditions, and what basic commands define each of them? How can they be represented in OCL contracts?
4. Produce the contract for the *searchBook* query in [Figure 8.2](#).

¹For example, after a user orders a book, while the user is closing the order and providing payment, the only available book could be sold to another user. This must be avoided with concurrency control mechanisms, as explained in [Chapter 13](#).

²The type *Whole* stands for the set {0, 1, 2, 3, ...}.

³Remember that race conditions are not yet being considered here and they could invalidate this precondition if concurrency is an issue.

⁴*Data Transfer Objects* are a design pattern that indicates that objects passed by the façade-controller to the interface and vice versa cannot be instances of domain classes (those in the conceptual model). They must be pure data, and not domain objects with behavior. This may be implemented with objects that only have attributes and their respective getters and setters (if any), nothing else. The *record* type in Pascal and the OCL *Tuple* are examples of *DTOs*.

⁵The OCL *and* operator may be used also to combine preconditions, exceptions, and invariants.

⁶Some dialects might use *unset*.

⁷In fact, in the general case, the lower bound of the role is the information that must be considered. For example, if a role has multiplicity 3..*, than removing a link when only 3 are left makes the object inconsistent.

CHAPTER 9

Domain Tier Design

This chapter shows how to design interaction among objects (dynamic modeling) in order to implement system operation contracts using UML interaction diagrams (communication and sequence). Initially, object responsibility and visibility are explained in order to show that objects may only communicate to each other to perform their responsibilities if they have a visibility line to follow. Next, the chapter discusses how to build dynamic models in which objects change messages in order to perform the actions that are necessary to meet the postconditions of a contract, and how the preconditions and exceptions affect it. Design patterns for responsibility distribution among objects are presented in order to improve the design of object-oriented code with delegation and low coupling. Finally, the chapter explains how to use the information learned during dynamic modeling to improve the design class diagram.

Keywords

Object responsibility; object visibility; delegation; low coupling; communication diagram; interaction diagram; design class diagram

Key Topics in this Chapter

- Responsibility
- Visibility
- Low coupling
- Delegation
- Dynamic modeling
- Design class diagram

9.1 Introduction to domain tier design

Software design in general aims to produce a solution to a problem that has already been sufficiently clarified by analysis. The static aspects of the *problem* are represented in the conceptual model and the functional aspects are represented in the expanded use cases, system sequence diagrams, and system operation contracts. Now it is time to design a software (and possibly hardware) *solution* to implement the logical and technological aspects of the system. In this sense *design is a solution for a problem modeled by analysis*.

During iterations, after expanding use cases, refining the conceptual model, and writing contracts, the team may conduct design activities, which can be divided into two groups:

- *Logical design* (this chapter), which includes the aspects of the problem that are related to the business logic. Usually the logical design is represented in the *design class diagram (DCD)*, which evolves from the conceptual model, and by interaction

diagrams, which show how objects exchange messages in order to perform system operations and achieve the postconditions of the contracts.

- *Technological design*, which includes all aspects of the problem that are inherent to the technology used: interface, data storage, security, communication, fault tolerance, etc. Activities related to technological design are addressed later:

- *Interface tier design* ([Chapter 12](#)), which consists of designing the user interface and preferably keeping it decoupled from the domain tier.
- *Persistence tier design* ([Chapter 13](#)), which consists of the definition of a persistency mechanism to allow data load and save, which can be performed automatically, preventing the designer from being (over) concerned about those aspects.

Logical design is also known as *domain tier design*. The domain tier corresponds to the set of classes that perform all data transformation and queries. Other tiers implement technological aspects of the system: persistency, interface, communication, security, etc. They are usually derived from and dependent on the domain tier and their utility is to connect the pure logic of the domain to the physical computer aspects (communication networks, human interfaces, storage devices, etc.).

The design of the domain tier consists basically of two activities that can be performed iteratively:

- *Dynamic modeling*, which consists of building execution models for the system operation contracts. In object-oriented systems, such models are usually represented by interaction diagrams such as communication and sequence diagrams, or even as pure text algorithms.
- *DCD building*, which consists basically of adding to the conceptual model some information that was not possible or desirable to obtain earlier, such as, for example, the association's navigation direction, and the methods to be implemented in each class. Those aspects can be effectively included in the software design if dynamic modeling is done. Additionally, the structure of design classes may be a little different from the conceptual model as new classes or a different class organization may be necessary to perform some functions that will be required at this point.

The logical design can be performed systematically if the team is in possession of two artifacts:

- The evolving design class diagram.
- The functional model represented by system operation contracts.

The logical design activities consist of building interaction diagrams for system operations found in the system sequence diagrams (especially system commands, as explained later), taking into account the DCD and the respective system operation contract.

Dynamic modeling, as mentioned before, can make use of communication or sequence diagrams, or even algorithms. Each of those forms has advantages and disadvantages:

- *Algorithms*, for programmers, are easier to write, but it may be difficult to realize clearly the connections among objects in simple text. Thus, the choice for algorithms to use object-oriented dynamic modeling may raise the risk of high coupling among classes, generating bad design.

- *Communication diagrams* are better than algorithms for visualization and responsibility distribution purposes, and they are better than sequence diagrams for providing an explicit view of the visibility lines between objects. However, these diagrams may be difficult to organize and in the case of complex collaborations they may become harder to read than sequence diagrams.
- *Sequence diagrams* are usually easier to read than communication diagrams, but they do not show explicitly the visibility lines between objects. If the designer is not vigilant, invalid or impossible communications may be included in the diagram.

This chapter uses communication diagrams for dynamic modeling in order to ease the visualization of the visibility lines between objects. In some cases, the equivalent sequence diagrams are shown for comparison.

The class diagram examples in this chapter assume that it is design time. Thus, associations may be directed and classes may have methods and private attributes.

The chapter initially discusses object responsibility, which is a key concept in understanding how to distribute methods among objects. Knowing which object should implement which responsibility allows design classes to be highly cohesive.

Another key concept that is presented in detail is *visibility*, which establishes that communication among objects must occur only when visibility lines exist. If the quantity of those lines is minimized, *low coupling* (Meyer, 1988) is achieved in the design.

Finally, the chapter explains how to transform contracts into dynamic models by using responsibility distribution and visibility, as well as other design patterns, to produce the best design possible.

9.2 Object responsibility distribution

Responsibility distribution among objects has to do with the following question: *What methods must be implemented in which classes?*

Many designers find it difficult to build an elegant solution to that problem when they simply try to add methods to a class diagram. The use of interaction diagrams and design patterns may, however, provide a more efficient and effective way to *discover* the most suitable place to implement each method.

The DCD is built from the conceptual model, which provides the basic set of classes attributes, and associations that are needed. Then, it is necessary to develop interaction diagrams for each system operation contract. Those diagrams show objects exchanging messages in order to achieve the postconditions of the respective contracts. There are techniques that help in building those diagrams in an elegant way. The use of diagrams is better than algorithms, especially for novice designers, because with algorithms, the designer may fall for the temptation of concentrating all responsibilities in a single class, while the diagrams allow better visualization of the distribution of responsibilities among the objects. An example of that situation is presented later in this chapter.

When a team builds object-oriented code without an adequate method for dynamic modeling (that is, simply making a class diagram and adding *ad-hoc* methods to it) there is a risk that the responsibilities may be poorly distributed among the classes, and the

final result may be as unstructured as the old *spaghetti code*. Usually, object-oriented systems are well organized at the class level, but unhappily the code written inside methods is often poorly designed.

Thus, for a system to be elegant, responsibilities have to be well distributed. If a systematic method is not used, responsibilities can become concentrated in the controller class, or in classes that represent business actors, such as *Customer* or *Clerk*. In the end, classes such as *Book*, *Order*, and *Payment* would have no relevant method besides their own attribute getters and setters.

When one or two classes do all the work, and others are passive, there is no object-oriented code worthy of that name, but a concentrating structure. It would be preferable perhaps to make a good *structured design* than a bad object-oriented design.

On the other hand, designers, sometimes, make the mistake of believing that an object-oriented system is a simulation of the real world. But that is not usually true. An object-oriented model represents *the information* about the real world, and not the *things* themselves. The difference is subtle: methods do not correspond to real-world action, but to the internal information processing of the system. That is why concepts such as *Book* that are passive in the real world may perform commands in object-oriented systems.

Thus, designing object-oriented software must be understood as a precise method, guided by learned patterns, and not simply as the act of creating classes and associating *ad hoc* methods to them.

Before explaining how to distribute responsibilities among objects, a taxonomy of those responsibilities will be defined. Basically, there are two high-level groups of responsibilities:

- The responsibility of *knowing*, which corresponds to the queries on objects.
- The responsibility of *updating*, which corresponds to the commands performed by objects.

Both high-level groups are divided into three subgroups:

- Things that the object knows or updates within *itself* (its attributes).
- Things that the object knows or updates in its *neighborhood* (its links).
- Other things that the object knows or updates that are not classified in the previous two groups. Normally these things correspond to *derived knowledge* (derived attributes and associations, and other queries in general) and *coordinated activities* (delegate methods).

In the case of the responsibilities of *knowing*, the three subgroups could be characterized as follows:

- *Things that the object knows about itself*: This is equivalent to being able to access the attributes of an object. Such responsibilities are incorporated to the classes by basic query methods named with the prefix *get* followed by the name of the attribute.¹ For example, if the class *Person* has an attribute *birthDate*, then the method *getBirthDate* has the responsibility of knowing the value of that attribute.
- *Things that the object knows about its neighborhood*: This is equivalent to being able to access directly linked objects. This responsibility is incorporated in the methods that obtain the set of objects linked through a given role. Such methods are usually

prefixed with *get* and followed by the role name (or the name of the class in the absence of the role name). For example, if *Car* is associated to *Person* and the role name for *Person* is *driver*, then the method *getDriver* returns the set of people that are drivers of a given car.

- *Derived things that the object knows*: This is equivalent to information that is composed or calculated from other information. This responsibility may be associated to derived attributes and derived associations, in which case the methods are also prefixed by *get* followed by the name of the derived attribute or association. For example, if *totalValue* is the name of a derived attribute of class *Order*, then *getTotalValue* is a method that returns that derived attribute. Derived knowledge also includes other queries about an object that cannot be represented as derived attributes or associations, such as, for instance, queries with parameters; in that case, names may vary.

In the case of responsibilities related to *updating* information, the three subgroups could be characterized as:

- *Things that the object does over itself*: This corresponds to the basic command that updates a single attribute, which is prefixed with *set*,² followed by the attribute name. For example, if the class *Person* has an attribute *weight* then *setWeight* is a method that incorporates the responsibility of changing that attribute.
- *Things that the object does over its neighborhood*: This corresponds to the basic commands that add and remove links between objects. These commands are identified respectively by the prefixes *add* and *remove*,³ followed by the name of the association role. For example, if the class *Customer* has an association with *Reservation*, the methods *addReservation* and *removeReservation* incorporate that responsibility.
- *Things that the object coordinates*: This corresponds to commands that perform multiple tasks, possibly in different objects. These commands are known as *delegated methods* and they are crucial for class design because the methods of the other groups are just basic commands with previously known behavior; but delegate methods must be designed. For example, if all the books in a given order must be marked as delivered, the coordination of the marking activities must be made by the instance of *Order* itself, because it is the object with the most immediate access to the objects that participate at the operation.

Table 9.1 summarizes the six kinds of responsibilities and the methods typically associated to each kind.

Table 9.1

Object Responsibilities

	Know	Update
About itself	Query on attributes: – <i>getAttribute</i>	Update attributes: – <i>setAttribute</i>
About its neighborhood	Query on association roles: – <i>getRole</i>	Update links: – <i>addRole</i> – <i>removeRole</i> – <i>replaceRole</i>
Others	Queries on derived attributes and roles, and general queries: – <i>getAttribute</i> – <i>getRole</i> – names vary (in the case of general queries)	Delegated methods: – names vary

There is no standard name for delegated methods and general queries, which are named depending on the operation they perform. Usually, these names must not include the name of the class where they are implemented. For example, if *Livir* implements a system command *finishOrder*, and that command delegates that responsibility to the class *Order*, then the name of the method implemented in class *Order* must be simply *finish*, because it would be probably invoked as *anOrder.finish()*.

9.3 Visibility

For two objects to be able to exchange messages to perform their responsibilities, it is necessary that some kind of *visibility* exists between them. There are four basic forms of visibility between objects:

- *By association*: If there is a link between two objects defined by an association between their classes.
- *By parameter*: If an object, when performing a method, receives another object as parameter.
- *Locally declared*: If an object that is performing a method receives another object as a query return.
- *Global*: If an object is visible by any other object.

None of the visibility forms are symmetric. That is, if *x* has visibility to *y* that does not mean that *y* necessarily has visibility to *x*. Visibility by association may be symmetric if the association is bidirectional, but this is not always the case.

9.3.1 Visibility by association

There is visibility by association between two objects only when there is an association between their respective classes. The kind of visibility allowed depends on the role multiplicity and other features, such as the association being qualified, ordered, and so on.

An association role may be considered a collection of instances, especially if its upper bound is greater than 1. But, in the case of multiplicity 1 or 0..1, it is possible to interpret the role as a set or as an individual object. OCL, in these cases, considers the role both as an object and a collection. Regarding multiplicity, then, the following kinds of visibility

may be identified:

- If the multiplicity is 1, then there is straight visibility to one instance and to the set that contains one instance.
- Any other multiplicity allows only visibility to a set of instances.⁴

The DCD is not the only specification that determines the multiplicity of a role. It is complemented by the preconditions of the contracts, which can constrain multiplicity bounds even more, as seen in the following sections.

9.3.1.1 Visibility to a single object

Figure 9.1 shows a *Payment* class that has a unidirectional association to *Order* with multiplicity 1. In this case, any instance of *Payment* has straight visibility to an instance of *Order*.⁵



FIGURE 9.1 Association role with multiplicity 1.

In this way, when an instance of *Payment* is represented in an interaction diagram, as shown in the communication diagram in Figure 9.2, it can send messages directly to the instance of *Order* that is linked to it.



FIGURE 9.2 Communication diagram where a payment has visibility to a single order.

9.3.1.2 Visibility to multiple objects

Multiplicity values different from 1, such as *, 1..*, 0..1, or 5, provide visibility to a collection⁶ of objects, not just to a single instance. From now on, all those cases will be addressed as * multiplicity, or association *to many*.

Figure 9.3 shows class *Order* with a unidirectional association with multiplicity * to class *Item*.



FIGURE 9.3 Association role with multiplicity *.

In this case, an instance of *Order* can send a message to a set of instances of *Item*. It is

possible to send messages to the set structure itself,⁷ as shown in [Figure 9.4](#). In the example, the message verifies if the set is empty.



FIGURE 9.4 Communication diagram with a message being sent to a set structure.

But it is possible to send a message iteratively to each element of the set, as shown in [Figure 9.5](#). In the example, the message requests the subtotal value of each item.

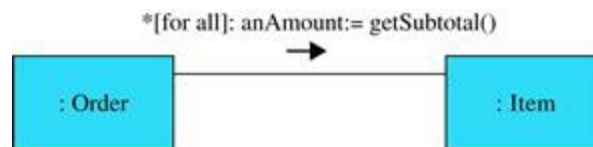


FIGURE 9.5 Communication diagram with a message being sent to each element of the set.

Thus, messages, in communication diagrams, may be addressed to the collection structure or to the collection elements individually. When a message is sent to the structure, the type of the object that receives the message should be a collection, such as *Set<Item>* in [Figure 9.4](#). The fact that the message is iterative and sent to all elements in the set is represented by the “*” that precedes the message. The condition *[for all]* indicates that all elements of the set receive the message.

A filter could be used here if the message was intended just for some selected elements. For example, if the message *getSubtotal* is intended only for items with quantity greater than 1 then it could be written as **[quantity>1]: anAmount:=getSubtotal()*.

9.3.1.3 Visibility by association with ordered roles

If the association role is ordered with the use of the *{ordered}* or *{sequence}* constraints, the visibility to the collection as a whole is kept, but additionally straight visibility to individual elements indexed by their position is allowed.

[Figure 9.6](#) presents a class diagram with a unidirectional association with an ordered role.



FIGURE 9.6 Ordered association role.

In [Figure 9.7](#) the message is sent to each element of the ordered collection, as in the case of [Figure 9.5](#).

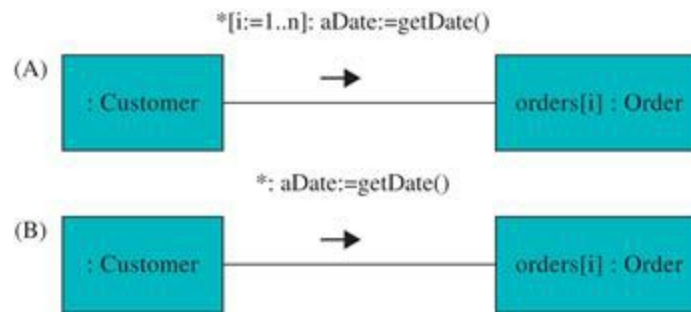


FIGURE 9.7 Communication diagram with a message being sent to an ordered set structure.

Part (A) of Figure 9.7 is the explicit way to represent an iterative message sent to the elements of an ordered set or sequence. Part (B) is a shorter acceptable way to represent the same thing. The form **[for all]*: used for sets would be acceptable as well.

In Figure 9.8 only the third element of the ordered set receives the message, and visibility is not to multiple objects like before, but to a single object. This is indicated in the index used to represent the name of the object: *orders[3]*. The value between brackets may be a constant, as in the example, or a variable known by the *Customer* instance, or even a math expression that results in an integer.



FIGURE 9.8 Communication diagram with a message being sent to a specific element of an ordered set.

In Figure 9.9 only the last element of the ordered collection receives the message.



FIGURE 9.9 Communication diagram with a message being sent to the last element of an ordered set.

9.3.1.4 Visibility by association with qualifiers

If the association is qualified as a map (only one element for each value of the qualifier), there are at least two forms of straight visibility:

- Visibility to the set of elements as a whole (as if it were a regular set).
- Visibility to a single element if the object has a key to access the element in the association role.

Figure 9.10 presents a class diagram with a qualified association from *Customer* to *CreditCard*.

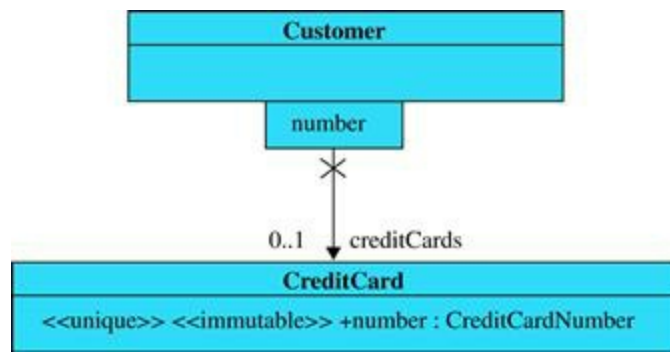


FIGURE 9.10 Class diagram with a qualified association defining a map.

Figure 9.11 shows that in this case, an instance of *Customer* has visibility to the set of instances of *CreditCard* that are linked to it.

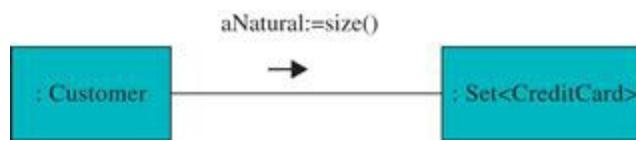


FIGURE 9.11 Communication diagram with a message being sent to the structure of the mapped set.

Figure 9.12 shows a message being sent to each of the credit cards in the mapped set.

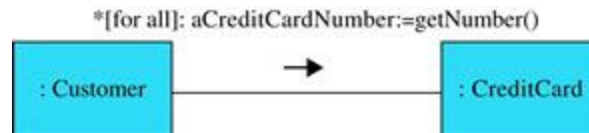


FIGURE 9.12 Communication diagram with a message being sent to each element of the mapped set.

Finally, Figure 9.13 shows that if the instance of *Customer* has a key value for the qualifier (*aNumber* in the example), then it has straight visibility to the element of the mapped set qualified by that key value.

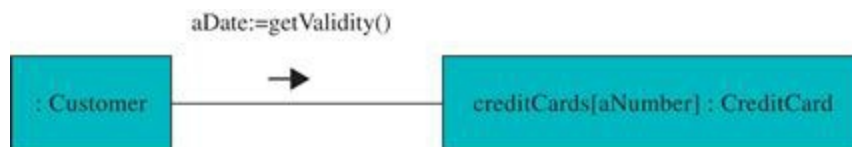


FIGURE 9.13 Communication diagram showing an object with straight visibility to a qualified object in a mapped set.

On the other hand, if the qualified association represents a partition, as in Figure 9.14, a set of values is associated to each qualifier.

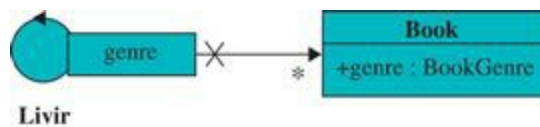


FIGURE 9.14 Class diagram with a qualified association defining a partition.

In this case, the whole set may be accessed (all the books in the example of [Figure 9.15](#)).

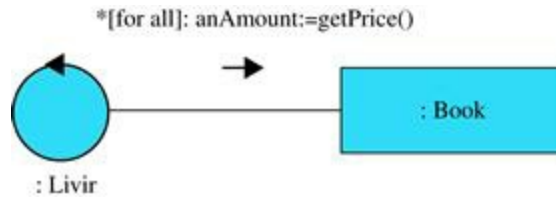


FIGURE 9.15 Communication diagram with a message being sent to each element of the partitioned set.

But the use of the qualifier allows for access to books belonging to a given subset (thriller books, in the example of [Figure 9.16](#)).

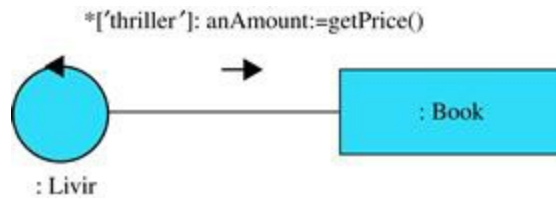


FIGURE 9.16 Iterative message to each element of a partition of a qualified set.

Both examples in [Figure 9.15](#) and [Figure 9.16](#) show messages that are sent to the set elements and not to the set structure.

9.3.1.5 Association visibility with association class

As explained in [Section 6.6.2](#), an association class instance is created and deleted when links are added and removed from the participating classes' instances. Thus, if *Person* has a unidirectional association to *Company* and that association has an association class *Job*, as shown in [Figure 9.17](#), when a link between instances of *Person* and *Company* is added, an instance of *Job* is created; and every time a link from *Person* to *Company* is removed, the respective instance of *Job* is deleted. Therefore, an instance of *Person* has visibility to two sets⁸ with the same size: one with instances of *Company*, and another with instances of *Job*. An instance of *Job* has visibility only to a single instance of *Company*.

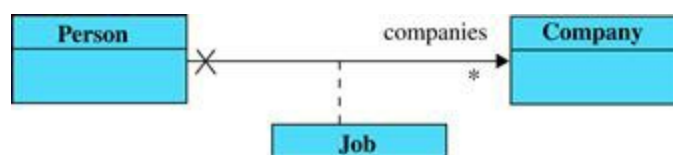




FIGURE 9.17 Association class.

If the association in [Figure 9.17](#) were bidirectional, then an instance of *Company* would also have visibility to a set of *Person* and a set of *Job*. Additionally, any instance of *Job* would have visibility to one instance of *Person* and one instance of *Company*.

[Figure 9.18](#) represents the visibility that an instance of *Person* has to the set of instances of *Company*. That corresponds to the default visibility obtained for multiplicity *.

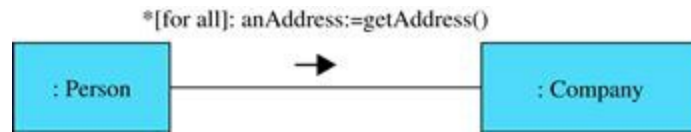


FIGURE 9.18 A message being sent to the elements of the set linked by an association with an association class.

[Figure 9.19](#) represents the visibility that an instance of *Person* has to a set of instances of *Job*, the association class.

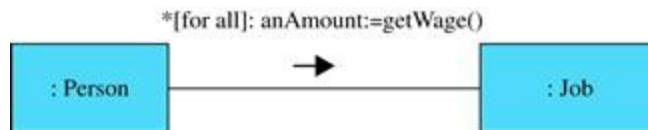


FIGURE 9.19 A message being sent to the elements of the set of instances of the association class.

The association class also works as a mapping in a way that is very similar to the qualified association. The model in [Figure 9.17](#) is equivalent to mapping a job for each company a person works for. Thus, if an instance of *Person* has visibility to a specific instance of *Company*, it may find its job at the company, as illustrated in [Figure 9.20](#).

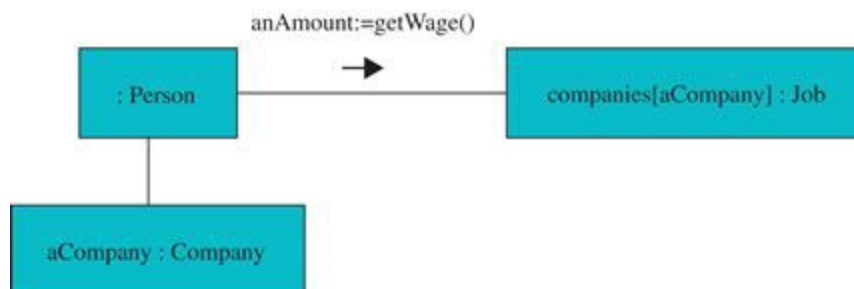


FIGURE 9.20 A message being sent to a specific element of the association class.

Also in [Figure 9.20](#), we can see that *aCompany* must be an instance of *Company* and that the instance of *Person* must have some kind of visibility to it in order to use it to access the respective instance of *Job*.

Finally, [Figure 9.21](#) shows the visibility that instances of the association class have to the instances of the participating classes: a visibility that is strictly one.

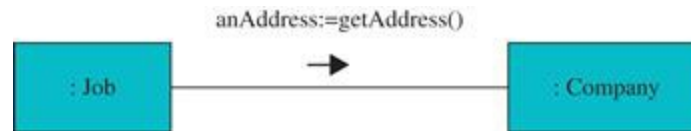


FIGURE 9.21 Visibility from the association class instance to one of the participating classes' instances.

9.3.1.6 The influence of preconditions on visibility by association

When a precondition in a contract states a multiplicity that is more restrictive than the one allowed in the class diagram, it is the visibility established by the precondition that must be considered valid in the context of that operation.

For example, consider the class diagram of [Figure 9.22](#), which defines that an order may or may not have a payment linked to it.



FIGURE 9.22 Association with optional role.

Consider now that a given operation contract has a precondition that states

pre:

```
anOrder.payment->notEmpty()
```

Then, in the context of that operation, and for the specific instance *anOrder*, a more restrictive visibility must be considered, as shown in [Figure 9.23](#).



FIGURE 9.23 How the original visibility of [Figure 9.22](#) must be considered for a specific instance if a contract has a more restrictive precondition.

9.3.2 Visibility by parameter

Visibility by parameter is obtained when an object *A*, performing a method, receives an object *B* as a parameter. In that case, the object *A* can communicate to *B* even if their classes are not associated.

Consider the example of [Figure 9.24](#), where the *Book* class has no association to the *Delivery* class. Therefore, no instance of *Book* may send a message directly to an instance of *Delivery* or vice versa.

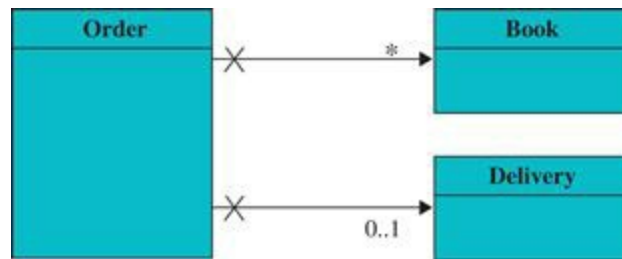


FIGURE 9.24 Reference class diagram.

Consider an instance of *Order* that is linked to an instance of *Book* and to an instance of *Delivery* as shown in [Figure 9.25](#).

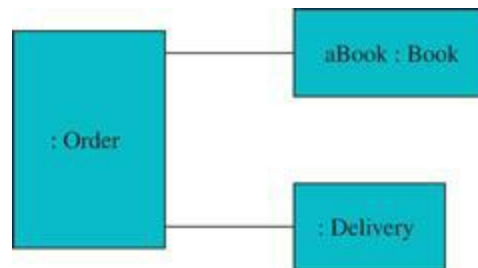


FIGURE 9.25 Initial situation where the instance of *Delivery* has no direct visibility to the instance of *Book*.

Now, consider that the instance of *Order* sends a message *registerWeight* to the instance of *Delivery*, passing a reference to the instance of *Book* as an argument, as shown in [Figure 9.26](#). Then, the instance of *Delivery* acquires visibility by parameter to the instance of *Book*.

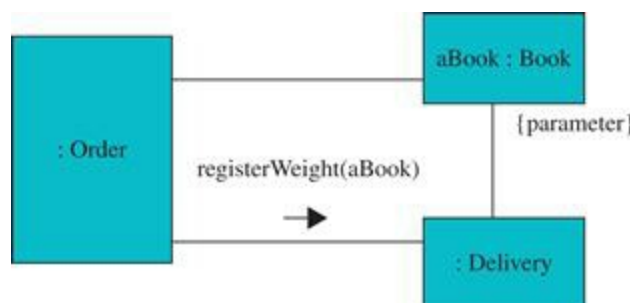


FIGURE 9.26 The instance of *Delivery* acquires visibility by parameter to *aBook*.

In the case of visibility by parameter, it should be assumed that the object that sends the argument originally has some kind of visibility to the object that was sent.

Good design practice demands care with visibility by parameter. This is not the same kind of coupling that is obtained with visibility by association. With visibility by

association, the participating classes are already coupled by a semantic association, that is, they are linked anyway because the meaning of one is attached to the meaning of the other (for example, an order and its items). But in the case of visibility by parameter, any object may be passed as a parameter to another object, even if they have absolutely no relation between them.

Each time a class declares methods with parameters belonging to classes it is not associated to, a new (and possibly unnecessary) coupling may be created. High coupling means poor design. Thus, it is convenient to avoid that whenever possible.

The best use for visibility by parameter is to receive and resend objects until they reach the terminal (basic) operations. In practice, what happens is that certain objects will be passed to other objects in a chain until they reach some object that must be linked or unlinked to them, as in the example below.

Initially, consider the class diagram of [Figure 9.27](#), with classes *Livir* (the façade-controller), *Book*, *Customer*, and *Reservation*.

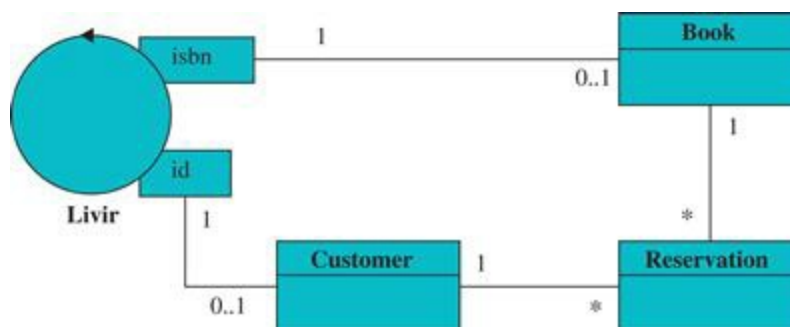


FIGURE 9.27 Reference class diagram.

[Figure 9.28](#) presents a collaboration where instances of these classes cooperate to perform the system command *reserveBook*, which adds a reservation for a given book to a given customer. The system command, implemented in the controller, is called by the interface tier, which passes as arguments the ISBN of the book and the ID of the customer. The controller, initially, gets visibility to the specific instance of *Book* (message labeled “1”) by sending to itself a query message on the role *book*: the message *getBook(anIsbn)*. Then it gets visibility to the instance of *Customer* (message labeled “2”) in a similar way. The instances returned by the messages are assigned to the local variables *aBook* and *aCustomer*. Finally, the controller delegates to the instance of *Customer* the reservation of the book (message labeled “3”).

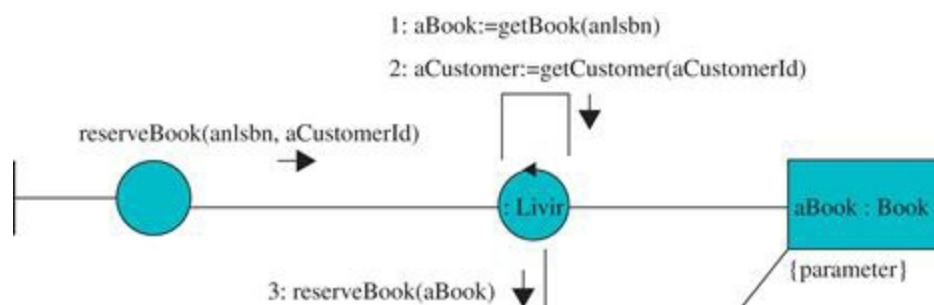




FIGURE 9.28 Creation of visibility by parameter.

The message labeled “3,” *reserveBook(aBook)*, is sent from the controller to the customer whose ID is *aCustomerId*. The controller should not be responsible for creating the reservation because it has no association to *Reservation*, and thus it delegates the responsibility to *aCustomer*. At this point, *aCustomer* acquires *visibility by parameter* to *aBook*.

At this point, *aCustomer* can start performing its responsibilities. It creates a new instance of *Reservation* and links it to itself and to *aBook*. Figure 9.29 shows the constructor message *Create* being sent in order to create a reservation with its initialization parameters: *aBook* and *aCustomer*. This message is labeled with 3.1 because it is called inside the implementation of message 3, which was received by *aCustomer*.

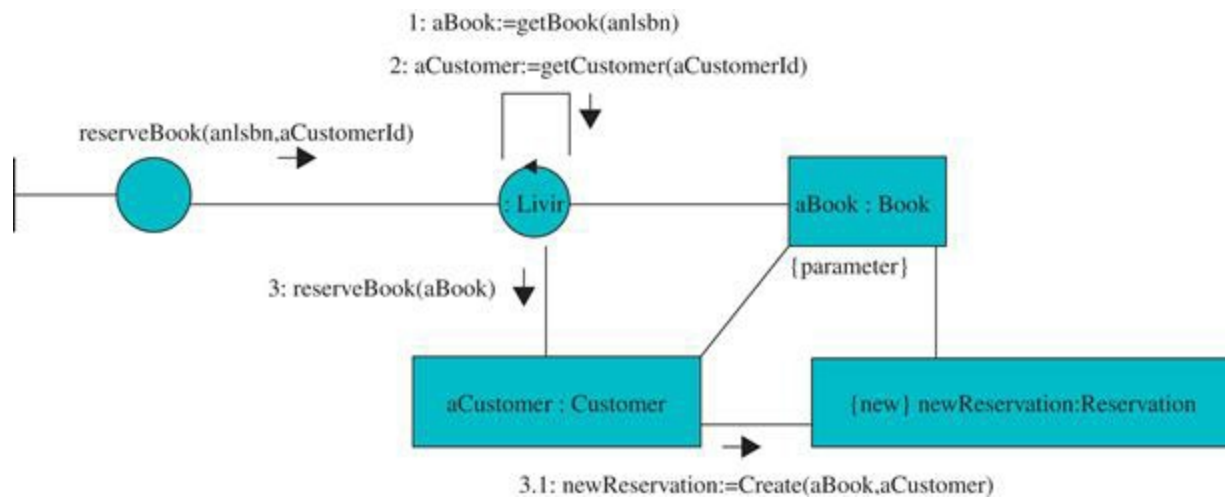


FIGURE 9.29 A communication diagram where a new instance of *Reservation* is created.

As a last observation on the example, try to figure out what would happen if instead of passing the instance of *Book* to the customer, the controller had passed its ISBN code as received from the interface. When the instance of *Reservation* is going to add the book to its links it would have to obtain the instance, because it cannot just add the ISBN code. How could the instance of *Reservation* seek an instance of *Book* by its ISBN if it has no access to the set of all books? It would have to send a message to the controller, asking for the book. This is not good quality design because the flow of messages in this case goes back to the controller for no reason. In the example shown in Figure 9.29, the controller obtains the book as soon as it gets the original message. There is no motive to delay it in order to reverse the flow of control later.

9.3.3 Locally declared visibility

Another form of visibility happens when an object sends a query to another and receives as return a third object, as shown in [Figure 9.31](#). The collaboration is based on the class diagram of [Figure 9.30](#).



FIGURE 9.30 Reference class diagram.

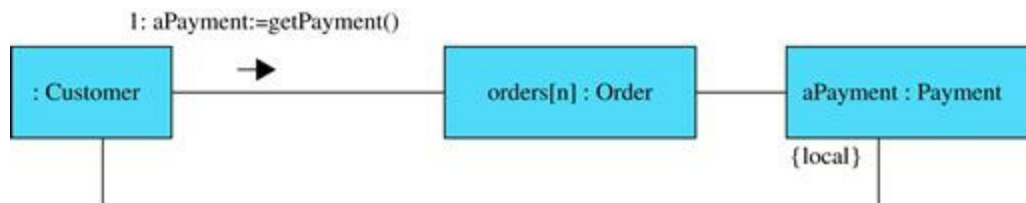


FIGURE 9.31 An example of the creation of local visibility.

In the example, an instance of *Customer* sends a message *getPayment* to an instance of *Order*. The customer and the order are initially linked, but the customer and the payment are not. However, after the *getPayment* method is performed, and an instance of *Payment* is returned to the customer, it acquires local visibility to the payment, because, at that point, the payment is assigned to a local variable (*aPayment*) of the method the customer is executing.

From this moment on, the instance of *Customer* acquires local visibility to the payment and may communicate directly with it.

However, a design pattern known as *Don't talk to strangers* ([Larman, 2004](#))⁹ does not recommend that an object send messages to objects that are only locally visible. According to that pattern the customer should not send messages to *aPayment*. As seen in [Section 9.6](#), there is always a design option that avoids this kind of communication. The problem here again is high coupling: if the customer acquires local visibility to an instance of *Payment*, it acquires a link that is not semantically supported by the conceptual model. On the other hand, the price to pay for low coupling in this case is sometimes having to implement more methods, as seen later.

An acceptable use for local visibility occurs when a basic command that creates an object is invoked. When an object is created, the new instance is usually referenced by a local variable. That local visibility may be immediately transformed into visibility by association, as shown in [Figure 9.32](#).

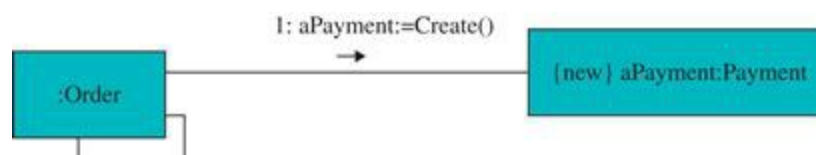




FIGURE 9.32 An example of *new local* visibility being transformed into visibility by association.

In the example, the first message is a basic command, a constructor that produces the instance of *Payment*. At that moment the instance of *Order* gets a new local visibility to it, through the local variable *aPayment*. The second message links that new instance to the order, which acquires visibility by association to it.

Both local and parameter visibilities are valid only in the scope of the method that originated them; just like parameters and local variables in programming languages, they are only valid inside the methods where they have been declared, and disappear after the methods go out of scope. But visibility by association is permanent, persisting until it is explicitly removed by a basic command that removes a link.

9.3.4 Global visibility

There is *global visibility* for an object when it is accessible by any object at any time. The *singleton* design pattern (Gamma, Helm, Johnson, & Vlissides, 1995) suggests that an instance may be globally visible if it is the *only instance of a class*. This makes sense, because if that class has only one instance, it is not necessary to create associations from other objects to it. Also, if that class has a single instance, it is not necessary to pass it as an argument to a function because the idea with parameters is that they may vary, and that does not happen if the class has a single instance. Global visibility is the choice in this case.

Examples of singletons are classes that represent services, such as *CurrencyConverter*, *TimeCounter*, *OrderNumberGenerator*, etc. There must be a single instance of service classes that can be accessed by any object at any time. An example is shown in Figure 9.33.

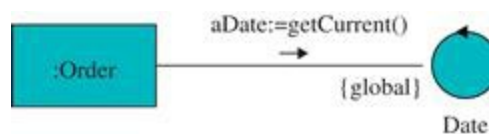


FIGURE 9.33 Global visibility.

If the singleton pattern is overused it may turn into an *anti-pattern*. Designers must not use singletons for entities of the conceptual model; for example, if the system is single user and a single shopping cart may exist at a time, it should not be declared a singleton because it is a conceptual entity, not a service class. A conceptual entity that is unique today may admit multiple instances in the future and declaring it a singleton would hinder the evolution of that class.

Also, a singleton, in principle, should not have direct references to entities, because

that reduces its reusability potential and creates high coupling problems. Service classes must not be associated to entity classes; they are globally visible but grant access to no other instances.

9.4 Dynamic modeling based on postconditions

We saw that system command contracts present a set of postconditions that correspond to certain basic commands for creating and deleting instances, adding and removing links, and modifying attribute values. These contracts only indicate *what* is supposed to happen, but they do not show *how* messages are exchanged between objects in order to achieve those goals. UML interaction diagrams may be used to exactly design how these collaborations may happen. The following principles are recommended:

- The kind of visibility among objects depends fundamentally on the role features, such as multiplicity, ordering, and qualification, that are defined in the class diagram and sometimes further constrained by preconditions (Section 9.3).
- Each postcondition in the contract must be achieved in the interaction diagram by a message that calls a basic command in the object that holds the responsibility (Section 9.2).
- The control flow in a dynamic model begins at the instance of the façade controller that receives a message from the interface.
- When an object that is executing has no visibility to the object that must perform a basic operation, it should *delegate* the responsibility and the execution to another object that is closer to the one that holds the responsibility.

Santos (2007) presents a systematization of those principles by defining an automatic search-based production system capable of generating well-designed communication diagrams from a broad selection of contracts.

In addition to these principles, design patterns should be applied whenever possible to help the designer build methods that are effectively reusable and maintainable.

9.4.1 Instance creation

When a contract establishes that some instance was created, some other object must actually create it by sending a basic *create* message in the respective communication diagram. The *creator* design pattern (Larman, 2004) states that the choice should be preferably:

- An instance of a class that has a composite or shared aggregation to the class of the object to be created.
- An instance of a class that has a *one to many* association to the class of the object to be created.
- An object that has the initialization data for the object to be created, and that is preferably associated to it.

Each rule applies only if the previous one does not apply.

To serve as a reference for the following examples, let's go back to the operations of

the use case 01: *Order books*, presented in Chapter 8. The reference conceptual model of Figure 8.1 is repeated here as Figure 9.34.

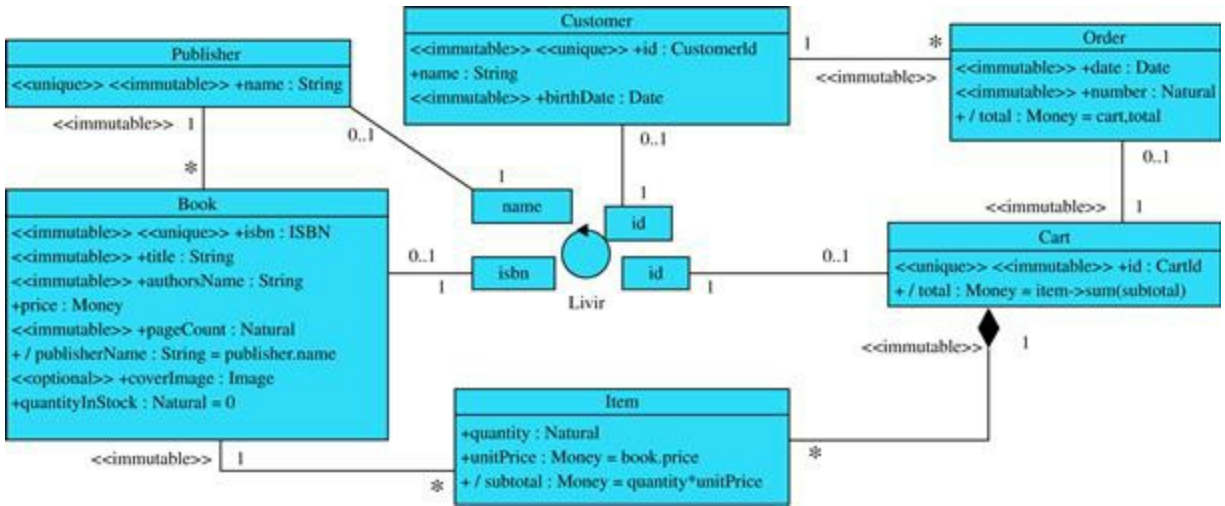


FIGURE 9.34 Reference class diagram.

The first system command of the system sequence diagram of use case 01 is *add2Cart(aCartId:CartId, anIsbn:ISBN, aQuantity:Natural)*. Its contract establishes that if the book is not in the cart already, a new item is created and linked to the book and cart. The item *quantity* is set to the value received as an argument, and *unitPrice* is set to the price of the book. Otherwise, if the book is already in the order, the new quantity is added to the current one. The contract again is the following:

```
Context Livir::add2Cart(aCartId:CartId; anIsbn:ISBN; aQuantity:Natural)
```

```
def:aCart=cart[aCartId]
def:aBook=book[anIsbn]
def:anItem=aCart.item->select(book.isbn=anIsbn)
pre:
  aCart->notEmpty() and
  aBook->notEmpty()
post:
  if anItem->isEmpty() then
    newItem.isNewInstanceOf(Item) and
    aCart^addItem(newItem) and
    newItem^setQuantity(aQuantity) and
    newItem^addBook(aBook) and
    newItem^setUnitPrice(aBook.price)
  else
    anItem^setQuantity(anItem.quantity@pre + aQuantity)
  endif
```

Among other postconditions, an instance of *Item* must be created if the book is not yet

in the cart. The question at this moment is: If the control flow starts at the façade controller (*Livir*), which class has to create an instance of *Item*? The options are:

- The façade controller itself, which already holds the control flow (it would be the choice of novice designers).
- An instance of the *Book* class, because it has a one-to-many association to *Item*.
- An instance of the *Cart* class, because it is a composite aggregation of *Item*.

If the controller option is chosen, then a coupling that did not exist before between it and the class *Item* would be created. That would be a bad choice for design.

Following the creator design pattern, the best choice would be *Cart*. The association between *Item* and *Book* is normal while the association between *Cart* and *Item* is a composite aggregation.

For the first version of the communication diagram, let us forget for a while the conditional *if*, and the postconditions that add links and change attributes values. Let us concentrate on creating an instance of *Item*.

The first task for the designer is to obtain the correct instance of *Cart* given its ID, and delegate to it the creation of the new instance of *Item*. This is shown in [Figure 9.35](#).

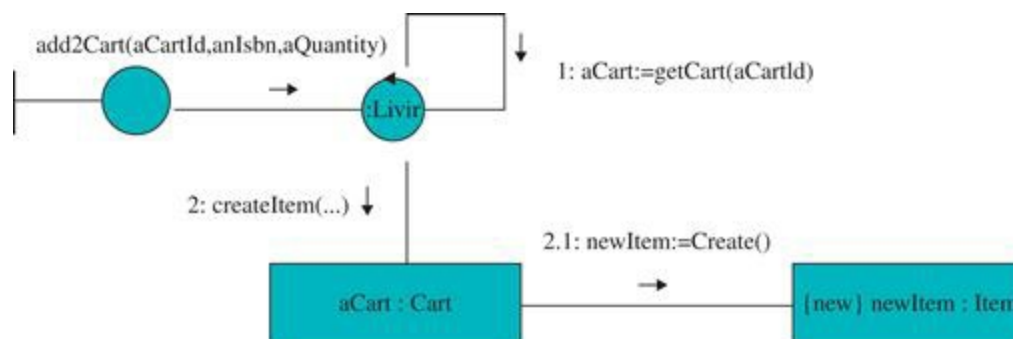


FIGURE 9.35 Communication diagram that shows an instance being created.

This diagram implements only one postcondition: creation of an instance of *Item*. Initially, the controller must determine which cart is given *aCartId*. This is performed by the message labeled “1,” *getCart*, which queries the *cart* role of the controller *Livir* and stores the resulting instance in the variable *aCart*, which is a local variable for the system command *add2Cart* that is being modeled. As the precondition *aCart->notEmpty()* assures that *aCartId* is valid, the dynamic model does not need to check it again.¹⁰

In the sequence, the message labeled “2,” *createItem*, delegates to the cart the responsibility of creating an item, because the controller does not have any association to *Item*. If the controller has created the new item without delegating, then *Livir* would need to be coupled to class *Item* and issues related to high coupling would occur. The parameters of *createItem* were left to be added later as they are not yet necessary.

Finally, the instance of *Cart* achieves the postcondition by sending a basic message *Create*, labeled “2.1” in the diagram, which produces a new instance of *Item*. This instance is stored in a variable named *newItem* that is local to the *createItem* method in the *Cart* class.

Some designers would prefer to visualize the collaboration among objects shown in

Figure 9.35 as a sequence diagram. Figure 9.36 presents a sequence diagram that is equivalent to the communication diagram of Figure 9.35.

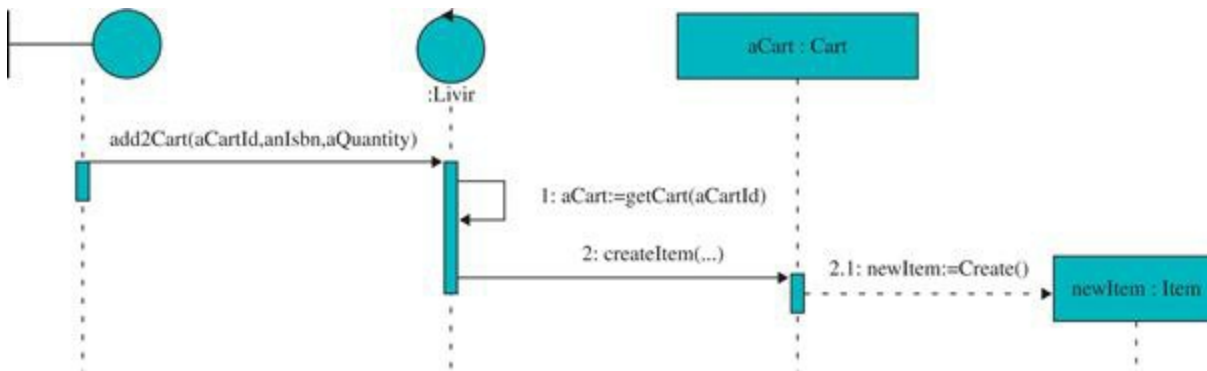


FIGURE 9.36 Sequence diagram equivalent to the communication diagram of Figure 9.35.

The sequence and communication diagrams used for dynamic modeling here have three kinds of messages:

- A message that invokes a *system operation*, which is sent by the interface and received by the controller: *add2Cart* in the example.
- *Basic messages* that invoke operations that need not be refined further: *getCart* and *Create* in the example.
- *Delegate messages* that may be used when an instance does not have visibility to the object that must perform a responsibility: *createItem* in the example.

The first and second types of messages are always present in those diagrams. The message that invokes the system operation is the root of the messages tree of the dynamic model. The basic messages are the leaves of the messages tree, which correspond to the final object queries and updates.

Basically, there is one basic message for each individual postcondition. Those messages are *basic* in the sense that they do not need to be refined further: the control flow ceases when they are reached.

The third kind of messages referred to above, *delegate messages*, are optional. They may be used if the controller, instead of sending the basic messages itself, has to delegate responsibilities to other objects that have more immediate visibility to the objects that must be queried or updated.

Delegated messages are the branches of the messages tree, and the flow usually must continue after them. The message flow only finishes when a basic message is reached.

9.4.2 Link addition

It was mentioned that when an instance is created, it must be immediately linked to another instance so that it may be accessible from the façade controller. The contract for *add2Cart* presented in Section 9.4.1 states that the recently created instance of *Item* must be linked to a book and an order. Figure 9.37 shows an evolution of the dynamic model with those two postconditions being achieved.

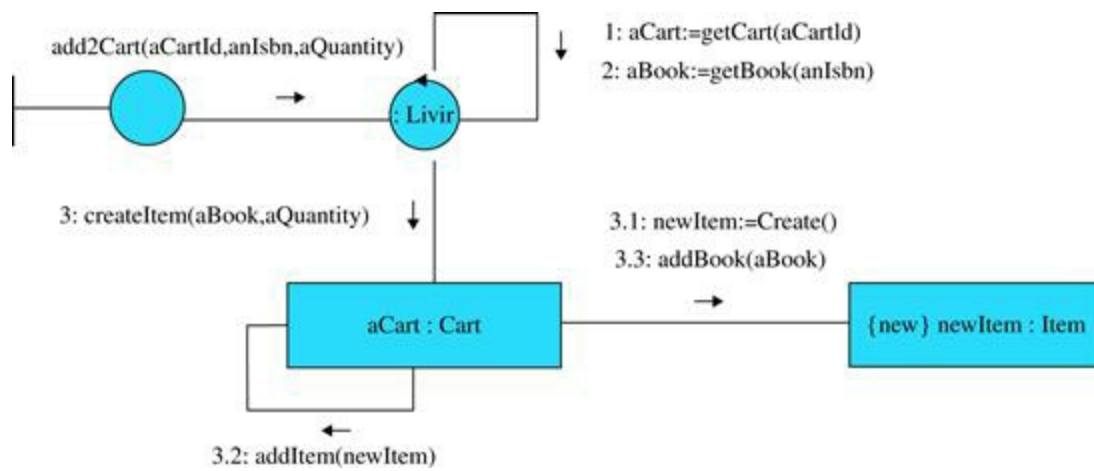


FIGURE 9.37 Evolution of the model in Figure 9.35 showing the addition of links.

Now, when the cart is performing the *createItem* method, it creates the new item (message 3.1), adds a link between itself and the new item (message 3.2), and adds a link between new item and the book identified by *anIsbn* (message 3.3).

This design could be acceptable because it reproduces almost literally what is specified by the contract. However, this would be the case only if the code generated by that design would not be manually changed later. If the final programming code is going to be directly changed some time in the future then additional precautions must be taken, and that is almost always the case.

The problem is that when an *Item* is created by a basic constructor with no parameters, it is not consistent with its definition until every mandatory attribute and link is defined for it. In the case of Figure 9.37, *newItem* is inconsistent in the interval between messages 3.1 and 3.3. The solution to this problem is to avoid having the constructors return inconsistent objects. The constructor in this case has to receive all parameters that are needed to generate a consistent instance. Therefore, the *Create* constructor is not simply a basic message in the initial conception of the term, because it must be further refined. Figure 9.38 shows a new version of the dynamic model where the constructor of *Item* produces an instance with the necessary mandatory associations.

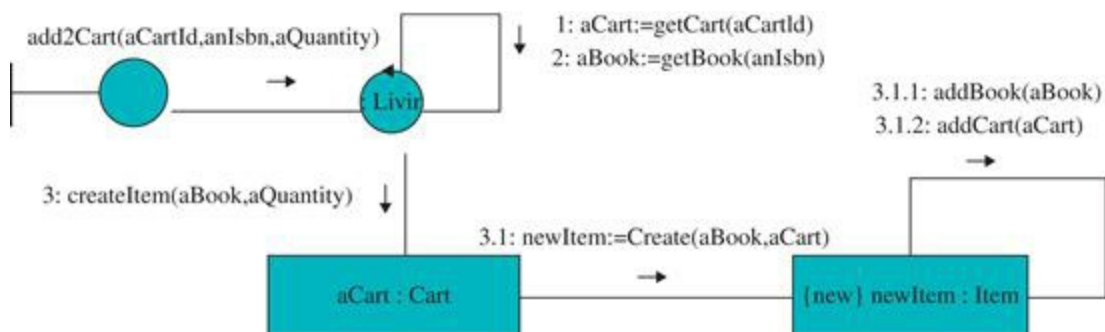


FIGURE 9.38 A variation of the model of Figure 9.37 where the constructor of *Item* produces a consistent instance.

Now, even if the final programming code is changed manually in the future, the designer knows that anytime an instance of *Item* is used it is consistent to its definition.

9.4.3 Attribute value modification

Another kind of postcondition that must be considered in dynamic modeling is the *attribute value modification*, which may be achieved by the basic *setter* message that can be sent by the object that owns the attribute or by one of its neighbors. Continuing the example, the *quantity* attribute of the new instance of *Item* must be filled with the respective *aQuantity* parameter of *add2Cart*, and the *unitPrice* attribute of *Item* must be initialized with the respective book price, as stated by the initial value declaration for that attribute in the diagram. These postconditions are added to the diagram and shown in [Figure 9.39](#).

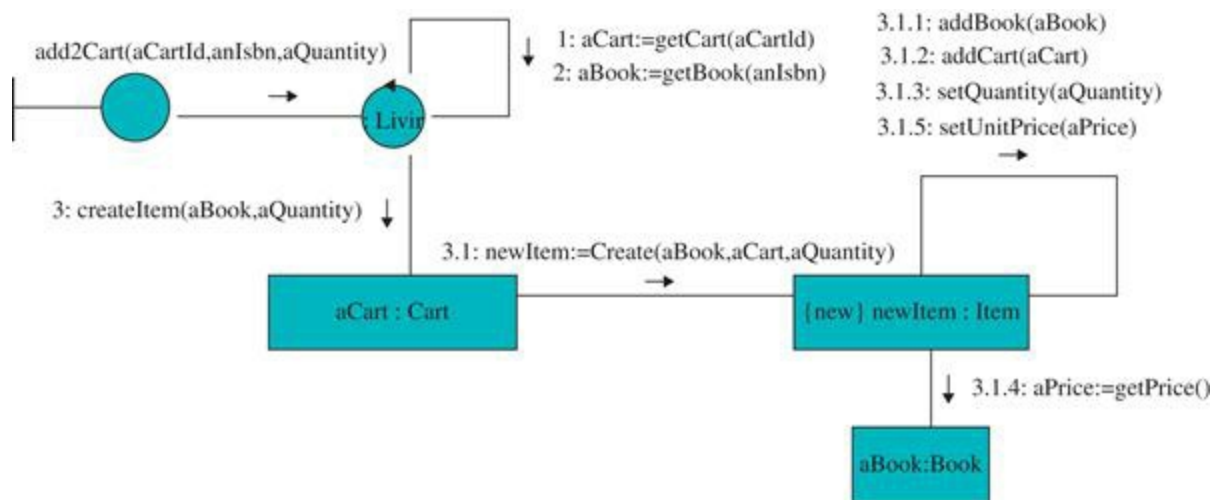


FIGURE 9.39 Evolution of the model in [Figure 9.38](#) showing attributes being changed.

9.4.4 Instance destruction

The destruction of an instance is dynamically modeled by sending a *destroy* basic message to an object. The same principles of the creator design pattern apply here: the object that destroys an instance must have a shared or composite aggregation relationship with the object that is supposed to be destroyed, or an association from one to many, or, at least, be associated to the object.

Care must be taken to avoid leaving objects with mandatory links inconsistent after an object is destroyed. If the contract is well formed no orphan objects will be left, and the communication diagrams only need to represent the postconditions that are mentioned in the contract.

[Figure 9.40](#) presents a dynamic model for the CRUD delete command for an instance of *Book*. The contract assumes that only ISBNs of instances that can be deleted may be passed as an argument:¹¹

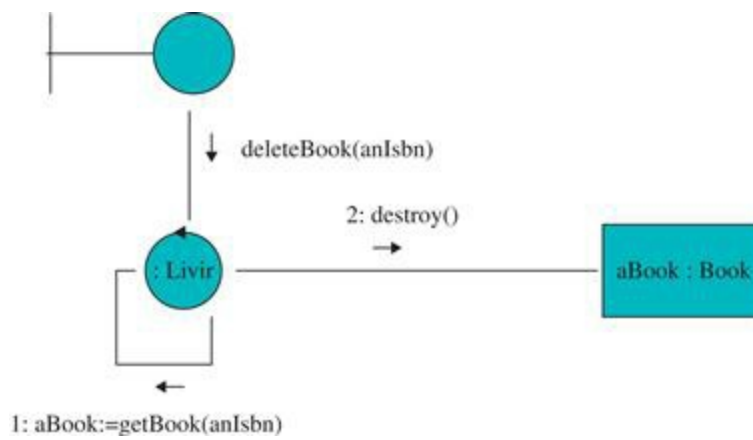


FIGURE 9.40 Communication diagram with an object being destroyed.

Context `Livir::deleteBook(anIsbn:ISBN)`

```

pre:
  book[anIsbn]->notEmpty() and
  book[anIsbn].item->isEmpty()
post:
  book[anIsbn]^destroy()
  
```

9.4.5 Removing and replacing a link

When a postcondition establishes that a link must be removed, some object in the dynamic model has to send a *remove* basic message to one of the participating instances. [Figure 9.42](#) shows an example of a dynamic model, based on the class diagram of [Figure 9.41](#), where a link is removed and replaced by another; it is the dynamic version of the following contract that specifies how a car changes its owner:

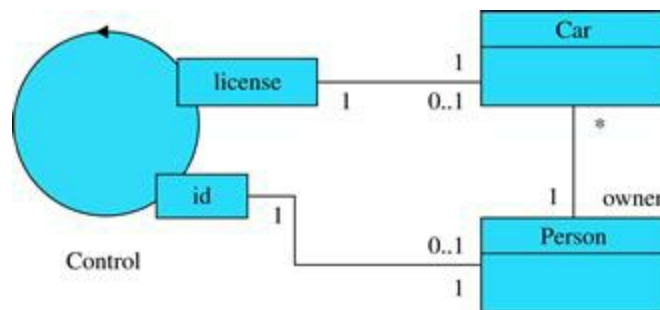


FIGURE 9.41 Reference class diagram.

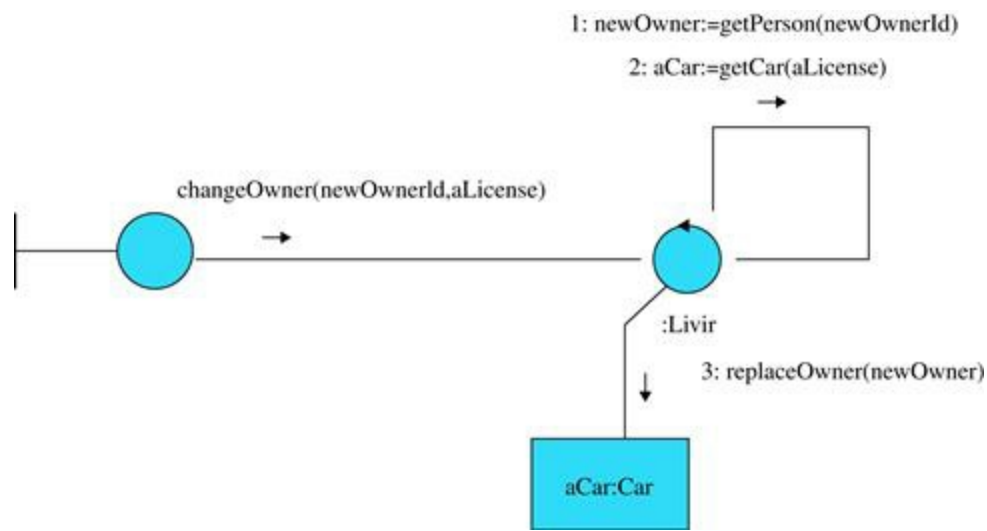


FIGURE 9.42 Communication diagram with a link replacing command.

Context Control::

```
changeOwner(newOwnerId:IdNumber, aLicense:CarLicense)
```

pre:

```
person[newOwnerId]->notEmpty() and
```

```
car[aLicense]->notEmpty
```

post:

```
car[aLicense]^removeOwner() and
```

```
car[aLicense]^addOwner(person[aNewOwnerId])
```

As the *owner* role has multiplicity 1, the command removing the link does not require the specification of the parameter.

Also, as the role has multiplicity 1, a new basic message to replace the association could be created instead of the *remove/add* sequence. That message could be called *setOwner(newOwner:Person)* or *replaceOwner(newOwner:Person)*, and it would be thread-safe. The *remove/add* sequence may be dangerous if in the short time between those commands some other process (thread) makes a query on the object that is in a temporarily inconsistent state. If a single command such as *replace* is used instead and if it is assured that while it is being performed the object is locked even for reading, then the dangerous situation described above would not occur.

9.4.6 Conditional postconditions

As explained before, sometimes a postcondition must be obtained only if the given conditions are valid. In these cases, it is possible to use conditioned messages in interaction diagrams that are similar to the guard conditions used in machine state and activity diagrams.

Resuming the *add2Cart* example, the conditional expression verifies if the cart already has one item with the book. That expression does not qualify as a derived attribute because it is parameterized (it is Boolean, but its value depends on the book), and

therefore it may be defined as a normal query in the *Cart* class:

```
Context Cart::includeBook(aBook:Book):Boolean
```

```
body:
  item->exists(aBook)
```

Figure 9.43 shows the evolution of the model of Figure 9.39, now testing if a book already belongs to the cart or not. Notice that message 3 was renamed *insertItem*, because it is not just creating a new item anymore: it may be created or incremented.

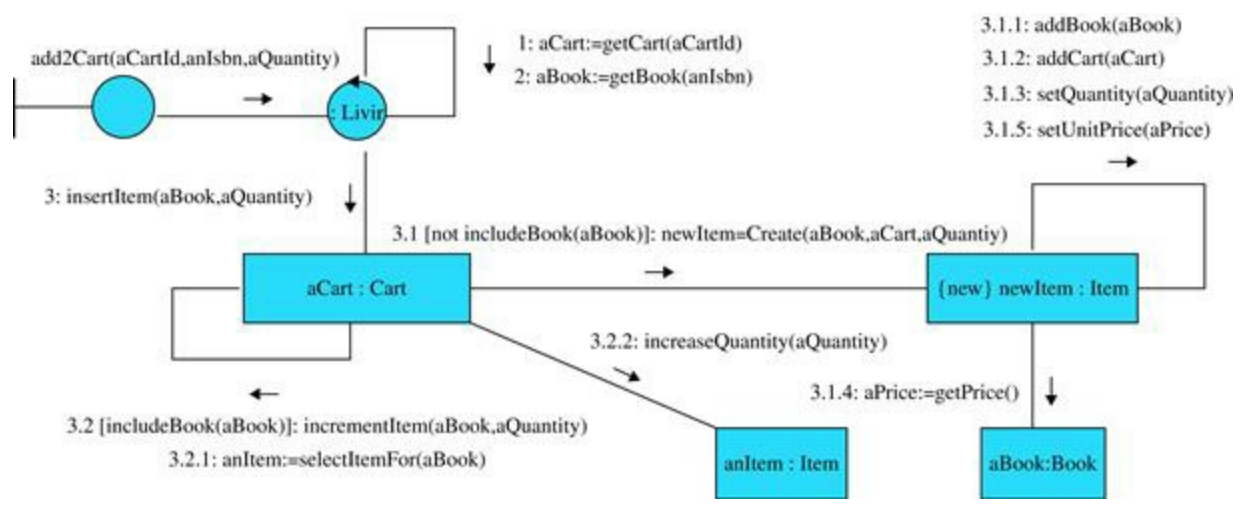


FIGURE 9.43 A complete draft of a dynamic model for the *add2Cart* contract.

If the conditional is simply an “*implies*,” then a conditional message would be sufficient in the diagram. But if the conditional involves a structure of the *if-then-else-endif* kind, as in Figure 9.43, two mutually exclusive conditional messages must be used. In Figure 9.43, message 3.1 is executed only if *includeBook(aBook)* is true, and message 3.2 is executed only if that condition is false.

Notice that messages 3.1.1 to 3.1.5 are subordinated to 3.1 and they are executed only if 3.1 is executed. Similarly, messages 3.2.1 and 3.2.2 are subordinated to 3.2 and are executed only if 3.2 is executed.

Message 3.2.1, *selectItemFor*, is another query that is useful for the *Cart* class. It may be defined as

```
Context Cart::selectItemFor(aBook:Book):Item
```

```
body:
  item->select(book=aBook)
```

Message 3.2.2 is also important: if only basic messages (as defined earlier) were used, instead of message 3.2.2, a sequence of *currentQuantity:=getQuantity()* and *setQuantity(currentQuantity+aQuantity)* would be used. However, that construction is not thread-safe. Unless concurrency is perfectly controlled, that sequence of messages could leave the quantity in an inconsistent state if it is updated between those two messages by another process. Thus, usually, instead of performing a sequence of *get* and *set* to

increment numeric attributes, it is preferable to define an *increase* basic message that simply adds the quantity passed as a parameter to the respective numeric attribute.

Up to this point, dynamic modeling produced two new queries for the *Cart* class: *includeBook* and *selectItemFor*. It also produced two delegate methods for the *Cart* class: *insertItem* and *incrementItem*. For the *Item* class, a constructor was defined, and as the *quantity* item must be incremented, a new basic message was defined too: *increaseQuantity*. The other messages in the diagram such as *getCart*, *setQuantity*, etc., are all basic messages.

However, another concern must be addressed now: Are the items, after their creation, supposed to change? This question is related to the *immutability* design pattern that establishes that an object that is not supposed to change should not implement public update methods. Specifically, in the case of the *Item* class, which has an instance created in [Figure 9.43](#), we must consider if its book, cart, price, or quantity may be changed later. That depends, of course on other dynamic models that will be constructed for other system operations, but we can assume now that *quantity* and *unitPrice* may change while the linked book and cart are immutable. Therefore, the item must not leave *addBook* and *addCart* as public methods. Private methods may be declared in UML class diagrams by adding a minus (“-”) sign before the method name. They are shown in the diagram in [Figure 9.50](#). Private methods exist but can be called only by the instance that owns them; for other objects they are not accessible.

9.4.7 Exceptions

Conditionals may also be used to model exceptions. Consider again the contract for deleting a book, but this time, instead of assuring that the book may be deleted by precondition, an exception is raised if the book is linked to at least one item:

```
Context Livir::deleteBook(anIsbn:ISBN)
```

```
pre:
```

```
book[anIsbn]->notEmpty() and
```

```
post:
```

```
book[anIsbn]^destroy()
```

```
exception:
```

```
book[anIsbn].item->notEmpty() implies
```

```
Exception.throw('Cannot delete a book that was sold')12
```

Exceptions usually are the first conditions to be tested in the dynamic model, because if one of them occurs, all other postconditions must not be obtained. Thus, a conditional message would be used to test the exception condition before anything else.

The diagram in [Figure 9.44](#) shows how to test and raise an exception if the book has items.

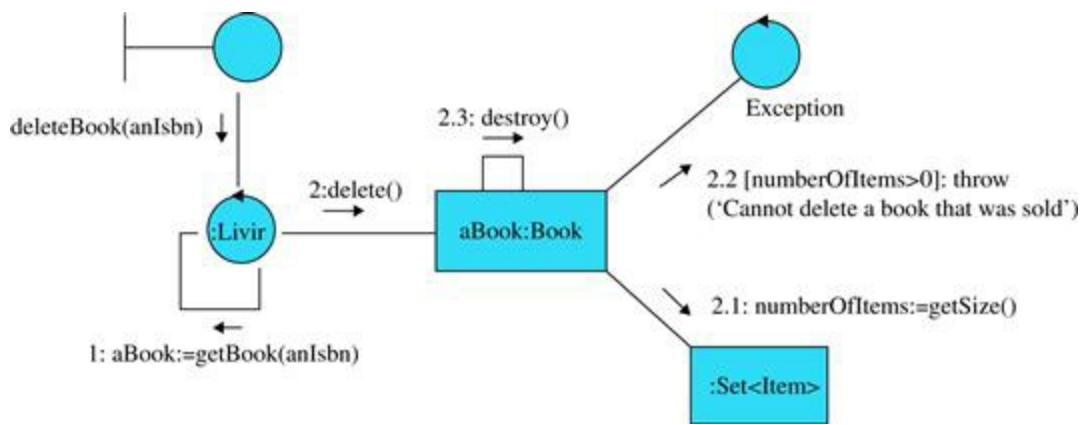


FIGURE 9.44 A dynamic model for a contract with an exception.

The model in [Figure 9.44](#) assumes that the *throw* message is terminal, that is, the execution of the system operation is interrupted by it. This means that if the exception condition is true, message 3 (the actual destructor) is not executed.

9.4.8 Postconditions over collections

When system commands have contracts with postconditions that specify updates over collections of objects, the iteration structure may be used in the communication diagram to indicate that a message is iteratively sent to all of the elements in the collection. For example, the following system command increases the price of every book by *aPercentage*:

```
Context Livir::raisePrices(aPercentage:Percentage)
```

```
post:
  book->forAll(aBook|
    aBook^setPrice(aBook.price@pre*(1+aPercentage))
  )
```

The diagram in [Figure 9.45](#) is a dynamic model for that contract.

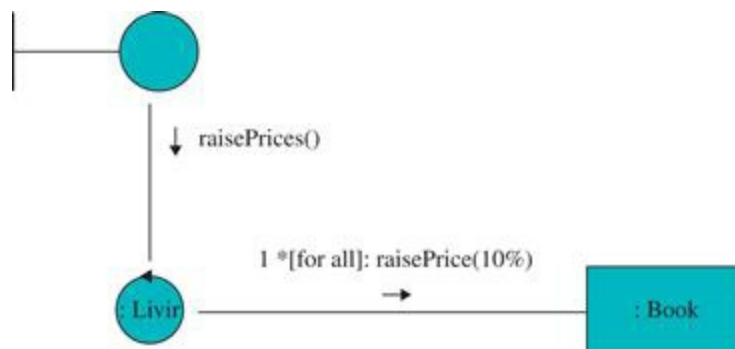


FIGURE 9.45 Dynamic model for a contract with a postcondition over a collection of objects.

Again, the choice to define a new basic message *raisePrice* instead of a sequence of *getPrice* and *setPrice* is because it is thread-safe. The basic message *raisePrice* changes the value of the attribute *price* by multiplying it by 1 plus the parameter *aPercentage*.

Another common situation occurs when the iteration must not occur over every object in the collection, but only over those that satisfy a given condition. The following command raises by *aPercentage* the prices of books with more than *aPageCount* pages:

```
Context Livir::raiseThickBooks(aPercentage:Percentage; aPageCount:Natural)

post:
  book->select(pageCount>aPageCount)->forall(aBook|
    aBook^setPrice(aBook.price@pre*(1+aPercentage))
  )
```

The communication diagram for that contract is shown in [Figure 9.46](#).

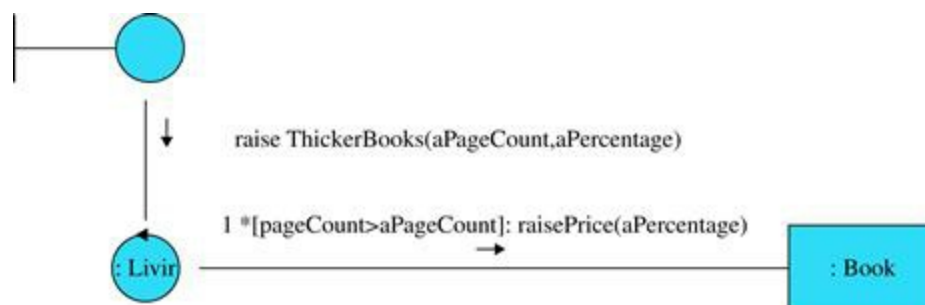


FIGURE 9.46 Communication diagram with iteration and selection.

9.5 System queries

The system query contracts usually define only combinations of data that are obtained from the objects. It is possible, although not always useful, to develop communication or sequence diagrams for such contracts, as, for example, the communication diagram of [Figure 9.47](#) that was built for the contract of the *listBooks* system query below:

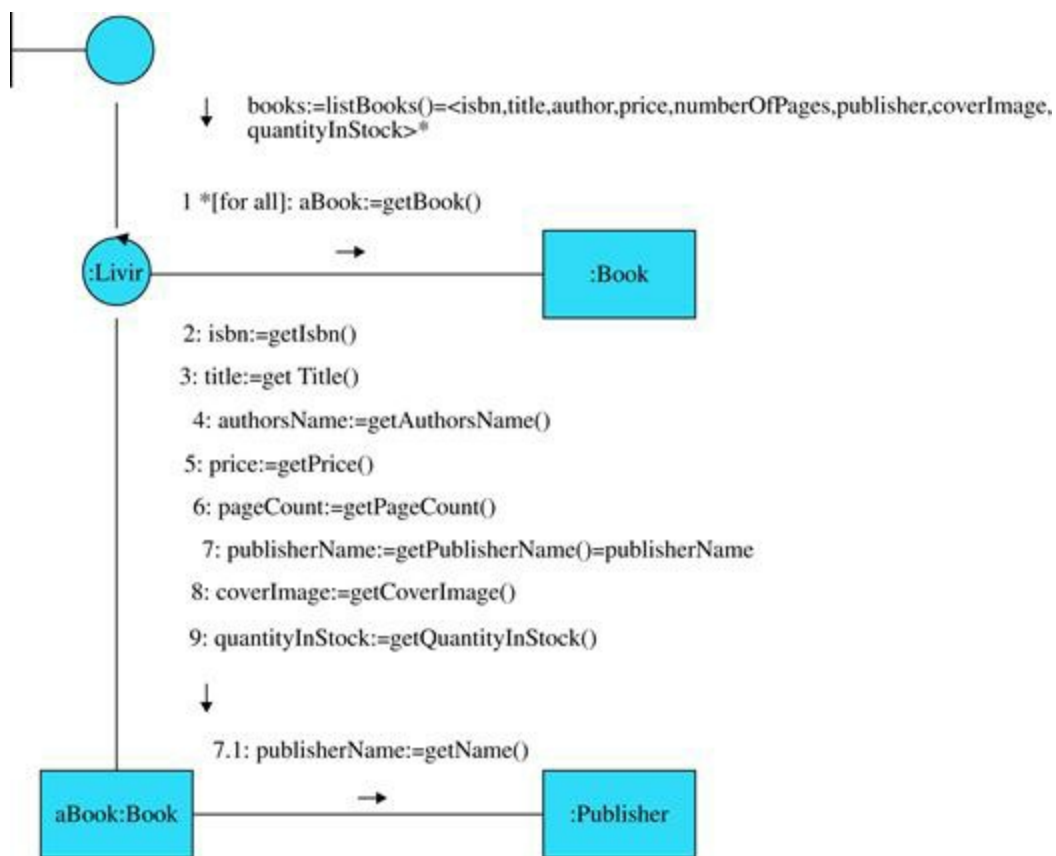


FIGURE 9.47 Collaboration diagram for a system query.

```
Context Livir::listBooks():Set
```

```
body:
book->collect(aBook|
  Tuple {
  isbn=aBook.isbn,
  title=aBook.title,
  authorsName=aBook.authorsName,
  price=aBook.price,
  pageCount=aBook.pageCount,
  publisherName=aBook.publisher.name,
  coverImage=aBook.coverImage,
  quantityInStock=aBook.quantityInStock
  }
)
```

The most difficult aspect related to these diagrams when they are used to model system queries is that they do not ease the representation of data composition. For example, in [Figure 9.47](#) the controller sends eight messages to each instance of *Book*; each message produces a different return. What does the controller do with all those values? We know, from the contract of the query, that it forms a tuple with the eight values and groups the tuples into a set that is the final return data for the query. But representing that on the diagram would only repeat what is already known in the contract. Therefore,

the utility of these diagrams in the case of queries is limited.

There are some important issues about modeling queries, however. First of all, three types of query messages may be identified:

- The *system query*, which is implemented in the controller and is called directly by the interface. As the domain objects are encapsulated into the domain tier and the interface deals with fields and buttons, system queries should receive alphanumeric data as parameters and return alphanumeric data as well.
- *Basic queries*, which consult actual attributes and links corresponding to the basic *get* queries.
- The *intermediate queries*, which correspond to the derived attributes and associations. Derived attributes return alphanumeric data and derived associations return objects or data structures holding objects. Another type of intermediary query is the parameterized query, which cannot be defined as a derived attribute or association, such as *includeBook* in [Figure 9.43](#).

In [Figure 9.47](#), the only intermediary query is *getPublisherName*, sent by the controller to the book. The book, during its turn, sends the basic query *getName* to its publisher. Thus, *publisherName* may be considered a derived attribute of *Book*.

With regard to the question, “Should the intermediary query return the publisher name or the publisher itself?” some considerations must be taken into account:

- If a single attribute of an object is needed, avoid returning the object, because it may generate a risk of nonconformance to the “*Don’t talk to strangers*” pattern. If a single alphanumeric value is needed, then it is better to define it as a derived attribute such as *publisherName* in the *Book* class.
- When a set of attributes or values must be obtained from one or more objects, and the intention is to operate over them, for example, performing a sum, or finding the highest value or the average, it is preferable to perform the operation on the class that has the most immediate access to the values. In other words, the operation should be performed as soon as possible, and the already processed result should be returned as a derived attribute. For example, *totalValue*, which is a derived attribute of *Cart*, must be calculated by the cart itself, not by the customer or the controller. It would be a bad design if the cart returned the items to the controller for it to perform the sum over the subtotals.
- When collections of values with a complex structure must be returned and cannot be transformed into a single alphanumeric value, it is preferable to return that information as a collection of objects, defining it as a derived association, because that way, the methods that the classes already have to perform calculations and operations over the data remain available. However, this choice can possibly create more coupling, and it must be used only if the other choices are really not applicable.

9.6 Delegation and low coupling

To this point, we have seen some techniques for building communication diagrams. But in most situations there is more than one design option, and more than one way to send

a message.

As discussed earlier, sometimes the object that has control does not have visibility to the object that can perform the responsibility. In this case, two opposite design approaches can be identified:

- The object that has control tries to obtain a reference to the object that can perform the responsibility, and communicate directly with it.
- The object delegates the message to another object that is closer to the object that can perform the responsibility.

In order to understand the difference between these two approaches, imagine that Mary is the boss of Peter. The boss wants to order a buffet for the office party, but she does not know any catering companies. The boss knows that Peter has contact with some catering companies, and then, two approaches are possible:

- The boss asks Peter for the phone numbers of the companies and makes the arrangements herself. The only thing that Peter does is to pass his contacts to the boss.
- The boss sends Peter the parameters for the party: how many people, what kind of food, how much to spend, etc., and asks Peter to do the arrangements. The party happens, and the boss was never informed about the catering contact: he delegated to Peter.

Although in real life it may be interesting to make contact with many people and companies, this is not the case in object-oriented systems. When objects are highly coupled, the system is more complex than it really needs to be. Thus, it is necessary to avoid creating connections between objects that were not connected in the first place.

The first approach discussed just above is called *concentration*. With it the object that has execution control manages to obtain references to all objects needed and performs the operation itself. The second approach is called *delegation*, and it consists of making the objects delegate responsibilities when they cannot perform them. The second approach is usually preferable, because it provides more potential for reusing classes and methods, because the intermediary queries or delegated methods that this approach creates usually are reusable.

A concrete example of this is shown below. Consider a system query `getCartTotal` defined as follows:

```
Livir::getCartTotal(aCartId:CartId):Money
```

```
body:  
  cart[aCartId].item->sum(quantity*unitPrice)
```

The system query contract above has as objective of obtaining the total value of a given cart (the definition of the contract intentionally does not use the derived attributes defined in [Figure 9.34](#), assuming that they are not yet defined). This value must be calculated from the value and quantity of each of the items of the order.

With the concentration approach, the whole process would be implemented in the *Livir* class. This class would obtain all necessary data to perform the calculation and return the result to the interface. The concentration communication diagram has all the

messages that are sent by *Livir*. No delegation happens, as no object sends messages besides the *Livir* instance. The pseudocode for that system query could be described as follows:

```
CLASS Livir

  ASSOCIATION VAR carts:MAP[CartId->Cart]

  ...

  METHOD getCartTotal():Money
    LOCAL VAR aCart:Cart
    LOCAL VAR cartsItems:SET<Item>
    LOCAL VAR total:Money
    aCart:=carts.at(aCartId)
    cartsItems:=aCart.getItem()
    total:=0
    FOR EACH item IN cartsItems DO
      total:=total+(item.getValue()*item.getQuantity())
    END FOR
    RETURN total
  END METHOD

END CLASS
```

We can take the following away from this code:

- It solves the problem.
- It produces no reusable software element besides the query itself that is implemented in the *Livir* class.
- It increases coupling among classes, because *Livir* acquires visibility to *Item*, which was not the case in the class diagram of [Figure 9.34](#).

By using the delegation approach, the query can be implemented correctly as well, but at the same time, new reusable structures with more cohesion and less coupling are created.

In the first place, the *Livir* class must be prevented from having access to any instance of *Item*, since there are no connections between those classes in the class diagram. In order to perform the required responsibilities, *Livir* must delegate the calculation to the cart. As the return value is *Money* (a specific alphanumeric), creating this intermediate query for the *Cart* class is equivalent to defining a derived attribute for it with the following OCL specification:

```
Context Cart::total:Money

  derive:
    item->sum(subtotal)
```

Besides creating a derived attribute in *Cart*, a derived attribute should also be created

in the *Item* class, in order to return the subtotal:

```
Context Item::subtotal:Money
```

```
  derive:  
    unitPrice*quantity
```

Now, with those two derived attributes, the pseudocode may be defined with responsibilities distributed among the classes:

```
CLASS Livir
```

```
  ASSOCIATION VAR carts:MAP[CartId->Cart]
```

```
  ...
```

```
  METHOD getCartTotal(aCartId:CartId):Money
```

```
    RETURN carts.at(aCartId).getTotal()
```

```
  END METHOD
```

```
END CLASS
```

```
CLASS Cart
```

```
  ASSOCIATION VAR items:SET<Item>
```

```
  ...
```

```
  METHOD getTotal():Money
```

```
    LOCAL VAR total:Money
```

```
    total:=0
```

```
    FOR EACH item IN items DO
```

```
      total:=total+item.getSubtotal()
```

```
    END FOR
```

```
    RETURN total
```

```
  END METHOD
```

```
END CLASS
```

```
CLASS Item
```

```
  ATTRIBUTE VAR quantity:Natural
```

```
  ATTRIBUTE VAR unitPrice:Money
```

```
  ...
```

```
  METHOD getSubtotal():Money
```

```
    RETURN quantity*unitPrice
```

```
  END METHOD
```

```
END CLASS
```

With that approach, most of the calculation happens in the class that effectively has the most immediate access to the necessary information, which is *Cart*. Thus, the

calculation procedure here is more reusable than the one produced by the concentration approach, because it is now inside the *Cart* class. If the *Cart* class is reused it can calculate its own total without depending on a class that is not a component of it (*Livir*). The same is true for *Item*, which now knows how to calculate its own subtotal.

In the case of system commands, the low coupling principle is also obtained when the designer chooses to delegate instead of returning an object. Consider again the class diagram of Figure 9.34, but suppose that the role from *Cart* to *Item* is marked with {*ordered*}. The communication diagram of Figure 9.48 shows the system command *deleteLastItem*, using the concentration approach. Figure 9.49 shows the same command with the delegation approach, avoiding unnecessary coupling among classes.

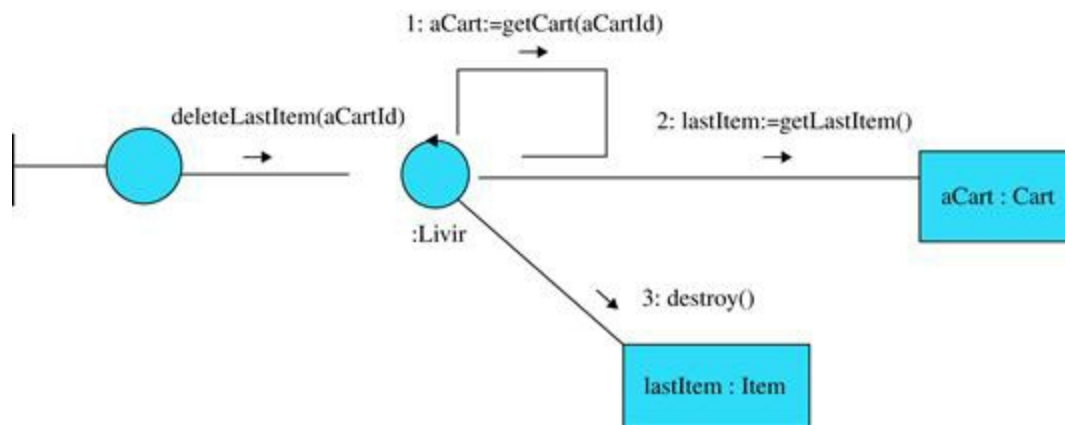


FIGURE 9.48 Concentration design style.

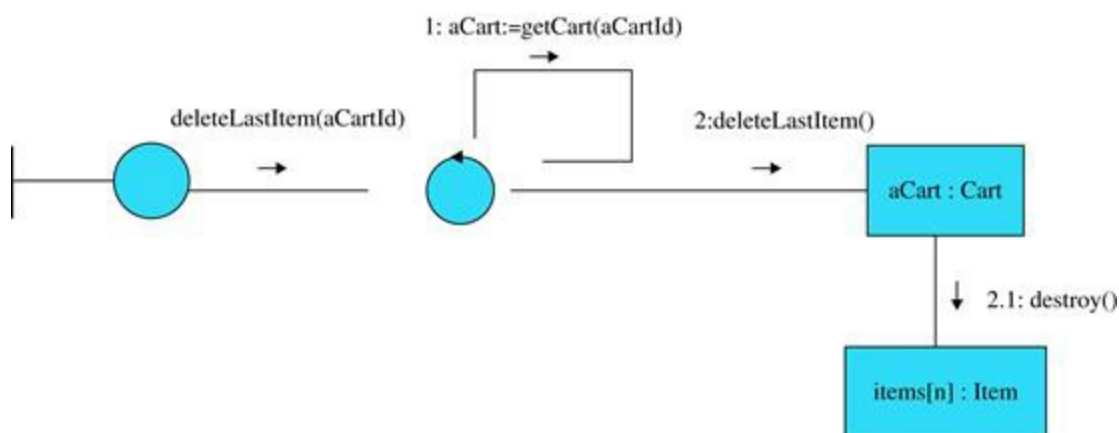


FIGURE 9.49 Delegation design style.

The price for using delegation is usually having more methods declared in the intermediary classes. However, reusable methods are usually produced to counteract this issue. The price for using the concentration approach is having a design where responsibilities are concentrated in the façade controller instead of being distributed among the objects; this produces a model that is hard to evolve and maintain in the long term.

9.7 Design class diagram

One of the goals of the logical design is to build and refine the design class diagram (DCD), which is created from the conceptual model and from information obtained during dynamic modeling (when the interaction diagrams for the contracts are being built).

The first version of the DCD is an exact copy of the conceptual model, which will be modified later. The basic modifications that are made in the DCD during domain tie design are the following:

- *Methods are added.* During analysis, only system commands and queries were discovered and added to the façade controller class. During design, delegate and basic methods belonging to other classes are added.
- *Associations are directed.* During analysis, associations were nondirectional. During design their direction is determined by the direction of the messages flowing between objects in the interaction diagrams.
- *Attributes and associations may be detailed.* It is not mandatory to present details about attributes and associations during analysis. These details may be added during design. Additionally, in analysis only abstract data types are used to specify association roles, and they may be replaced by concrete data types during design (for example, replacing *sequence* by *array list* or *linked list*).
- *The structure of classes and associations may change.* It may be necessary to create new classes to implement certain design structures, such as strategies and services, for example. Thus, it is possible that the class structure of the DCD does not correspond exactly to the structure of the conceptual model, in some cases.
- *Possible creation of protected and private attributes.* In the conceptual model all attributes are public, because they represent static information and not behavior. However, when the dynamic aspects of objects are represented, it may be necessary to work with private attributes in order to encapsulate internal states that determine the behavior of some objects. These attributes sometimes may not be directly accessible through a getter or setter, but their influence on some methods is felt.

During the construction of a communication diagram, each time an object receives a delegated message, its class must implement the corresponding method.

In [Figure 9.43](#), the communication diagram shows that a number of messages must be implemented as methods in some classes:

- *Livir* must implement:
 - *add2Cart(aCartId:CartId, anIsbn:ISBN, aQuantity:Natural)* – the system command.
 - *getCart(aCartId:CartId):Cart* – basic query.
 - *getBook(anIsbn):Book* – basic query.
- *Cart* must implement:
 - *insertItem(aBook:Book, aQuantity:Natural)* – delegate command.
 - *includeBook(aBook:Book):Boolean* – query or predicate.
 - *incrementItem(aBook:Book, aQuantity:Natural)* – delegate command.
 - *selectItemFor(aBook:Book):Item* – query.

- *Item* must implement:
 - *Create(aBook:Book, aCart:Cart, aQuantity:Natural)* – constructor.
 - *addBook(aBook:Book)* – private basic command.
 - *addCart(aCart:Cart)* – private basic command.
 - *setQuantity(aQuantity:Natural)* – basic command.
 - *increaseQuantity(aQuantity:Natural)* – basic command.
 - *setUnitPrice(aValue:Money)* – basic command.
- *Book* must implement:
 - *getPrice():Money*– basic query.

The DCD may also include information about the direction of the associations if the designer chooses not to implement every association as bidirectional. This is usually the case because unidirectional associations tend to be simpler to implement and more efficient when compared to bidirectional associations.

Deciding whether an association is unidirectional or bidirectional depends on the flow of messages on the interaction diagrams. No association should be, in principle, navigable in the direction of the controller: the controller sees the classes, but they do not see it. Other associations may be unidirectional if the messages only flow in one direction, and bidirectional if messages flow in different directions, even if that occurs in different diagrams. [Figure 9.50](#) presents an evolution of the diagram in [Figure 9.34](#) where methods and association directions discovered in the dynamic model of [Figure 9.39](#) are included.

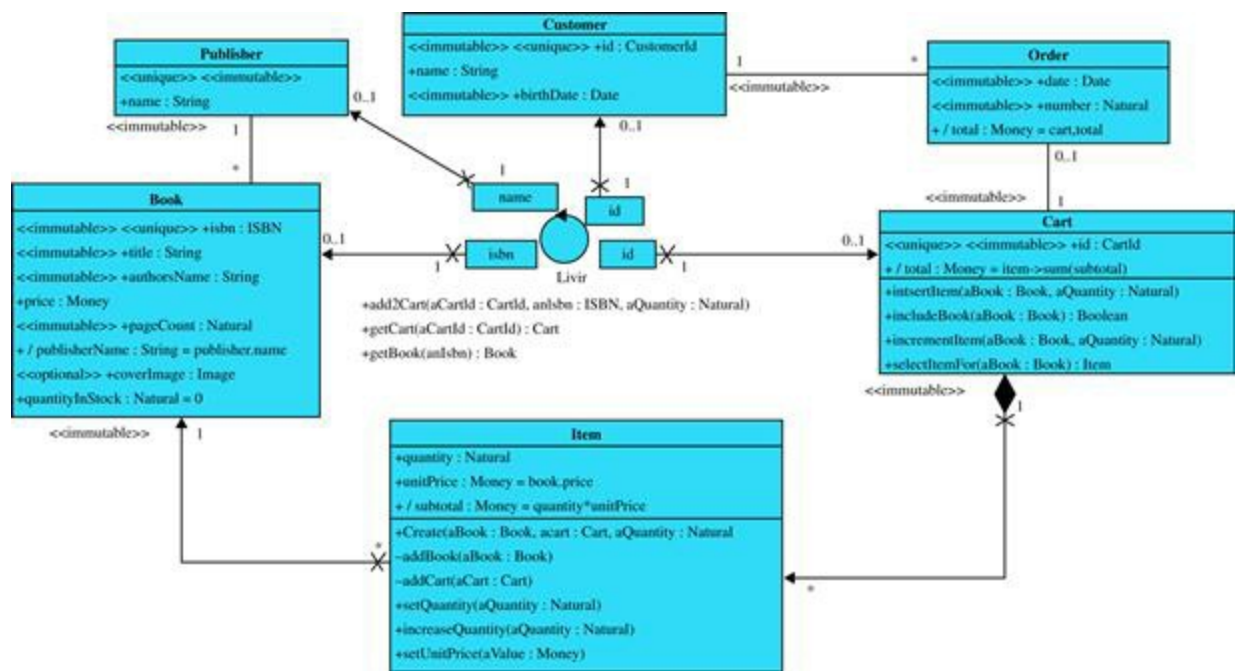


FIGURE 9.50 Design class diagram with methods and association direction.

Not every association in [Figure 9.50](#) is directed because the diagram in [Figure 9.39](#) shows messages being sent just from the controller to *Cart*, *Cart* to *Item*, and *Item* to *Book*. Associations that were not followed by messages in the communication diagram are left undefined until another diagram eventually decides on their direction.

If other collaboration diagrams developed for different system operations require messages navigating in the opposite direction as those that were already discovered, then the respective associations should be bidirectional.

As the direction of the associations depends on the direction of the messages sent in the dynamic models, it would not make sense to try to define that direction during preliminary conceptual modeling (Chapter 3). At that time the information available is insufficient to make those design decisions. The same is valid for deciding on which methods a class must implement.

It is assumed by default that unidirectional associations have no restrictions on their origin. However, that is not always the case. It is true, for example, for the association from *Item* to *Book*, which has no multiplicity restriction at its origin. But the association from *Cart* to *Item* is restricted at the origin, because it demands that an item is linked to exactly one cart. If the association is implemented simply as a variable in the *Cart* class holding a set of *Items*, then that constraint would not be easily assured. This is why, in the next chapter, unidirectional associations with restrictions at the origin are handled as bidirectional associations: they may not be navigated in both directions, but they must be implemented at both ends.

The logical design activity finishes when the DCD has enough information to implement the domain tier classes. That usually happens when all system operation contracts have been examined and their dynamic models incorporated into the design.

9.8 The process so far

	Inception	Elaboration
Business Modeling	Build a general view of the system: <ul style="list-style-type: none">• Build a business use case diagram and determine the automation scope for the project.• Build preliminary activity diagrams for business use cases.• Build preliminary state machine diagrams for key business objects.	

(Continued)

	Inception	Elaboration
Requirements	<p>Prepare the system use case diagram (functional requirements):</p> <ul style="list-style-type: none"> Identify the system actors from the business use case model. Identify the system use cases from the business use case model, and the activity and state machine diagrams from business modeling. <p>Identify nonfunctional requirements as use case annotations:</p> <ul style="list-style-type: none"> Identify the main business rules associated to use cases. Identify the main quality issues associated to use cases. <p>Identify supplementary requirements.</p>	<p>Detail requirements by expanding use cases:</p> <ul style="list-style-type: none"> Identify the main flow. Identify alternate flows: variants and exception handlers.
Analysis and Design	<p>Prepare the preliminary conceptual model by observing system use cases and the concepts needed by them.</p>	<p>Elaborate the system sequence diagrams:</p> <ul style="list-style-type: none"> Represent the main flow of a use case as a system sequence diagram. Represent the system commands and queries using stateful or stateless strategies. Complete system sequence diagrams with alternate flows. <p>Refine the conceptual model:</p> <ul style="list-style-type: none"> Identify concepts, attributes, and associations in the text of expanded use cases. Detail attributes and associations with stereotypes, multiplicity, and constraints as needed. Organize the model by using inheritance, association classes, and temporal specifications. Add invariants as needed. Improve the conceptual model with the application of analysis patterns. <p>Write system operation contracts for commands and queries in system sequence diagrams:</p> <ul style="list-style-type: none"> Identify preconditions and exceptions based on invalid parameters and complementary constraints. Identify postconditions for commands and returns for queries.

(Continued)

	Inception	Elaboration
		<p>Design the domain tier:</p> <ul style="list-style-type: none"> • Create a sequence or communication diagram for each system command contract. • Use those diagrams to decide which methods to implement in each class. • Inspect those diagrams to discover which associations can be unidirectional and define them as such. • Look at system query contracts and decide which delegate queries could be implemented; some of them could be derived attributes or derived associations. • Produce or refine the design class diagram.
Implementation		
Test		
Project Management	<p>Estimate total effort, ideal calendar time, and average team size for the project.</p> <p>Estimate the duration and quantity of iterations for each phase.</p> <p>Prepare the phase plan and iteration plan for the first iteration.</p>	

9.9 Questions

1. What are the six kinds of responsibilities that objects may have? What kind of method is used to perform each responsibility?
2. What are the four kinds of visibility that may exist between objects? Are they symmetrical? How does each of them influence high coupling among classes?
3. Read again the last paragraph of [Section 9.3.2](#). Try to draw a communication diagram that illustrates that situation. Why is it a worse design than the diagram presented in [Figure 9.29](#)?
4. Explain the influence of contract preconditions and exceptions in dynamic models. How do they affect the complexity of the dynamic model?
5. Draw a sequence diagram equivalent to the one in [Figure 9.43](#).

¹That is valid for normal attributes belonging to classes of the conceptual model, though. During the construction of the DCD, new design attributes that represent the internal state and influence the behavior of an object may become necessary and those attributes do not necessarily have to be accessed by other objects: in that case, they should not have getters (they are private or protected).

²Remember that not every attribute must have a setter. Immutable attributes should not have a setter. Derived attributes never have setters, and private attributes also should not have public setters.

³The *add* and *remove* prefixes could be used here even for associations to 1, because conceptually they are links like any other. However, usually associations to 1 do not implement commands such as *add* and *remove*: they implement the *set* or *replace* command instead, which can replace the current mandatory link with another, leaving the role always consistent.

⁴It could be considered that a 0..1 role is filled with a single object or a null value. However, that is not coherent with the interpretation of other multiplicities such as, for example, 0..2 or *, where 0 stands for the empty set and not for the null value. To avoid such inconsistency it is preferable to consider that multiplicity 0..1 refers to a set that can be empty or contain a single element.

⁵And additionally to a set that contains an order, even if that option would not be used often.

⁶A *set* by default.

⁷In this text it is assumed that the operations available for sets and other collections are the ones defined by OCL ([Object Management Group, 2010](#)). However, if different languages or packages are used, a different set of operations would be available.

⁸The association probably would be physically implemented not as two sets, though (see [Chapter 10](#)). Here the visibility means that two sets of objects may be obtained by an instance; this does not mean that they are implemented like that.

⁹A summarization of the *Law of Demeter* ([Lieberherr & Holland, 1989](#))

¹⁰This is true in normal conditions. However, if the system is subjected to complex race conditions and if security is a big issue, sometimes double-checking would be advisable.

¹¹For example, the *delete* command could be called only for books selected from a list that contains only books that can be deleted, i.e., books with no items linked to them.

¹²Usually exceptions would not receive the very message as a parameter, but an exception code that refers to a list of messages. However, to keep examples simpler they are left here as they are.

CHAPTER

E 10

Code Generation

This chapter presents some techniques to automatically or systematically generate source code from design diagrams, specifically for the design class diagram and communication diagrams. Structural rules are presented to explain how code may be generated for a class's structures, such as attributes and associations. A significant part of the chapter is dedicated to the explanation of the many kinds of associations and the code generation patterns available for them. Finally, the chapter explains how running code may be generated for interaction diagrams that represent dynamic models.

Keywords

Automatic code generation; delegation; implementation; programming

Key Topics in this Chapter:

- Code generation for classes, attributes, and associations
- Code generation for delegated methods and system operations
- Patterns for filtered queries

10.1 Introduction to code generation

The *Implementation* discipline of UP includes code generation activities for design models. It is necessary to generate code for the classes from the domain tier, and also for the other technological tiers of the system. This chapter concentrates on code generation for the domain tier.

Once the communication diagram and the DCD are produced, code generation is an activity that may be systematized to a point that it can be practically done automatically. Automatic code generation is feasible for domain classes, which perform all the logical processing specified by the system operation contracts.

This chapter presents rules for code generation for the DCD and communication diagrams. Examples are presented in pseudocode, which may be translated to most programming languages (preferably object-oriented ones).

10.2 Classes and attributes

DCD classes are usually directly converted into programming language classes. Class attributes are converted into *private instance variables* in the respective class.

As explained in [Chapter 6](#), attribute types should be alphanumeric (such as *Integer*, *Real*, *String*, *Boolean*, etc.), primitive (such as *Date*, *Money*, *Isbn*, etc.), or enumerations (such as *CalendarDay*, *Gender*, *PhoneType*, etc.).

If other objects can access the attribute then it should be implemented with a getter method. If the attribute can be updated then it must be implemented with a setter, which can be a straight *setAttribute* method or another type such as *incrementAttribute* if the attribute is numeric, for example. [Figure e10.1](#) presents an example of a design class that is used to show how programming code is generated.

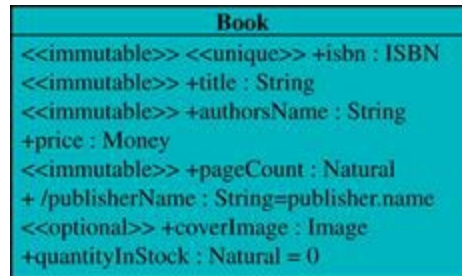


FIGURE E10.1 A reference class for code generation.

The following code corresponds to the pseudocode implementation of the class shown in [Figure e10.1](#). First, the attributes of the class are transformed into programming code private attributes:

```
CLASS Book

PRIVATE ATTRIBUTE VAR isbn:Isbn
PRIVATE ATTRIBUTE VAR title:String
PRIVATE ATTRIBUTE VAR authorsName:String
PRIVATE ATTRIBUTE VAR price:Money
PRIVATE ATTRIBUTE VAR pageCount:Natural
PRIVATE ATTRIBUTE VAR coverImage:Image
PRIVATE ATTRIBUTE VAR quantityInStock:Natural
```

...

Observe that all attributes, except for *publisherName*, which is derived, are defined as pseudocode attributes. Most languages would not differentiate *ATTRIBUTE VAR* from *ASSOCIATION VAR*. In this book those names are used to make a clear distinction between variables that represent attributes and variables that implement association roles. However, for most languages any instance variable would be simply declared as *VAR*, or an equivalent language specific expression.

On the other hand, most languages differentiate between *PRIVATE* and *PUBLIC* variables and methods. Whenever private variables are allowed, they are the best choice, because this way, attributes are encapsulated into the object, and they may not be changed by objects that have no authorization to do that.

Now, we must consider how the class attributes are going to be updated. Some of them can be updated virtually at any time, while others should be defined only at creation time and prevented from changing after that: these are *immutable attributes*. In [Figure e10.1](#) attributes that must not be changed after the object is created are stereotyped as <<

immutable>>.

A set of *setters* and one¹ *constructor* may be defined following these recommendations:

- The constructor must receive as parameters the initial values for all attributes that are not optional, derived or with a defined initial value (if any).
- The implementation of the constructor initializes attributes with initial values with the respective values (if any).
- The constructor must receive as parameters the objects to fill the mandatory roles (if any) of the associations (later this will be shown in a more complete example).
- Only attributes that are not derived and immutable should define one or more setters (*setAttribute*, *incrementAttribute*, etc., depending on the case).

Thus, the following code may be created as a continuation of the definition of the implementation class that started above:

```
...
CONSTRUCTOR METHOD create(anIsbn:Isbn; aTitle:String,
    anAuthorsName:String; aPrice:Money; aPageCount:Natural)
    isbn:=anIsbn
    title:=aTitle
    authorsName:=anAuthorsName
    price:=aPrice
    pageCount:=aPageCount
    quantityInStock:=0
END CONSTRUCTOR METHOD
METHOD setPrice(aPrice:Money)
    price:=aPrice
END METHOD
METHOD raisePrice(aPercentage:Percent)
    price:=aPrice*(1+aPercentage)
END METHOD
METHOD setCoverImage(aCoverImage:Image)
    coverImage:=aCoverImage
END METHOD
METHOD increaseQuantityInStock(anIncrement:Integer)
    quantityInStock:=quantityInStock+anIncrement
END METHOD
...
```

Notice that inside the constructor, each attribute is assigned to the respective argument and that *quantityInStock* is assigned 0 (its initial default value).

Only three attributes may be updated. The attribute *price* has two setters: *setPrice*, which defines a brand new price independent from the original one, and *raisePrice*, which raises the price by a percentage.

The *quantityInStock* attribute is updated only by adding an increment to it. As the *anIncrement* parameter may be negative, decrements may be done with this method as

well.

Let us suppose now that the design produced interaction diagrams that demonstrate that all attributes of this class must be accessed by other objects. In this case, all attributes must implement a *getter* method, even the derived attribute. Thus, the following code may be a continuation of the code started above:

```
...
METHOD getIsbn():ISBN
  RETURN isbn
END METHOD
METHOD getTitle():String
  RETURN title
END METHOD
METHOD getAuthorsName():String
  RETURN authorsName
END METHOD
METHOD getPrice():Money
  RETURN price
END METHOD
METHOD getPageCount():Natural
  RETURN pageCount
END METHOD
METHOD getPublisherName():String
  RETURN ...2
END METHOD
METHOD getCoverImage():Image
  RETURN coverImage
END METHOD
METHOD getQuantityInStock():Natural
  RETURN quantityInStock
END METHOD
```

At this point the reader could be asking why *authorsName* is still an attribute typed as *String* and not a class with a many-to-many association to *Book*. The reason is that the example is simulating an ongoing project and not the final version of it. Up to this point, the only use cases examined did not require authors to be addressed as a complex concept. The *authorsName* attribute is perfectly adequate for the functionalities discovered to this point. We may even believe that this would change as other use cases are examined in the future, but as [William of Ockham \(1495\)](#) said “*Numquam ponenda est pluralitas sine necessitate*” (plurality must never be posited without necessity). This also adheres to the agile principle of never implementing functionality before it is necessary just because it would be easy to do so now.

10.3 Unidirectional associations

Unidirectional associations with no multiplicity constraints at the origin and unidirectional associations from singletons such as the façade controller may be implemented similarly to attributes as *instance variables*, and if necessary they should have methods for updating and querying.

However, if there are multiplicity constraints at the origin role, these associations must be handled as *bidirectional*, even if they are navigable only in one direction. Figure e10.2 shows an example of that situation, where although the association is only navigable from *Car* to *Person*, every car must be linked to a single person. If the association is implemented as an instance variable in the *Person* class, it will be hard to assure that a car belongs to a single person.



FIGURE E10.2 Unidirectional association with multiplicity constraint at the origin role.

There are still some considerations we must discuss about the differences between an attribute and a unidirectional association. First, attributes are always implemented by variables whose types are alphanumeric, primitive, or enumerations. Associations, on the other hand, are implemented by variables whose types are domain classes (in the case of associations to one) or data structures (in the case of associations to many).

In addition, considering different role multiplicities and other features of associations, there are distinctions to be made regarding the methods to be implemented for each type of association.

Code generated for transient or persistent associations and attributes is the same. The only difference between these kinds of elements resides in the way they are stored.

In general and if necessary, a class may implement three kinds of methods for each association:

- Methods to add links: Usually referred to as *addRole*.
- Methods to remove links: Usually referred to as *removeRole*.
- Methods to get linked objects. Usually *getRole* would be implemented to return the whole set of linked objects. This set *must be protected* in the sense that it may be consulted and its elements may be iterated, but no element can be added or removed from the original set. Additionally, methods to get a specific element or subset given a key or other search criterion may be implemented if necessary.

Usually associations to one may be implemented as a single variable and not as a collection. In this case, the *addRole* and *removeRole* methods do not apply and the class implements instead a *replaceRole* method that replaces the current link with a new link.

Associations to 0..1 also require the implementation of the *replaceRole* method. But, in this case, *addRole* and *removeRole* may be implemented as well. If the *addRole* and *removeRole* methods are implemented for 0..1 roles, then they must check the role

bounds.

Derived associations must implement only the *get* method, in accordance with their definition.

Other methods would still be necessary depending on the kind of the association, such as:

- If the association is *qualified*, there may be an additional *get* that receives as an argument the qualifier key and returns the qualified object or subset. Additionally, if the qualifier is external, the *add* method should receive the value for the qualifier; if it is an internal qualifier that argument must not be passed because it may be obtained from the object itself. A new *remove* method also may be added to remove a link to an object based in the value of its qualifier.
- In the case of *ordered* associations, an additional *get* method may return an object based on its position in the collection. Also, the *add* method could add elements at a given position, which is indicated as a new parameter for the method, and the *remove* method may remove a link from a given position. Ordered associations may also have methods to access, add, and remove elements from their head or tail.
- *Stacks* and *queues* may have special methods such as *push* and *pop* that follow specific rules for those structures.

[Table e10.1](#) presents a summary of the methods that *may* be implemented for each kind of association in a class, depending on the design needs.

Table e10.1

Typical Operations over Associations Depending on their Type

Type	To 1	To 0..1	To Many (*)
Get	<i>getRole():Object</i>	<i>getRole():Object</i>	<i>getRole():Set</i>
Add	Does not apply	<i>addRole(obj)</i>	<i>addRole(obj)</i>
Remove	Does not apply	<i>removeRole()</i>	<i>removeRole(obj)</i>
Replace	<i>replaceRole(obj)</i>	<i>replaceRole(obj)</i>	<i>replaceRole(oldObj,newObj)</i>
Type	Ordered Set	Sequence	
Get	<i>getRole():OrderedSet</i> <i>getRole(position):Object</i> <i>getFirstRole():Object</i> <i>getLastRole():Object</i>	<i>getRole():Sequence</i> <i>getRole(position):Object</i> <i>getFirstRole():Object</i> <i>getLastRole():Object</i>	
Add	<i>addRole(position,obj)</i> <i>addFirstRole(obj)</i> <i>addLastRole(obj)</i>	<i>addRole(position,obj)</i> <i>addFirstRole(obj)</i> <i>addLastRole(obj)</i>	
Remove	<i>removeRole(obj)</i> <i>removeRole(position)</i> <i>removeFirstRole()</i> <i>removeLastRole()</i>	<i>removeRole(position)</i> <i>removeFirstRole()</i> <i>removeLastRole()</i>	
Replace	<i>replaceRole(oldObj,newObj)</i> <i>replaceRole(position,obj)</i> <i>replaceFirstRole(obj)</i> <i>replaceLastRole(obj)</i>	<i>replaceRole(position,obj)</i> <i>replaceFirstRole(obj)</i> <i>replaceLastRole(obj)</i>	
Type	Map (Internal Qualifier)	Map (External Qualifier)	
Get	<i>getRole():Set</i> <i>getRole(key):Object</i>	<i>getRole():Set</i> <i>getRole(key):Object</i>	
Add	<i>addRole(obj)</i>	<i>addRole(key,obj)</i>	
Remove	<i>removeRole(obj)</i> <i>removeRole(key)</i>	<i>removeRole(obj)</i> <i>removeRole(key)</i>	
Replace	<i>replaceRole(oldObj,newObj)</i>	<i>replaceRole(oldKey,newObj)</i> <i>replaceRole(oldObj,newObj)</i>	
Type	Partition (Internal Qualifier)	Partition (External Qualifier)	
Get	<i>getRole():Set</i> <i>getRole(key):Set</i>	<i>getRole():Set</i> <i>getRole(key):Set</i>	
Add	<i>addRole(obj)</i>	<i>addRole(key,obj)</i>	
Remove	<i>removeRole(obj)</i>	<i>removeRole(obj)</i>	
Replace	<i>replaceRole(oldObj,newObj)</i>	<i>replaceRole(oldObj,newObj)</i>	

(Continued)

Type	Set with Association Class	Bag	Array (Fixed Size)
Get	<i>getRole():Set</i> <i>getAssociationClass():Set</i> <i>getAssociationClass(obj):Object</i>	<i>getRole():Bag</i>	<i>getRole():Array</i> <i>getRole(position):Object</i>
Add	<i>addRole(obj)</i>	<i>addRole(obj)</i>	Does not apply
Remove	<i>removeRole(obj)</i> <i>removeAssociationClass(obj)</i>	<i>removeRole(obj)</i>	Does not apply
Replace	<i>replaceRole(oldObj,newObj)</i>	Does not apply	<i>replaceRole(position,obj)</i>
Type	Stack	Queue	
Get	<i>getRole():Object</i>	<i>getRole():Object</i>	
Add	<i>pushRole(obj)</i>	<i>queueRole(obj)</i>	
Remove	<i>popRole()</i>	<i>removeRole()</i>	
Replace	Usually does not apply	Usually does not apply	

Observe that in [Table e10.1](#), each kind of association has a different set of methods for accessing and changing links. The implementation of these methods follows definitions that are usually kept the same from class to class. The following subsections show examples of some of those methods.

10.3.1 Unidirectional association to one

The unidirectional association with role multiplicity 1 may be stored in a single instance variable in the origin class, and its type should be the destination class. [Figure e10.3](#) shows a unidirectional association *to one* from *Car* to *Person* with role name *owner*.



FIGURE E10.3 A class with unidirectional association to 1.

The implementation of the *Car* class requires an instance variable named *owner* implemented with type *Person*. Regarding the association methods, following [Table e10.1](#), only *getOwner* and *replaceOwner* should be implemented. The pseudocode for the *Car* class in this case could be

```

CLASS Car

  PRIVATE ASSOCIATION VAR owner:Person

  CONSTRUCTOR METHOD Create(anOwner:Person)
    owner:=anOwner
  END CONSTRUCTOR METHOD

  METHOD getOwner():Person
    RETURN owner
  
```

```

END METHOD
METHOD replaceOwner(newOwner:Person)
    owner:=newOwner
END METHOD
END CLASS

```

In the case of a role with multiplicity 0..1, there are two possibilities:

- It may be implemented as a set, that is, similar to the role with multiplicity *. In this case, if there is no object linked, the *get* method would return the empty set.
- It may be implemented as a single variable, that is, similar to the roles with multiplicity 1. In this case, as shown above, if there is no object linked, the *get* method would return the *null* object.

The second approach is widely adopted. There is even the possibility of using the *null object* design pattern (Woolf, 1998) instead of the language-provided null value. The advantage is that the null object is language independent. The null object is an instance of a *Null* class whose behavior consists of doing nothing. If there is iteration over a *null object* it produces nothing, just as with an empty set. On the other hand, iterating over a null or nil *value* would produce an exception, which would not be desired in that case.

If we assume that the role multiplicity in [Figure e10.3](#) is 0..1, there are two possible implementations. The first one considers the association as a set:

```

CLASS Car

    PRIVATE ASSOCIATION VAR owner:Set<Person>
    CONSTRUCTOR METHOD Create()
        owner:=Set.new()
    END CONSTRUCTOR METHOD
    METHOD getOwner():Set<Person>
        RETURN owner.protected()
    END METHOD
    METHOD addOwner(newOwner:Person)
        IF owner.size()>0 THEN
            Exception.throw('Car already has an owner')
        ENDIF
        owner.add(newOwner)
    END METHOD
    METHOD removeOwner()
        IF owner.size()==0 THEN
            Exception.throw('Car does not have an owner')
        ENDIF
        owner.removeOneElement()
    END METHOD

```

```

METHOD replaceOwner(newOwner:Person)
  IF owner.size()==1 THEN
    owner.removeOneElement()
  ENDIF
  owner.add(newOwner)
END METHOD

```

```

END CLASS

```

Every time a method such as *getOwner* returns a set of objects, the collection must be protected against change. That is explained in more detail later in this chapter.

The *removeOwner* method was implemented by calling the *removeOneElement* method over a set. That method removes one element (any one) from the set without the need to specify which one. As the set would have only one element at this point it would not be necessary to know which element it is in order to remove it from the set. The *removeOwner* method could also be implemented as *owner:=Set.new()*, with the same final result. However, as most programmers should notice, that implementation would produce a lot of garbage in memory if objects are often added and removed from that role, and this could degrade performance.

The second approach considers the association as a single variable, and uses the Null object design pattern:

```

CLASS Car

```

```

  PRIVATE ASSOCIATION VAR owner:Person
  CONSTRUCTOR METHOD Create()
    owner:=NullObject.instance()
  END CONSTRUCTOR METHOD
  METHOD getOwner():Person
    RETURN owner
  END METHOD
  METHOD addOwner(newOwner:Person)
    IF owner<>NullObject.instance() THEN
      Exception.throw('Car already has an owner')
    ENDIF
    owner:=newOwner
  END METHOD
  METHOD removeOwner()
    IF owner=NullObject.instance() THEN
      Exception.raise('Car does not have an owner')
    ENDIF
    owner:=NullObject.instance()
  END METHOD

```



```

METHOD replaceOwner(newOwner:Person)
    owner:=newOwner
END METHOD

```

```

END CLASS

```

Both implementations of *replaceOwner* shown above assume that if the owner exists it is replaced, and that if it does not exist it is defined. No exception is raised if *replaceOwner* is called for a car that does not have an owner. In that case, it behaves just like *addOwner*.

10.3.2 Unidirectional association to many

The unidirectional association *to many* may be implemented as a data structure. If it is a simple association with multiplicity * in the destination role, then it may be implemented as a set. Below is a pseudocode example for the association represented in [Figure e10.4](#):



FIGURE E10.4 A class with a simple unidirectional association to many.

```

CLASS Customer

```

```

PRIVATE ASSOCIATION VAR wishes:SET<Book>
METHOD getWishes():SET<Book>
    RETURN wishes.protected()
END METHOD
METHOD addWish(aBook:Book)
    wishes.add(aBook)
END METHOD
METHOD removeWish(aBook:Book)
    wishes.remove(aBook)
END METHOD

```

```

END CLASS

```

It is not necessary to implement the *replaceWish* method here, because wishes usually are only added and removed from a customer. It is not common to replace a wish with another. Thus, that method is not considered for implementation.

The *protected* message sent to a collection produces a noneditable version of that collection, to avoid elements being removed or added to the original collection by means other than the *add* and *remove* messages that are implemented in the class.

The implementation of that protection may be done in a number of ways; some of them are language specific. Examples include making a copy of the original collection (shallow copy or deep copy depending on the degree of protection intended), or using the *protection proxy* design pattern (Gamma, Helm, Johnson, & Vlissides, 1995), which suggests the implementation of a class that encapsulates the original collection, and implements only a method to iterate over the elements but not to modify the original collection.

If the association role is labeled with `{ordered}` or `{sequence}` the data type of the variable that represents the role must be replaced by the corresponding data type supported by the language, and additionally, other specific methods must be implemented, as mentioned in Table e10.1.

In the case of a role with identical lower and upper bounds, it may be implemented as an array. For example a multiplicity of 5 could be implemented as an array of 5 positions. The original *add* and *remove* commands are not applicable to the array structure because its size cannot be changed. However, elements may be replaced based on their position, as shown in Table e10.1.

Roles whose multiplicity is an interval, such as 3..8, may be implemented just like roles to many (*). The only difference is that the collection must have its bounds checked when objects are added or removed from it.

10.3.3 Unidirectional qualified association

The unidirectional qualified association is implemented in a manner similar to the association with multiplicity to many. However, instead of the data type *Set*, we use a mapping, or dictionary structure (*MAP*) that associates an alphanumeric, primitive, or enumeration type (key) to one object or a set of objects (values).

As for any other unidirectional associations, we must keep in mind that the unidirectional implementation is only possible when the association has no multiplicity bounds in its origin, that is, the origin role must have multiplicity *. If this is not the case, a bidirectional implementation must be considered.

Figure e10.5 shows the implementation of a qualified association defining a map to 0..1 with an internal qualifier. The customer wish list is now considered a qualified association. The respective code is presented below:

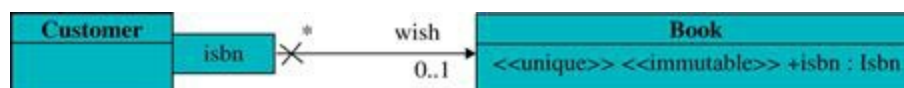


FIGURE E10.5 Qualified association with an internal qualifier.

```
CLASS Customer
```

```
PRIVATE ASSOCIATION VAR
```

```
wishes:MAP<ISBN,Book>
```

```

METHOD getWishes():SET<Book>
    RETURN wishes.getValues()
END METHOD

METHOD getWish(anIsbn:Isbn):Book
    RETURN wishes.atKey(anIsbn)
END METHOD

METHOD addWish(aBook:Book)
    wishes.add(aBook.getIsbn(),aBook)
END METHOD

METHOD removeWish(aBook:Book)
    wishes.removeValue(aBook)
END METHOD

METHOD removeWish(anIsbn:Isbn)
    wishes.removeKey(anIsbn)
END METHOD

METHOD replaceWish(oldBook,newBook:Book)
    -- not implemented
END METHOD

END CLASS

```

The *replace* method is not implemented for wishes again because it would not make sense to replace a book with another.

We note here that the basic *map* data structure has the usual operations for accessing a value given its key (*atKey*), accessing the set of all values (*getValues*), including a key/value pair (*add*), removing a pair given its key (*removeKey*) or given its value (*removeValue*), and so on.

It is important to stress that the design of the qualified association with an internal qualifier only works if the qualifier attribute is immutable. If this was not the case, the attribute could change and that change would not necessarily propagate to the value that is the key for the association. This would create an inconsistency. That constraint, however, does not apply to maps with external qualifiers, because in that case the qualifier is not an attribute of the qualified class.

[Figure e10.6](#) presents a map with an external qualifier. The corresponding code is shown below:

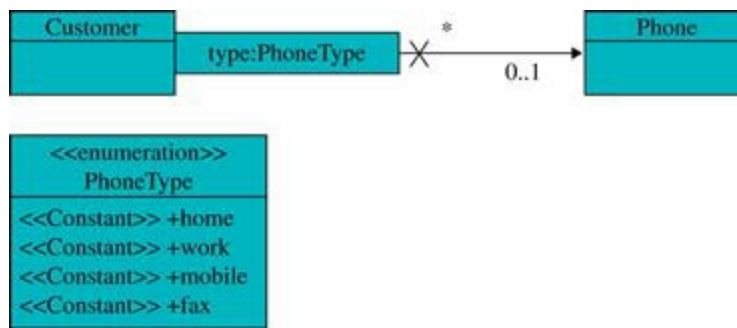


FIGURE E10.6 Qualified association with an external qualifier.

CLASS Customer

```

PRIVATE ASSOCIATION VAR phones:MAP<PhoneType, Phone>
METHOD getPhones():SET<Phone>
  RETURN phones.getValues()
END METHOD
METHOD getPhone(aType:PhoneType):Phone
  RETURN phones.atKey(aType)
END METHOD
METHOD addPhone(aType:PhoneType; aPhone:Phone)
  phones.add(aType, aPhone)
END METHOD
METHOD removePhone(aPhone:Phone)
  phones.removeValue(aPhone)
END METHOD
METHOD removePhone(aType:PhoneType)
  phones.removeKey(aType)
END METHOD
  
```

END CLASS

In the case of [Figure e10.6](#) there is an issue to be considered: as the origin of the association has no multiplicity restriction, and the qualifier is not an attribute of the class, nothing assures that the same phone is not associated to two or more keys. For example, the following sequence of commands would produce a phone associated to two different types:

```

aCustomer.addPhone("residential", aPhone)
aCustomer.addPhone("comercial", aPhone)
  
```

If that is what is expected, nothing else must be said. But if that is not what is meant, the role on the left side should be 1 and the association should be implemented as a bidirectional association, and control mechanisms should be implemented to avoid a phone from being associated to more than one type.

For the next example, we consider that different publishers could edit the same book. Thus we have books and book specifications: a *book* has one single publisher, but a *book specification* has a set of publishers. The ISBN of the *book* for each publisher may be different, but the *book specification* is the same for different publishers, and so is its genre. Figure e10.7 illustrates that a book specification has the genre as an attribute, and therefore each book specification has only one genre that may be associated to different publishers. Figure e10.7 presents a *partition*, that is, a qualified association *to many*, with an internal qualifier. The corresponding code is shown below:

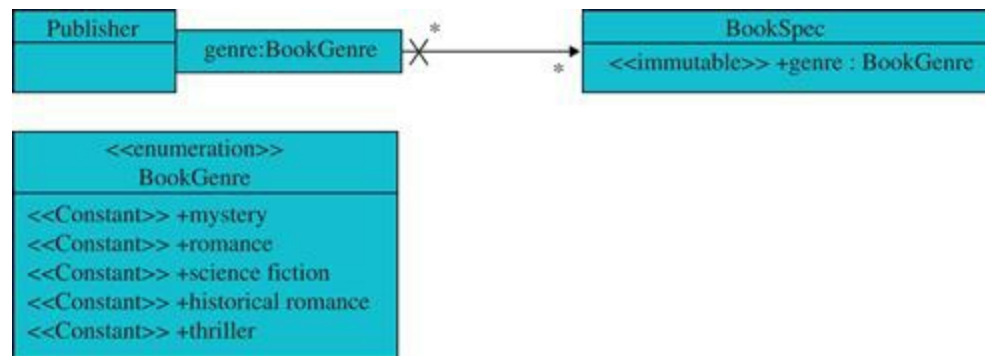


FIGURE E10.7 Qualified association as a partition with an internal qualifier.

CLASS Publisher

```
PRIVATE ASSOCIATION VAR bookSpecs:RELATION<BookGenre,BookSpec>
```

```
METHOD getBookSpecs():SET<BookSpec>
```

```
  RETURN bookSpecs.getValues()
```

```
END METHOD
```

```
METHOD getBookSpec(aGenre:BookGenre):SET<BookSpec>
```

```
  RETURN bookSpec.atKey(aGenre)
```

```
END METHOD
```

```
METHOD addBookSpec(aBookSpec:BookSpec)
```

```
  bookSpecs.add(aBookSpec.getGenre(),aBookSpec)
```

```
END METHOD
```

```
METHOD removeBookSpec(aBookSpec:BookSpec)
```

```
  bookSpecs.removeValue(aBookSpec)
```

```
END METHOD
```

END CLASS

The implementation above uses a data structure named *RELATION*, which does not exist in most programming languages. But it can be easily implemented. It has an interface similar to *MAP*, but it allows the same key to be associated to many values and not just one. In the example above, for each *BookGenre* the structure associates a set of instances of *BookSpec*.

There is an issue here too. The qualifier attribute *genre* in class *BookSpec* must be *immutable* because the qualifier is internal. If this was not the case, problems like the one aforementioned for the qualified map could jeopardize the design. The attribute is not *unique* because different book specifications may have the same genre.

Does the design presented in [Figure e10.7](#) allow a book specification to be linked to a publisher or different publishers with different genres? The answer is *no*. As the genre is immutable and the *addBookSpec* method takes the key from the *BookSpec* attribute *genre*, it is only possible to add a book specification once to the publisher. As the genre is immutable, the book specification should never be added again with a different genre.

This would not be the case if the genre were not immutable: the same book specification could change its genre and be linked again to one or another publisher. In that case, an inconsistent link to the book specification with the old genre would be left behind.

10.3.4 Unidirectional association with association class

When the association has an association class, it is necessary to implement the creation and destruction of instances of that class each time a corresponding link is added or removed.

Association classes may exist in associations with any multiplicity. However, they are more common and useful in associations that are many to many.

One possible implementation for this kind of association is to create a *map* associating instances of the opposite class to instances of the association class.

[Figure e10.8](#) shows an association class, and its implementation is shown below:

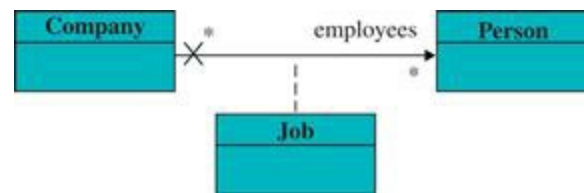


FIGURE E10.8 Unidirectional association with association class.

```
CLASS Company
```

```
PRIVATE ASSOCIATION VAR employees:MAP<Person,Job>
```

```
METHOD getEmployee():SET<Person>
```

```
RETURN employees.getKeys()
```

```
END METHOD
```

```
METHOD getJob():SET<Job>
```

```
RETURN employees.getValues()
```

```
END METHOD
```

```
METHOD getJob(aPerson:Person):Job
```

```

RETURN employees.atKey(aPerson)
END METHOD
METHOD addEmployee(aPerson:Person)
  employees.add(aPerson,Job.Create())
END METHOD
METHOD removeEmployee(aPerson:Person)
  LOCAL VAR aJob:Job
  aJob:=employees.atKey(aPerson)
  employees.removeKey(aPerson)
  aJob.destroy()
END METHOD
METHOD removeJob(aJob:Job)
  employees.removeValue(aJob)
  aJob.destroy()
END METHOD
METHOD replaceEmployee(oldEmployee,newEmployee:Person)
  self.removeEmployee(oldEmployee:Person)
  self.addEmployee(newEmployee:Person)
END METHOD
END CLASS

```

In [Figure e10.8](#) we see that when a new link is created from *Company* to *Person*, a new instance of *Job* is automatically created.

The operations that explicitly destroy a job when the link is removed are included in the code to make clear that this is what must be done, as the job cannot exist independently of the link that created it. In languages with a garbage collector and no explicit disposal of objects, it must be ensured that after the link is removed, no other references remain to the object.

The *replace* command only deletes the old link with its job and creates a new one. The instance of *Job* is not maintained when the role is replaced. However, in some cases that could be the meaning intended.

10.4 Bidirectional associations

As mentioned before, the unidirectional implementation of associations is only possible when they are navigable in one direction and there is no multiplicity constraint in the origin role. In other situations, the bidirectional implementation is necessary.

At least three patterns for implementing bidirectional associations have been proposed ([Fowler, 2003](#)):

- Implementing the association as two unidirectional associations (*mutual friends* pattern).
- Implementing the association as a unidirectional association in just one of the classes.

Navigation would be possible from the opposite direction by means of a query.

- Implementing an intermediary object that represents the association.

In all of the cases above, if the association is navigable in both directions the *get* method must be implemented in both participating classes, because navigation must be allowed in both directions. However, if we have the case of a unidirectional association with a multiplicity restriction at the origin, then the *get* method must be implemented only at the origin class.

The *add* and *remove* methods, if required, may be implemented only in one of the classes, because if they exist in both classes they would be redundant as they would do exactly the same thing.

10.4.1 Mutual friends

The option for implementing bidirectional associations in both directions is the most efficient in terms of time, but it is less efficient in terms of space allocation because each association is implemented twice. It may also require more control overhead, because the implementation of the association in both classes must be synchronized.

The implementation of the unidirectional components of the association follows the recommendations given in [Section 10.3](#). However, three subcases still have to be considered here:

- Both roles are optional.
- Only one role is mandatory.
- Both roles are mandatory.

10.4.1.1 Both roles optional

If both roles are optional and no other constraints are present, then links may be added and removed at will. As links must be added and removed from both sides of the association, auxiliary methods would be needed to add and remove the individual unidirectional links.

These auxiliary methods must not be called in any other place except the *add* and *remove* methods. That is why they must be declared as *private*, but the opposite class must have access to them; thus, they are exported exclusively to that class.

[Figure e10.9](#) presents a bidirectional many-to-many association. The corresponding implementation code is shown below:



FIGURE E10.9 Bidirectional association from many to many.

CLASS Customer


```
PRIVATE ASSOCIATION VAR wishes:SET<Book>
```

```
METHOD getWishes():SET<Book>
```

```
  RETURN wishes.protected()
```

```
END METHOD
```

```
METHOD addWish(aBook:Book)
```

```
  self.privateAddWish(aBook)
```

```
  aBook.privateAddWisher(self)
```

```
END METHOD
```

```
METHOD removeWish(aBook:Book)
```

```
  self.privateRemoveWish(aBook)
```

```
  aBook.privateRemoveWisher(self)
```

```
END METHOD
```

```
PRIVATE METHOD privateAddWish(aBook:Book)
```

```
EXPORTED TO: Book
```

```
  wishes.add(aBook)
```

```
END METHOD
```

```
PRIVATE METHOD privateRemoveWish(aBook:Book)
```

```
EXPORTED TO: Book
```

```
  wishes.remove(aBook)
```

```
END METHOD
```

```
END CLASS
```

```
CLASS Book
```

```
PRIVATE ASSOCIATION VAR wishers:SET<Customer>
```

```
METHOD getWishers():SET<Customer>
```

```
  RETURN wishers.protected()
```

```
END METHOD
```

```
PRIVATE METHOD privateAddWisher(aCustomer:Customer)
```

```
EXPORTED TO: Customer
```

```
  wishers.add(aCustomer)
```

```
END METHOD
```

```
PRIVATE METHOD privateRemoveCustomer(aCustomer:Customer)
```

```
EXPORTED TO: Customer
```

```
  wishers.remove(aCustomer)
```

```
END METHOD
```

```
END CLASS
```

The auxiliary methods in the code above are declared as private but exported to the other participating class.

As mentioned before, the access (*get*) methods must be implemented in both classes if

the association is navigable in both directions. But the *add* and *remove* methods may be implemented in just one class. In the example above, they are implemented only in *Customer* class.

10.4.1.2 Only one mandatory role

If one of the roles is mandatory, as for example 1 to *, then the team must ask if the mandatory role is immutable or not. For example, a payment has one order and cannot change to another order (it is immutable), but a car has an owner and can change to another owner if it is sold.

Let us first examine the case of an association link that can be updated, as shown in [Figure e10.10](#).



FIGURE E10.10 A bidirectional association with one mandatory role that can be updated.

The code for implementing the classes in [Figure e10.10](#) might be like the following:

```
CLASS Car

PRIVATE ASSOCIATION VAR owner:Person
CONSTRUCTOR METHOD Create(anOwner:Person)
    owner:=anOwner
    anOwner.privateAddCar(self)
END CONSTRUCTOR METHOD
METHOD getOwner():Person
    RETURN owner
END METHOD
METHOD replaceOwner(anOwner:Person)
    owner.privateRemoveCar(self)
    anOwner.privateAddCar(self)
    owner:=anOwner
END METHOD

END CLASS

CLASS Person

PRIVATE ASSOCIATION VAR cars:SET<Car>
METHOD getCars():SET<Car>
    RETURN cars.protected()
END METHOD
```

```

PRIVATE METHOD privateAddCar(aCar:Car)
EXPORTED TO: Car
cars.add(aCar)
END METHOD
PRIVATE METHOD privateRemoveCar(aCar:Car)
EXPORTED TO: Car
cars.remove(aCar)
END METHOD
END CLASS

```

As we can see, the *Car* class implements the only public updating method, which is *replaceOwner*. An ownership link is created every time a new instance of *Car* is created, as defined in the car's constructor. Then, the ownership may only be changed from one person to another. The *Person* class only implements the private methods to add and remove a car from its own local collection.

When a car owner is replaced, it is done in three steps: first the car is removed from the old owner, then it is added to the new owner, and finally the owner is updated to be the new owner.

The second case to be considered here is when the role is immutable. This is the case of a 1 to * association from *Customer* to *Order*, for example: it is mandatory for *Order* and an order can never change its customer. This situation is shown in [Figure e10.11](#), and the corresponding code follows:

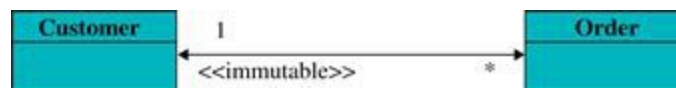


FIGURE E10.11 A bidirectional association with one mandatory immutable role.

```

CLASS Order

```

```

PRIVATE ASSOCIATION VAR customer:Customer
CONSTRUCTOR METHOD create(aCustomer:Customer)
customer:=aCustomer
aCustomer.privateAddOrder(self)
END CONSTRUCTOR METHOD
METHOD getCustomer():Customer
RETURN customer
END METHOD

```

```

END CLASS

```

```

CLASS Customer

```

```

PRIVATE ASSOCIATION VAR orders:SET<Order>
METHOD getOrders():SET<Order>
  RETURN orders.protected()
END METHOD
METHOD removeOrder(anOrder:Order)
  orders.remove(anOrder)
  anOrder.destroy()
END METHOD
PRIVATE METHOD privateAddOrder(anOrder:Order)
EXPORTED TO: Order
  orders.add(anOrder)
END METHOD

```

```

END CLASS

```

In this case, an order is immediately associated to a customer when it is created. If an order is removed from a customer then it must be destroyed.

10.4.1.3 Two mandatory roles

There are situations when the two roles of an association are mandatory. In this case, no object can exist without being linked to another. This means that the creation of the links must happen in the constructor of both classes. [Figure e10.12](#) shows an example of a 1 to 1..* association; it is mandatory on both ends.

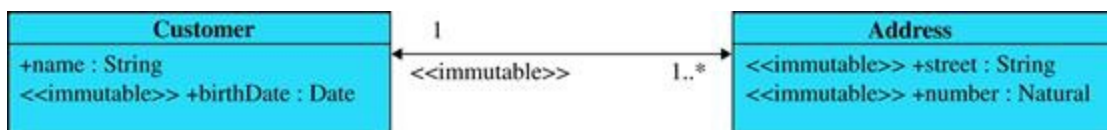


FIGURE E10.12 A bidirectional association that is mandatory on both ends.

Once more, the methods that will be implemented depend on deciding if the roles are immutable or not. The customer role from the point of view of an address is immutable. This means that an address cannot change from one customer to another.³

When a customer is created it must be created with at least one address. This means that the creator of a customer should receive all the data needed to instantiate that address (only *street* and *number* in the example, for simplification).

An existing customer may add a new address, but the method that adds an address cannot simply receive an instance of address as an argument because no address may exist without being linked to a customer. Thus, the customer should implement a method to add a new address that receives the data necessary to instantiate such an address. The code for implementing this situation is shown below:

```

CLASS Address

```

```

PRIVATE ASSOCIATION VAR customer:Customer
PRIVATE ATTRIBUTE VAR street:String
PRIVATE ATTRIBUTE VAR number:Natural
CONSTRUCTOR METHOD Create(aCustomer:Customer;
aStreet:String; aNumber:Natural)
customer:=aCustomer
street:=aStreet
number:=aNumber
END CONSTRUCTOR METHOD
METHOD getCustomer():Customer
RETURN customer
END METHOD

```

END CLASS

CLASS Customer

```

PRIVATE ASSOCIATION VAR addresses:SET<Address>
PRIVATE ATTRIBUTE VAR name:String
PRIVATE ATTRIBUTE VAR birthDate:Date
CONSTRUCTOR METHOD create(aName:String; aBirthDate:Date;
aStreet:String; aNumber:Natural)
name:=aName
birthDate:=aBirthDate
addresses:=Set.new()
addresses.add(Address.Create(self,aStreet,aNumber))
END CONSTRUCTOR METHOD
METHOD getAddresses():SET<Address>
RETURN addresses.protected()
END METHOD
METHOD addAddress(aStreet:Street; aNumber:Natural)
addresses.add(Address.create(self,aStreet,aNumber))
END METHOD
METHOD removeAddress(anAddress)
IF address.size()>1 THEN
addresses.remove(anAddress)
anAddress.destroy()
ELSE
Exception.throw('Customer must have at least one address')

```

```
ENDIF
```

```
END METHOD
```

```
METHOD getName():String
```

```
RETURN name
```

```
END METHOD
```

```
METHOD getBirthDate():Date
```

```
RETURN birthDate
```

```
END METHOD
```

```
METHOD setName(aName:String)
```

```
name:=aName
```

```
END METHOD
```

```
END CLASS
```

If the role from *Address* to *Customer* was not immutable, then a *replaceCustomer* command could be implemented as well in the *Address* class. But that is not the case.

10.4.2 Unidirectional implementation

Even if the association is bidirectional, it may be the case that navigation occurs much more often or is more critical in just one direction. If that happens, an option is to implement the association physically in *only one direction*, and implement the *get* method for the opposite side as a search query. The advantage is that the code is simpler, faster in one direction, and space saving. The disadvantage is that the navigation from the opposite direction would be much slower.

This form of implementation is also only possible when the role on the origin of the implemented direction has no multiplicity restriction. If there is a restriction, then the unidirectional implementation is possible only if additional mechanisms to control multiplicity at the origin of the association are implemented, with loss of performance. Below there is an example of unidirectional implementation for the association of [Figure e10.10](#), where the association is physically implemented only in the *Car* class:

```
CLASS Car
```

```
PRIVATE ASSOCIATION VAR owner:Person
```

```
CONSTRUCTOR METHOD Create(anOwner:Person)
```

```
owner:=anOwner
```

```
END CONSTRUCTOR METHOD
```

```
METHOD getOwner():Person
```

```
RETURN owner
```

```
END METHOD
```

```
METHOD replaceOwner(newOwner:Person)
```

```
owner:=newOwner
```

```
END METHOD
```

```

END CLASS

CLASS Person
METHOD getCars():SET<Car>
  LOCAL VAR cars:SET<Car>
  cars:=Set.new()
  FOR EACH car IN Car.getAllInstances() DO
    IF car.getOwner()==self THEN
      cars.add(car)
    END IF
  END FOR
  RETURN cars
END METHOD
END CLASS

```

The *Car* class above is implemented just like if the association was unidirectional from it to *Person*. The *Person* class implements only the *getCars* method through a search query on the set of all instances of *Car*. This implementation only works well if the role multiplicity at the origin has is unconstrained.

Regarding the time complexity of the *get* methods, the bidirectional implementation runs in *constant time*⁴ in both directions, and the unidirectional implementation has constant time for *getOwner* and *linear time*⁵ for *getCars*, that is, the performance of the *getCars* method depends on the number of cars registered. This design may be optimized by using *hash* techniques to index the set of customers relative to their orders. This way, the complexity of the *getCars* query may almost become constant in practice with the cost of some extra memory space.

A limitation of this technique is that the programming language must provide a method to access all instances of a class. Otherwise, the programmer must supply such a mechanism. Unfortunately, this is a hazardous design that may cause trouble due to the global visibility of those instances (Cardoso, 2011).

10.4.3 Implementation with an intermediary object

A bidirectional association may also be implemented by means of an intermediary object that represents the association. The intermediary object consists of a table with pairs of linked instances. Possible implementation for the bidirectional association from many to one of [Figure e10.11](#) with an intermediary object follows:

```

GLOBAL VAR orderXcustomer:MAP<Order, Customer>

VISIBILITY RESTRICTED TO: Order, Customer

CLASS Customer

  METHOD getOrders():Set<Order>
    RETURN orderXcustomer.getKeysFor(self)

```

```
END METHOD
```

```
METHOD addOrder(anOrder:Order)
```

```
    orderXcustomer.add(anOrder,self)
```

```
END METHOD
```

```
METHOD removeOrder(anOrder:Order)
```

```
    orderXcustomer.remove(anOrder,self)
```

```
END METHOD
```

```
END CLASS
```

```
CLASS Order
```

```
    CONSTRUCTOR METHOD Create(aCustomer:Customer)
```

```
        orderXCustomer.add(self,aCustomer)
```

```
END CONSTRUCTOR METHOD
```

```
METHOD getCustomer():Customer
```

```
    RETURN orderXcustomer.atKey(self)
```

```
END METHOD
```

```
END CLASS
```

If the programming language allows it, the association should be declared a global variable that is visible only by the participating classes. Unfortunately, most commercial languages would not allow such feature and the association would be implemented as a global variable with no visibility restriction.

If the association was many to many instead of one to many, then the *MAP* data type used above should be replaced by *RELATION* with small adjustments in the getters and setters.

The intermediary object approach tends to be much simpler and maintainable than the former ones. It also mimics the structure of a relational database because associations, as seen in [Chapter 13](#), are implemented as intermediary tables, which correspond to the intermediary object here.

The disadvantage of this method is that getters are slower in both directions when compared to the bidirectional approach and the global visibility of the intermediary object may cause design hazards if not used carefully.

10.5 Delegated methods and system operations

To this point, we have shown how to generate code for classes, attributes, associations, and their corresponding basic methods that may be considered part of their basic structure. Now it is the time to explain how to implement delegate methods and system operations. These should be implemented by following the dynamic models explained in

Chapter 9.

System operations are implemented as a sequence of messages labeled 1, 2, 3, ..., n that are sent by the controller. A delegate method labeled with x in the interaction diagram should be implemented by a sequence of messages labeled $x.1, x.2, x.3, x.n$ that are sent by the object that receives the message labeled with x .

Let us look again at [Figure 9.43](#), reproduced here as [Figure e10.13](#).

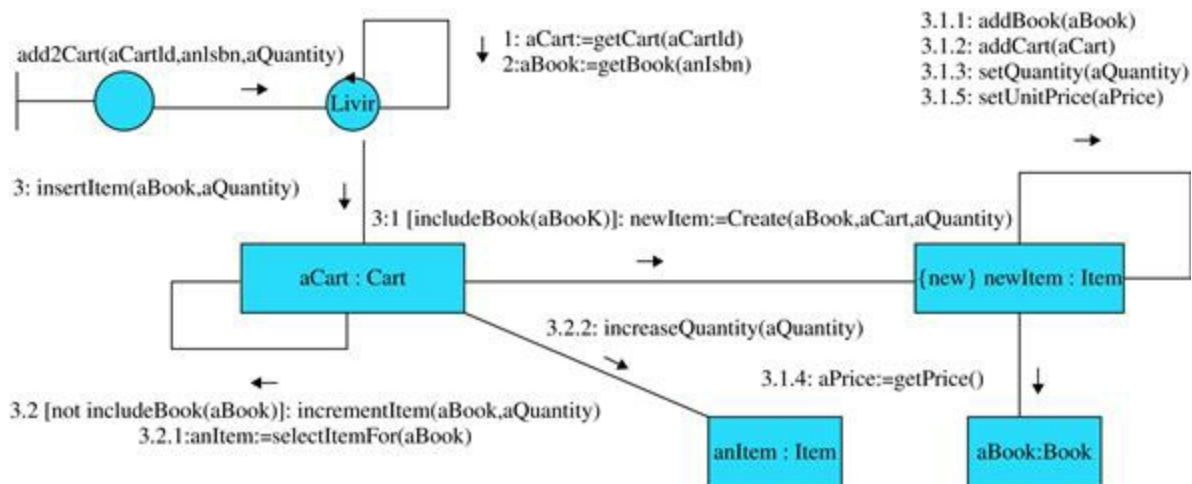


FIGURE E10.13 Reference communication diagram.

The *add2Cart* system command should therefore be implemented as a sequence of messages labeled with 1, 2, and 3. The corresponding pseudocode could be something like the following:

```
CLASS Livir
```

```
ASSOCIATION VAR carts:MAP<CartId, Cart>
```

```
ASSOCIATION VAR books:MAP<Isbn, Book>
```

```
METHOD add2Cart(aCartId:CartId; anIsbn:Isbn;  
aQuantity:Natural)
```

```
LOCAL VAR aCart:Cart
```

```
LOCAL VAR aBook:Book
```

```
aCart:=carts.at(aCartId) -- message 1
```

```
aBook:=books.at(anIsbn) -- message 2
```

```
aCart.insertItem(aBook,aQuantity) -- message 3
```

```
END METHOD
```

```
...
```

```
END CLASS
```

The implementation of the *insertItem* delegate method in the *Cart* class consists of the sequence of messages 3.1 and 3.2, because *insertItem* is labeled with 3. These messages are conditional and mutually exclusive. Thus, their implementation corresponds to an *if-then-else-endif* structure, as shown below:

```
CLASS Cart
```

```
...
```

```
METHOD insertItem(aBook:Book,aQuantity:Natural)
  LOCAL VAR newItem:Item
  IF self.includeBook(aBook) THEN
    newItem:=Item.Create(aBook,self,aQuantity) --3.1
  ELSE
    self.incrementItem(aBook,aQuantity) --3.2
  ENDIF
END METHOD
```

```
END CLASS
```

Notice that the variable name *aCart* is not known in this method and must be replaced here by *self*, which stands for the instance of *Cart* that is performing the method.

The *incrementItem* delegate method in the *Cart* class is labeled with 3.2. Therefore, its implementation is the sequence of messages 3.2.1 and 3.2.2:

```
CLASS Cart
```

```
...
```

```
METHOD incrementItem(aBook:Book; aQuantity:Natural)
  LOCAL VAR anItem:Item
  anItem:=self.selectItemFor(aBook) --3.2.1
  anItem.increaseQuantity(aQuantity) --3.2.2
END METHOD
```

```
END CLASS
```

The constructor of the *Item* class is complex and as it is labeled with 3.1, its implementation consists of the sequence of messages 3.1.1 to 3.1.5. The *Item* class would look like this:

```
CLASS Item
```

```
CONSTRUCTOR METHOD Create(aBook:Book; aCart:Cart;aQuantity:Natural)
  LOCAL VAR aPrice:Money
  self.addBook(aBook) --3.1.1
  self.addCart(aCart) --3.1.2
  self.setQuantity(aQuantity) --3.1.3
  aPrice:=aBook.getPrice() --3.1.4
  self.setPrice(aPrice) --3.1.5
END CONSTRUCTOR METHOD
```

```
END CLASS
```

The other methods that appear in [Figure e10.13](#) are basic ones and must be implemented following their default definition.

10.6 Patterns for filtered queries

In the examples shown to this point, basically two kinds of *get* queries were implemented: those that return all objects linked by a role, and those that return a single object given its identification, such as a qualifier or position.

However, sometimes it will be necessary to obtain a subset of objects in a given role. That subset is usually obtained by a filter such as “customers older than 30 years old,” “orders between 100 and 200 dollars,” “orders above 100 dollars issued last week,” etc.

There are at least three patterns to deal with this diversity of queries ([Fowler, 2003](#)):

- Implement a *single query* that returns all objects, and let the object that needs the information apply a filter to that collection.
- Implement *specific queries*, each one with a different filter, so that the object that needs the information calls a specific method in each case.
- Implement a generic query that uses a *filter object*.

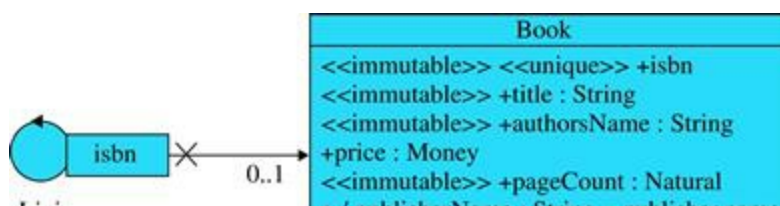
The single query and filter object approaches imply the implementation of a single method that leaves the class that has the responsibility simpler. However, it requires more code to be written in the classes that need the information.

The specific queries approach has as a consequence a greater number of methods in the class that holds the responsibility. But the classes that need the information would make simple calls and receive the information without any need for postfiltering.

The choice of one pattern or another must be made by the designer depending on the number of possible filters and potential calls. If there are only a few possibilities for filtering and many calls, the best choice is specific queries. If the quantity of filters is high and there are only a few calls for each filter, then the single query or filter object approach would be better. Single query is more straightforward, but produces more work during the coding phase: each time a filter is used, the respective code must be implemented in the class that calls the query. Filter object requires defining a class that implements the *filter object*, but after that investment is done, filter object tends to be simpler than single query.

A *filter object* is a parameter that is passed to the general query method. The filter object must contain attributes and associations to other objects. Those attributes and associations are used by the query to decide which objects must be returned.

For example, let us revisit the definition of the *Book* class shown in [Figure e10.14](#).



```

+publisherName : String = publisherName
<<optional>> +coverImage : Image
+quantityInStock : Natural = 0

```

FIGURE E10.14 Reference class for queries.

Suppose now that we need to query books filtered by *title*, *authorsName*, or *price*.

Applying the single query approach, a *getBooks()* method should be implemented in the *Livir* class, which is responsible for returning the set of all books. If another object needs to filter that set, it must apply the filter to the resulting set.

If the specific queries approach is used, the *Livir* class should implement three queries that may be considered variants of the base query:

- *getBooksByTitle(aTitle:String)*.
- *getBooksByAuthorsName(anAuthorName:String)*.
- *getBooksByPrice(aRange:Range<Money>)*.

Each of these queries return only the elements that satisfy the respective filters indicated as arguments. If a combination of criteria is needed, then new queries usually should be created for the combined criteria.

The filter object approach requires the definition of a class to represent the filter object. For the example, a *BookFilter* class could be defined as shown in [Figure e10.15](#).

```

<<filter>>
BookFilter
<<optional>> +title : String
<<optional>> +authorsName : String
<<optional>> +priceRange : Range<Money>

```

FIGURE E10.15 A class for a filter object.

In this case, the *Livir* class should implement a single *getBooks(aBookFilter:BookFilter):Set<Book>* method for querying the set of all books.

Using this method, a query to return the books written by “Douglas N. Adams” that cost less than 10 dollars could be written like this:

```

...
myFilter:=BookFilter.Create()
myFilter.setAuthorsName("Douglas N. Adams")
myFilter.setPriceRange(Range(US$0,US$10))
cheapAdamsBooks:=self.getBooks(myFilter)
...

```

As a result, the variable *cheapAdamsBooks* contains the set of all instances of *Book* whose author is “Douglas N. Adams” and that cost less than 10 dollars. As the attribute *title* of *BookFilter* was not filled for the query, it is ignored, and any title would qualify for the filter.

10.7 The process so far

	Inception	Elaboration
Business Modeling	<p>Build a general view of the system:</p> <ul style="list-style-type: none"> • Build a business use case diagram and determine the automation scope for the project. • Build preliminary activity diagrams for business use cases. • Build preliminary state machine diagrams for key business objects. 	
Requirements	<p>Prepare the system use case diagram (functional requirements):</p> <ul style="list-style-type: none"> • Identify the system actors from the business use case model. • Identify the system use cases from the business use case model, and the activity and state machine diagrams from business modeling. <p>Identify nonfunctional requirements as use case annotations:</p> <ul style="list-style-type: none"> • Identify the main business rules associated to use cases. • Identify the main quality issues associated to use cases. <p>Identify supplementary requirements.</p>	<p>Detail requirements by expanding use cases:</p> <ul style="list-style-type: none"> • Identify the main flow. • Identify the alternate flows: variants and exception handlers.
Analysis and Design	<p>Prepare the preliminary conceptual model by observing system use cases and the concepts needed by them.</p>	<p>Elaborate the system sequence diagrams:</p> <ul style="list-style-type: none"> • Represent the main flow of a use case as a system sequence diagram. • Represent the system commands and queries using stateful or stateless strategies. • Complete system sequence diagrams with alternate flows. <p>Refine the conceptual model:</p> <ul style="list-style-type: none"> • Identify concepts, attributes, and associations in the text of expanded use cases. • Detail attributes and associations with stereotypes, multiplicity, and constraints as needed. • Organize the model by using inheritance, association classes, and temporal specifications.

(Continued)

	Inception	Elaboration
Implementation		<ul style="list-style-type: none"> • Add invariants as needed. • Improve the conceptual model with the application of analysis patterns. <p>Write system operation contracts for commands and queries in system sequence diagrams:</p> <ul style="list-style-type: none"> • Identify preconditions and exceptions based on invalid parameters and complementary constraints. • Identify postconditions for commands and returns for queries. <p>Design the domain tier:</p> <ul style="list-style-type: none"> • Create a sequence or communication diagram for each system command contract. • Use those diagrams to decide which methods to implement in each class. • Inspect those diagrams to discover which associations can be unidirectional and define them as such. • Look at system query contracts and decide which delegate queries could be implemented; some of them could be derived attributes or derived associations. • Produce or refine the design class diagram. <p>Generate code:</p> <ul style="list-style-type: none"> • Generate code for classes. • Generate code for attributes. • Generate code for associations. • Generate code for necessary basic methods such as get, set, add, remove, create, destroy, and others. • Generate code for system operations and delegate methods using dynamic models as reference.

(Continued)

	Inception	Elaboration
Test Project Management	<p>Estimate total effort, ideal calendar time, and average team size for the project.</p> <p>Estimate the duration and quantity of iterations for each phase.</p> <p>Prepare the phase plan and iteration plan for the first iteration.</p>	

10.8 Questions

1. Explain in detail how a delegate message that appears in a communication diagram must be implemented.
2. What are the basic operations that must be implemented for most kinds of associations? Which ones must be implemented for specific kinds of associations?
3. Propose a generic implementation for the query `getBooks(aBookFilter:BookFilter):Set<Book>`. Try to keep it as general and reusable as possible.
4. Propose an implementation for the system commands and delegate methods presented in [Figures 9.42](#) and [9.44](#).

¹In some cases, classes may allow more than one constructor, though.

²This method involves associations, which are explained in [Sections 10.3](#) and [10.4](#).

³In fact, in real life an address could change from one customer to another if we consider that a customer could move to the house of another customer whose address is already registered. However, usually systems do not treat addresses in such a precise way: pragmatically, if a customer moves, it is always a new address that is registered.

⁴Time remains the same even if the number of cars increase.

⁵Time increases as the number of customers increase. Linear time means that time is a function $t(x)=ax+b$, where x is the size of the set for customers and a and b are constants.

CHAPTER 11

Testing

This chapter explains how object-oriented software developed with the techniques shown in previous chapters can be tested. Specifically, three levels of testing are presented: unit tests for methods and classes, system operations tests, and use case tests. The chapter focuses on the functional approach to tests based on the operation contracts. For each parameter of a system or basic operation, the designer should define if it has acceptable and/or unacceptable values. A success test must be created for each success scenario and a failure test for each set of unacceptable values. With this approach test cases may be systematically designed. Equivalence partition and limit value analysis are also presented in order to determine the best choices for the test cases.

Keywords

Unit test; system operations test; system test; test-driven development; functional test

Key Topics in this Chapter

- Functional testing
- Stubs and drivers
- Test-driven development
- Unit testing
- System operations testing
- Use case testing

11.1 Introduction to testing

No matter how sophisticated the modeling and specification techniques used to develop software are, no matter how disciplined and competent the team is, there is a factor that makes software testing always necessary: *human error*. It is a myth to think that good developers working with state-of-the-art tools are capable of developing error-free software ([Beizer, 1990](#)).

Murphy's Law ([Bloch, 1980](#)) in many of its corollaries seems to speak directly to the software industry. For example:

- Anything that can go wrong will go wrong.
- If everything seems all right, you have not checked appropriately.
- Nature always sides with the hidden flaw.

For many years, software-testing activities were considered a punishment to programmers. Testing was considered as a waste of time because software was supposed to be correct from the beginning.

However, things have changed. The test discipline now is considered extremely important. Today it is an integral part of the software development process and one of

the disciplines of the Unified Process.

Furthermore, leading software development companies started to outsource software testing, by hiring *test factories*. That means that not only developers, but teams are specially trained to conduct testing.

This chapter presents test activities that are strongly adapted for use with the techniques shown in previous chapters.

Unit tests are used basically to check the behavior of classes, including their basic and delegate methods. If automatic code generation is used, such tests may be suppressed, because it is assumed that automatic code generators do not make human errors.

System operation tests are top-level functional tests that may be based on the examination of the system operation contract: its parameters, preconditions, and exceptions determine sets of valid and invalid parameters that must be systematically checked, and the postconditions determine the results that must be accomplished when valid entries are tested.

Finally, *system tests* are performed as systematic *use case tests*. Each use case is a script that establishes normal and alternative flows for accomplishing business goals. Use case tests may be executed manually or may be automated. If the client executes use case tests, they are called *acceptance tests*.

There are two major categories of test techniques:

- *Structural tests*, which evaluate the internal structure of the code.
- *Functional tests*, which evaluate operations based only on their inputs and outputs.

In this book, only functional techniques applied specifically to the three levels of testing are introduced:

- Functional unit test for basic and delegate operations.
- Functional system operation tests based on contracts.
- System and acceptance tests based on system use cases.

Readers interested in understanding automated testing in more detail should take a look at the book of [Meszaros \(2007\)](#), which presents a significant set of *xUnit* style test patterns, a family of frameworks for innumerable languages based on the design originally created by Kent [Beck \(1989\)](#).

11.2 Functional testing

Functional testing consists of a sequence of tests that define entry values for an operation and observe if the result is what was expected. Functional tests may be run without any knowledge of the programming code that implements the operation; only its behavior is observed. The quantity of tests to be conducted in order to assure that an operation is correct may be virtually infinite. Functional testing may use techniques to reduce the number of necessary tests without losing coverage. The most useful techniques for accomplishing that goal are *equivalence partitioning* and *limit value analysis*, which are explained in the following subsections.

11.2.1 Equivalence partitioning

One of the principles of functional testing is the identification of equivalent situations. For example, if an operation accepts a set of data (normality) and rejects another set (exception) then it may be said that there are two *equivalence sets*¹ of input data for that operation: *accepted* and *rejected values*. It is usually impossible to test every value included in those sets, because they may be virtually infinite. Thus, the technique of *equivalence partitioning* indicates that at least one element in each equivalence set must be tested (Burnstein, 2003).

Classically, the equivalence partitioning technique considers the division of the inputs through the following criteria (Myers, Sandler, Badgett, & Thomas, 2004):

- If the valid values are specified as an *interval* (for example, from 10 to 20), then we define one valid set (10 to 20) and two invalid sets (less than 10, and greater than 20).
- If the valid values are specified as a *quantity of values* (for example, a list with five elements) then we define a valid set (list with five elements) and two invalid sets (lists with less than five elements, and lists with more than five elements).
- If the valid entries are specified as a *set of acceptable values* that may be processed in different forms (for example, the strings of an enumeration such as “male” and “female”), then we define a valid set for each of the valid options and an invalid set for any other value.
- If the valid values are specified by a logical condition (for example “final date must be greater than initial date”) then we must define a valid set (when the condition is true) and an invalid set (when the condition is false).

The sets of valid inputs may be defined not only in terms of restrictions on the input data but also in terms of the results that may be produced. If the operation that is being tested has different behavior depending on the value of the input, then different valid sets must be defined for each behavior. For example, consider a simple operation $half(x:Integer):Integer$. This operation accepts that x may be any positive number. Consider also that the operation is defined so that it cannot accept zero or negative numbers. Finally, consider that the operation produces $(x-1)/2$ for odd numbers and $x/2$ for even numbers. Thus, we must consider that $half$ has three equivalence sets for the parameter x :

- A valid set composed of odd positive integers: 1, 3, 5,
- A valid set composed of even positive integers: 2, 4, 6,
- An invalid set composed of nonpositive integers: 0, -1, -2, -3,

The equivalence set values must be always restricted to the domain defined by the type of the parameter. For example, if the type of the parameter were something other than *Integer*, then different equivalence sets would be defined. If the operation was $half(x:Natural):Integer$, then the equivalence sets should be redefined so that they remain inside the *Natural* domain, which does not include zero and negative numbers:

- A valid set composed of odd positive integers: 1, 3, 5,
- A valid set composed of even positive integers: 2, 4, 6,
- No invalid sets.

On the other hand, assume the operation is defined with a broader parameter type such as $half(x:Real):Integer$; then the equivalence sets would be:

- A valid set composed of odd positive integers: 1, 3, 5,
- A valid set composed of even positive integers: 2, 4, 6,
- An invalid set composed of nonnatural numbers: $\mathbb{R} - \mathbb{N}$.

We can see from the examples above that if the designer restricts the type of the parameter as much as possible, then fewer or simpler invalid equivalence sets are defined. That is why examples in previous chapters use the *Natural* type instead of *Integer* whenever zero or negative numbers could not be accepted. And that is why enumerations should be used rather than unrestricted strings whenever possible.

11.2.2 Limit value analysis

People that work in software testing used to say that (software) bugs hide in the slits. Due to this, the equivalence partitioning technique is usually used in conjunction with another technique known as *limit value analysis*.

Limit value analysis consists of not choosing any random value from an equivalence set, but choosing two or more limit values if they can be determined.

In ordered domains, such as numbers, this criterion may be applied. For example, if an operation requires an integer parameter that is valid only if included in the interval $[10..20]$, then there are three equivalence sets:

- Invalid for any $x < 10$.
- Valid for $x \geq 10$ and $x \leq 20$.
- Invalid for any $x > 20$.

Limit value analysis suggests that eventual errors in the logic of a program are not at arbitrary points inside those intervals but at the boundary points where two intervals meet. Thus if the domain of the parameter is *Integer*:

- For the first invalid class, the value 9 must be tested.
- For the valid class, 10 and 20 must be tested.
- For the second invalid class, the value 21 must be tested.

Thus, if there is a logic error in the operation for some of these inputs it is much more probable that it will be found in this way rather than if a random value is chosen inside each interval that defines an equivalence set.

11.3 Stubs and drivers

Frequently, parts of the software must be tested separately from the main body of code, but at the same time they must communicate with other parts.

When a component *A* that is going to be tested calls operations from a component *B* that has not yet been implemented, component *B* may be replaced by a simplified version of it that implements only the behavior that is absolutely necessary to perform the test of component *A*. This simplified implementation that is used in the place of a component that has not yet been implemented is called a *stub*.

For example, suppose that a class *A* needs a prime number generator *B* that has not yet been implemented. The *n*th prime number would be generated by a function

prime(n:Natural):Natural. A simplified version of *B* may be implemented just for allowing class *A* to be tested. Suppose that the first five prime numbers are enough to adequately test class *A*. Thus, a stub for *B* could be implemented like this:

```
CLASS B
```

```
STUB METHOD prime(n:Natural):Natural
CASE n OF
  1: RETURN 2
  2: RETURN 3
  3: RETURN 5
  4: RETURN 7
  5: RETURN 11
OTHERWISE Exception.throw('Stub implementation
requires argument less or equal to 5')
END CASE
END STUB METHOD
```

```
END CLASS
```

In this way, class *A* may be tested without investing time in implementing a more complex function in class *B*.

On the other hand, operations implemented in class *A* will be called by other components of the system. But when class *A* is being tested, these components may not yet exist. Furthermore, if we are testing class *A* we usually desire that that test occurs in a systematic and reproducible way. In order to assure this, a *test driver* for class *A* may be implemented. The *driver* is a component of code (possibly a new class) that aims uniquely to systematically test a component.

For class *A* mentioned above, a driver could be implemented as a new class named *Driver4A*, for example. An example of a driver is shown in [Section 11.5](#).

Thus, stubs and drivers are simple implementations that simulate the behavior of other components. The stub is used in the place of a component that should be called but is not yet implemented. The driver is used in the place of a component that should call the component to be tested in a systematic and reproducible way.

Stubs are usually *throwaway code*. When the real component is implemented the stub is not necessary anymore and may be discarded.

On the other hand, drivers are important pieces of code that must be kept forever. Every time a component must be tested – and it happens frequently during software development and evolution – the driver would be available for automatically performing the test.

11.4 Test-driven development

Test-driven development or TDD (Beck, 2003) is a technique and a programming philosophy that incorporates automatic testing to the process of producing code. It works like this:

1. First, the programmer that receives the specification for a new functionality that must be implemented should create a set of automatic tests for the code that does not yet exist.
2. This set of tests must be executed and observed to fail. This is done to show that the tests won't succeed unless a new feature is implemented in the code. If the test passes at this time there are two explanations: either the test is badly written or the feature that is being tested is already available in the code, supposing that it is an update on existing code that is being produced and not new code from scratch.
3. Then, the code must be developed with the only objective being passing the tests. If any other feature is detected, then the tests must be updated before it is introduced in the code. In that case, the process restarts from step 1.
4. After the code passes all tests it must be cleaned and improved if necessary in order to meet internal quality standards. After passing final tests it is considered stable and may be integrated into the main body of code of the system.

The contracts that were developed in [Chapter 8](#) and the dynamic models of [Chapter 9](#) are exceptional sources of information for developing complete and consistent test cases before producing any code, as explained in the following sections.

The motivation for TDD is to incentivize the programmer to keep the code simple and to produce a test asset that allows any part of the system to be automatically tested.

11.5 Unit testing

Unit tests are the most elementary tests and consist of verifying if an individual component (unit) of the software was implemented correctly. That component may be a complex method, or an entire class. Usually, this unit is still disconnected from the system it will be integrated with later.

Unit tests are usually performed by the programmer and not by the test team. TDD requires that the programmer develop the test code *before* writing the code to be tested. This test code is a driver that must systematically generate the set of data necessary to test all valid and invalid equivalence sets, applying limit value analysis when possible.

An example of unit testing consists of checking if a method was correctly implemented in a class. Take a look again at [Figure 10.13](#) and delegate method 3.2, *incrementItem(aBook:Book; aQuantity:Natural)*, which must be implemented in the *Cart* class.

Now let us consider what the equivalence sets are for these parameters. First let us consider the parameter *aBook:Book*. As the type of the parameter is *Book*, it can be assumed that no other kind of object could be passed as an argument. For example, this operation would never receive an instance of *Customer* or *Delivery* instead of a *Book*² because the language typing mechanisms would avoid that. The team must be concerned about whether objects that *can* be received here are valid or not.

If the language mechanisms cannot prevent a method from receiving a null value as a parameter, then the null case would always be considered a (usually invalid) equivalence set for any parameter that is typed with a class name.

The *incrementItem* method was developed with the following goal: to increment the quantity of an item that is already in the cart. We may start by considering that there are three equivalence sets for the *aBook* parameter:

- A valid equivalence set that contains instances of *Book* that are linked to an item in *aCart*.
- An invalid equivalence set that contains instances of *Book* that are not linked to any item in *aCart*.
- An invalid equivalence set that contains only the *null* value.

The second parameter of *incrementItem* is a natural number identified as *aQuantity*. The operation assumes that the existing quantity of the item already in the order will be incremented by *aQuantity*. The *Natural* type does not include zero and thus that potentially invalid quantity is not allowed as an argument. However, ordering a number of copies of a book that exceeds its quantity in stock could be considered an error condition unless the system allows ordering books that are not yet available. Assuming that the system operation *add2Cart* that calls *incrementItem* has not defined as a precondition that the order quantity is available in stock (see its contract in the next section), then the *incrementItem* operation may receive invalid quantities regarding that business rule. Therefore, the following equivalence sets may be considered for the *aQuantity:Natural* parameter:

- A valid equivalence set that includes natural values that added to the quantity of the respective item produce a result that is less or equal to the *quantityInStock* attribute of *aBook*.
- An invalid equivalence set that includes all natural values that added to the quantity of the respective item are greater than the *quantityInStock* attribute of *aBook*.

The potential test cases for *incrementItem* are combinations of the equivalence sets for the two parameters, as seen in [Table 11.1](#).

Table 11.1

Combinations of Equivalence Sets for *IncrementItem*

		aBook		
		aBook is Linked to an Item in the Cart (Valid)	aBook is not Linked to an Item in the Cart (Invalid)	aBook is Null (Invalid)
aQuantity	aQuantity plus current quantity is less than or equal to stock (valid)	Success	Exception	Exception
	aQuantity plus current quantity is greater than stock (invalid)	Exception	Exception (cannot be tested)	Exception (cannot be tested)

There is only one success condition, which is obtained when valid values are obtained for both parameters. It happens when *aBook* is linked to an item in the cart and *aQuantity* plus the current quantity is less than or equal to the quantity in stock. In OCL the condition can be expressed in the context of *aCart* as

```
self.item.book->include(aBook) and  
self.item->select(book=aBook).quantity+aQuantity<=aBook.quantityInStock
```

Although there are five combinations in [Table 11.1](#) that produce failure, only three have to be tested because if the book is not linked to any item or if the book is null, the value of *aQuantity* cannot be determined to be valid or invalid because there is no current quantity to be compared to it. Usually the rules to obtain combinations for testing are:

- Get all the success combinations and define a test case for each of them.
- Get one invalid set at a time. Even if the operation has *n* parameters, consider only one invalid set for just one parameter at a time. The other parameters should be valid or it should be impossible to determine their validity.
- Only take a combination with more than one invalid set if that combination is possible and useful.

Therefore, there are three failure tests that must be performed to determine how *incrementItem* behaves with invalid parameters. The first happens when *aBook* is not in *aCart*. The second happens when *aBook* is null. The third happens when the desired quantity surpasses the quantity in stock:

- (1) `not self.item.book->includes(aBook)`
- (2) `aBook.isNull()`
- (3) `self.item->select(book=aBook).quantity+aQuantity > aBook.quantityInStock`

In order to avoid tests that leave side effects on the system, they are usually performed in three steps:

1. *Prepare the stage for the test.* Usually special objects called *fixtures* are created with the sole purpose of allowing the test to be performed.
2. *Execute the test.*
3. *Clean the environment.* The system must return to the state it was in prior to the test.

Fixtures must be disposed of at this point.

Let's consider first the success test scenario for *incrementItem*. Any fixture created for this test should evaluate true for the success condition mentioned above. An example of code for generating such fixtures is the following:

```
CLASS Driver4Cart  
  
METHOD testIncrementItemSuccess()  
  
FIXTURE VAR aBook:Book  
FIXTURE VAR aCart:Cart  
FIXTURE VAR anItem:Item  
aBook:=Book.Create('0752201360',
```

```
'Bring me the head of Willy the mailboy!', 'Scott Adams',
US$12.30, 128, 5) -- quantity in stock is 5.
aCart:=Cart.Create() -- cart id is assigned internally
anItem:=Item.Create(aBook,aCart,3)
...

```

As we are going to test the *Cart* class it is possible that some of its methods or methods from other classes such as *Book* and *Item* are not yet implemented. In that case, stubs could be used here to allow the test on *incrementItem* to be performed.

After the initialization code, the test itself may be performed:

```
...
aCart.incrementItem(aBook,2) -- using limit value analysis
IF anItem.getQuantity()=5 THEN
Write('incrementItem success test: successful')
ELSE
Write('incrementItem success test: failed')
ENDIF
...

```

Finally, we must dispose of the fixtures:

```
...
anItem.destroy()
aCart.destroy()
aBook.destroy()
END METHOD

```

Now, the failure conditions should be tested as well. Here, they are implemented as three independent methods in the *Driver4Cart* class:

```
...
METHOD testIncrementItemFailure1 ()
-- not self.item.book->includes(aBook)
FIXTURE VAR aBook:Book
FIXTURE VAR aCart:Cart
aBook:=Book.Create('0752201360',
'Bring me the head of Willy the mailboy!', 'Scott Adams',
US$12.30, 128, 5)
aCart:=Cart.Create() -- the book is not in the cart
TRY
aCart.incrementItem(aBook,2)
Write('incrementItem failure test 1: failed')
CAPTURE Exception.itemAbsent
Write('incrementItem failure test 1: successful')
END TRYCAPTURE
aCart.destroy()

```



```

aBook.destroy()
END METHOD
METHOD testIncrementItemFailure2 ()
-- aBook.isNull()
FIXTURE VAR aBook:Book
FIXTURE VAR aCart:Cart
aCart:=Cart.Create() -- the book is null
TRY
aCart.incrementItem(aBook,2)
Write('incrementItem failure test 2: failed')
CAPTURE Exception.bookIsNull
Write('incrementItem failure test 2: successful')
END TRYCAPTURE
aCart.destroy()
END METHOD
METHOD testIncrementItemFailure3 ()
-- item->select(book=aBook).quantity+aQuantity > aBook.quantityInStock
FIXTURE VAR aBook:Book
FIXTURE VAR aCart:Cart
FIXTURE VAR anItem:Item
aBook:=Book.Create('0752201360',
'Bring me the head of Willy the mailboy!', 'Scott Adams',
US$12.30, 128, 5) -- quantity in stock is 5.
aCart:=Cart.Create() -- cart id is filled internally
anItem:=Item.Create(aBook,aCart,3))
TRY
aCart.incrementItem(aBook,3) -- limit analysis
Write('incrementItem failure test 3: failed')
CAPTURE Exception.quantityAboveStock
Write('incrementItem failure test 3: successful')
END TRYCAPTURE
anItem.destroy()
aCart.destroy()
aBook.destroy()
END METHOD
END CLASS

```

The style of the code of the test case depends on the language and framework used. Here a generic pseudocode implementation is illustrated to allow the logic of the test to be understood by readers with knowledge about programming languages but not necessarily experience with test frameworks.

As we can see in the code above, fixtures and test procedures are usually similar, and thus uniform and parameterized methods could be used to create such objects, reducing the quantity of code that is being produced. This is usually provided by frameworks such as *xUnit*.

Unit tests may be extremely minimized if an automatic code generator is used to generate the basic operations and delegate methods. In that case, it may be assumed that the code is correct because the generator does not make human mistakes. The test could begin at the level of system operations, which is explained in the next section.

11.6 System operations testing

The top level of functional testing occurs when the team verifies if the system operations really perform according to the contract specifications.

System operations may call methods from many classes, delegate responsibilities, and coordinate the execution of basic and delegate methods. Thus, at the level of integration of the classes that perform a consistent set of responsibilities, the system operation test verifies if each system operation correctly implements its contract.

The system operations test, then, consists of verifying if in normal conditions (preconditions are true and exceptions false), the desired postconditions are really obtained, and if in abnormal conditions, exceptions are effectively raised. System operation tests are very similar to unit tests, except that now the operation has a contract, and this eases the identification of the necessary test cases.

Let us now examine the system operation *add2Cart* in [Figure 10.13](#). Its contract was defined in [Chapter 8](#) and is the following:

```
Context Livir::add2Cart(aCartId:CartId, anIsbn:Isbn, aQuantity:Natural)
```

```
def:aCart=cart[aCartId]
def:aBook=book[anIsbn]
def:anItem=aCart.item->select(book.isbn=anIsbn)
pre:
  aCart->notEmpty() and
  aBook->notEmpty()
post:
  if anItem->isEmpty() then
    newItem.isNewInstanceOf(Item) and
    aCart^addItem(newItem) and
    newItem^setQuantity(aQuantity) and
    newItem^addBook(aBook) and
    newItem^setUnitPrice(aBook.price)
  else
    anItem^setQuantity(anItem.quantity@pre + aQuantity)
  endif
```

exception:

```
anItem.quantity+aQuantity > aBook.quantityInStock implies  
Exception.throw('quantity not available')
```

The precondition $aCart \rightarrow \text{notEmpty}()$ implies that there is an invalid set for the parameter $aCartId$. The precondition $aBook \rightarrow \text{notEmpty}()$ implies that there is an invalid set for the parameter $anIsbn$. Finally, the exception in that contract establishes that there are invalid values for $aQuantity$ as well.

Regarding valid values, each parameter must have at least one valid equivalence set, otherwise the method would always fail. However, as the postconditions are conditional (they include an *if-then-else-endif* structure), there are at least two possible behaviors for this operation, and this defines two valid equivalence sets: one when $anItem$ is empty and another when $anItem$ is not empty. The test cases for this system operation are presented in [Table 11.2](#).

Table 11.2
Functional Test Cases for a System Operation

Type	Condition to Be Tested	Expected Result
------	------------------------	-----------------

Success

```
aCart->notEmpty() and  
aBook->notEmpty() and  
anItem->isEmpty() and  
aQuantity <=  
aBook.quantityInStock
```

```
aCart.item->  
select (book=aBook).quantity=  
aQuantity
```

Success

```
aCart->notEmpty() and  
aBook->notEmpty() and  
anItem->notEmpty() and  
anItem.quantity+aQuantity <=  
aBook.quantityInStock
```

```
anItem.quantity=  
anItem.quantity@pre+  
aQuantity
```

Exception

```
aCart->isEmpty()
```

```
'There is no such cart'
```

Exception

```
aBook->isEmpty()
```

```
'There is no such book'
```

Exception

```
anItem->notEmpty() and  
  anItem.quantity+aQuantity <=  
  aBook.quantityInStock
```

```
'Quantity not available'
```

Combinations of exceptional conditions could be tested as well. For example, one situation that does not appear in [Table 11.2](#) is when both the ISBN and cart ID are invalid. If that is the case, just one of the exceptions would be raised in any case. Furthermore, as mentioned before, sometimes invalid conditions are not compatible. For example, if the ISBN is invalid, there is no *quantityInStock* to be compared to the ordered quantity, and thus the corresponding exceptions could never occur together.

This is usually why the only combinations that are tested are the ones that just include success conditions or just one failure condition combined with success conditions. However, if the team finds it easy and safer to add new variations that include combinations of invalid failure conditions that may be implemented.

Thus, although not every combination is tested, it is necessary to test at least:

- All possible combinations of valid equivalence sets.
- All combinations of one invalid equivalence set with valid sets.

This means that after testing the valid combinations, the tester must include one invalid set at a time for the exception test.

The code for testing system operations is very similar to the one presented for unit tests: fixtures must be created before each test, success and failure tests are run, and the environment is cleaned after each test.

11.7 Use case testing (System, acceptance, and business cycle tests)

Use case tests consist of manually or automatically following all the possible flows of a use case and checking if the desired results (usually stated as the use case postconditions) are obtained.

If the test is run over a single use case by the development team, then it is a *system test*. If the test is run by the client or under her supervision, then it is an *acceptance test*. If a whole set of system use cases related to a business use case are performed in a logical sequence such as the one defined by the business activity diagram, then it is a *business cycle test*.

The *use case test* aims to verify if the current version of the system correctly performs the use case processes from the point of view of a user that performs a sequence of system operations in an interface (not necessarily a graphic interface) and is able to obtain the expected results.

The use case test may be understood as a systematic execution of the use cases flows. If each of the system operations and queries has already passed their own tests, then it must be verified that the main flow and alternate flows of the use case can be performed correctly and that they produce the desired results.

It is possible to automate these tests so that they can be performed by a test robot that makes calls directly to the controller or simulates the actions of the user on the graphic interface.

Consider, for example, a system test that must be conducted to evaluate the use case of [Figure 5.7](#), reproduced here as [Figure 11.1](#).

Use case 01: *Order books*

1. The customer provides keywords for searching books.
2. The system generates a list of books for sale that matches the keywords, including at least the title, author's name, price, page count, publisher, ISBN, and cover image.
3. The customer selects books from the list and indicates the desired quantity for each one.
4. The system generates the order summary (title, author, quantity, unit price, and subtotal for each book) and total value.
5. The customer finishes the order.

Exception 3a: The quantity requested is higher than the quantity in stock for one or more books.

3a.1. The system informs the use of the available quantities for each book that was ordered over availability.

3a.2. The user changes the desired quantities to meet the quantities in stock.

3a.3. Advance to step 4.

Exception 5a: The customer is not yet identified.

5a.1. The customer provides valid identification.

Exception 5b: The customer does not have an account.

5b.1. The customer registers herself by providing a username, password, and email address.

Exception 5c: No book was selected for purchase.

5c.1. Return to step 1.

Variant 5d: The customer decides to continue shopping.

5d.1. Return to step 1.

FIGURE 11.1 Reference use case.

A use case test would deal with scenarios that are successful but run through different paths. First of all, the main flow of the use case must be considered as a base scenario for test purposes. Then, each variant or exception may be included one at a time for testing alternate scenarios. This is the least that is expected of a use case test suite. Combinations of exceptions and variants may be tested as well if the team considers them useful and viable.

The technique for elaborating use case tests then consists of selecting a set of paths that passes at least once by each variant and exception flow. The test cases for the example of [Figure 11.1](#) may be defined as the following:

- 1,2,3,4,5.
- 1,2,3a(1,2,3),4,5.
- 1,2,3,4,5a(1).
- 1,2,3,4,5b(1).

- 1,2,3,4,5c(1),1,2,3,4,5.
- 1,2,3,4,5d,1,2,3,4,5.

The system test plan for the use case of [Figure 11.1](#) is shown in [Table 11.3](#).

Table 11.3
Example of a Test Plan for a Use Case

Goal	Path	How to Test	Result
Main flow	1,2,3,4,5	A previously identified customer searches and selects at least one book and finishes the order.	An order is created with the book(s) selected.
Exception 3a	1,2,3a(1,2,3),4,5	A previously identified customer searches and selects a quantity of at least one book that is above stock. Then she changes the quantity to meet the stock and finishes the order.	An order is created with the book(s) selected.
Exception 5a	1,2,3,4,5,5a(1).	A registered customer that was not yet identified yet searches and selects at least one book. She provides identification and finishes the order.	An order is created with the book(s) selected.
Exception 5b	1,2,3,4,5,5b(1).	A nonregistered customer searches and selects at least one book. She registers herself and finishes the order.	An order is created with the book(s) selected. A record for a new customer is created.
Exception 5c	1,2,3,4,5,5c(1),1,2,3,4,5.	A previously identified customer tries to finish an empty order and is prevented from doing so by the system. Then she searches and selects at least one book and finishes the order.	An order is created with the book(s) selected.
Variant 5d	1,2,3,4,5d,1,2,3,4,5	A previously identified customer searches and selects at least one book. Then she searches and selects another book and finishes the order.	An order is created with the books selected.

The use case test is only performed on a system version in which all system operations were already tested. If many errors are still detected in system operations during the system test, the usual approach is to abort the use case test and redo the system operation tests until a sufficiently stable version of the system is produced and made available for use case tests.

Although automatic use case testing³ may check if the functionality is correct or not, human-powered use case tests are also recommended because there are aspects of the test that are hard for a machine to assess. For example, are the error messages clear? Are the buttons and text fields well positioned and sized?

Furthermore, a whole set of *nonfunctional tests* may be necessary depending on the supplementary specifications and nonfunctional requirements of the system ([Chapter 3](#)). Examples of nonfunctional tests are compatibility, compliance, endurance, load,

localization, performance, recovery, resilience, security, scalability, stress, and usability. Explaining all these kinds of tests is beyond the scope of this book. Nonfunctional testing demands heterogeneous techniques and are covered by a large number of books such as: [Molyneaux \(2009\)](#) on performance testing, [Nelson \(2004\)](#) on stress testing, and [Nielsen \(1994\)](#) on usability testing.

11.8 The process so far

	Inception	Elaboration
Business Modeling	Build a general view of the system: <ul style="list-style-type: none">• Build a business use case diagram and determine the automation scope for the project.• Build preliminary activity diagrams for business use cases.• Build preliminary state machine diagrams for key business objects.	

(Continued)

	Inception	Elaboration
Requirements	<p>Prepare the system use case diagram (functional requirements):</p> <ul style="list-style-type: none"> • Identify the system actors from the business use case model. • Identify the system use cases from the business use case model, and activity and state machine diagrams from business modeling. <p>Identify nonfunctional requirements as use case annotations:</p> <ul style="list-style-type: none"> • Identify the main business rules associated to use cases. • Identify the main quality issues associated to use cases. <p>Identify supplementary requirements.</p>	<p>Detail requirements by expanding use cases:</p> <ul style="list-style-type: none"> • Identify the main flow. • Identify the alternate flows: variants and exception handlers.
Analysis and Design	<p>Prepare the preliminary conceptual model by observing system use cases and the concepts needed by them.</p>	<p>Elaborate the system sequence diagrams:</p> <ul style="list-style-type: none"> • Represent the main flow of a use case as a system sequence diagram. • Represent the system commands and queries using stateful or stateless strategies. • Complete system sequence diagrams with alternate flows. <p>Refine the conceptual model:</p> <ul style="list-style-type: none"> • Identify concepts, attributes, and associations in the text of expanded use cases. • Detail attributes and associations with stereotypes, multiplicity, and constraints as needed. • Organize the model by using inheritance, association classes, and temporal specifications. • Add invariants as needed. • Improve the conceptual model with the application of analysis patterns. <p>Write system operation contracts for commands and queries in system sequence diagrams:</p> <ul style="list-style-type: none"> • Identify preconditions and exceptions based on invalid parameters and complementary constraints. • Identify postconditions for commands and returns for queries.

(Continued)

	Inception	Elaboration
Implementation		<p>Design the domain tier:</p> <ul style="list-style-type: none"> • Create a sequence or communication diagram for each system command contract. • Use those diagrams to decide which methods to implement in each class. • Inspect those diagrams to discover which associations can be unidirectional and define them as such. • Look at system query contracts and decide which delegate queries could be implemented; some of them could be derived attributes or derived associations. • Produce or refine the design class diagram. <p>Generate code:</p> <ul style="list-style-type: none"> • Generate code for classes. • Generate code for attributes. • Generate code for associations. • Generate code for necessary basic methods such as get, set, add, remove, create, destroy, and others. • Generate code for system operations and delegate methods using dynamic models as reference. <p>Plan and execute tests:</p> <ul style="list-style-type: none"> • Perform unit tests for methods and classes that were not generated automatically. • Perform system operation tests based on system operation contracts. • Perform use case tests based on use case scenarios.
Test		
Project Management	<p>Estimate total effort, ideal calendar time, and average team size for the project.</p> <p>Estimate the duration and quantity of iterations for each phase.</p> <p>Prepare the phase plan and iteration plan for the first iteration.</p>	

11.9 Questions

1. Consider the methods defined in [Section 10.2](#) for the class in [Figure 10.1](#), and write a sequence of unit tests for each of the basic operations defined for that class.
2. Look at the four contracts for the *deleteBook* operation in [Section 8.7.3](#). Write a system test in your favorite language or in pseudocode to test each of the versions of that operation.
3. Prepare a use case test table for the use case in [Figure 5.4](#).

¹Although literature usually uses the term *equivalence classes*, which is a mathematical concept, in this book the term *equivalence sets* is used instead to avoid becoming confused with object-oriented classes.

²Unless they were subclasses of *Book*, which is not the case.

³An example of a robot framework for interface testing is:

<http://robotframework.googlecode.com/hg/doc/userguide/RobotFrameworkUserGuide.html?r=2.7.4>. Accessed September 1st, 2013.

CHAPTER 12

Interface Tier Design with IFML

This chapter shows how to design a web interface for an information system using the Interaction Flow Modeling Language (IFML). Initially, the information view components of IFML are presented. They are used in connection with the design class diagram to manage the presentation of information. The integration of view components and flows into containers such as pages and areas is presented as well as some interface design patterns. Operation components may also be used in connection with view components to define basic operations over objects. The chapter concludes with the presentation of a complete CRUD interface modeled in IFML and a high-level design for a system use case.

Keywords

IFML; Interaction Flow Modeling Language; user interface; user interface pattern; view component; web navigation; web page

Key Topics in this Chapter

- The interaction flow modeling language (IFML)
- View components
- Hypertext organization
- Web page patterns
- Operations in IFML

12.1 Introduction to interface tier design

The interface tier design of a system depends fundamentally on the detailed use cases or system sequence diagrams, because the interface must allow a user to follow all use case flows. Design class diagrams and contracts may be very useful as well.

During interface design the nonfunctional requirements that were annotated with the use cases should be revised again, because they may contain indications on how to design the use case interface.

The Interaction Flow Modeling Language (IFML) is an extension of UML for modeling data intensive user interfaces. It is an evolution of WebML (Ceri, Fraternali, Bongio Brambilla, Comai, & Matera, 2003). IFML is being adopted as an OMG standard.¹

This chapter intends to present the most fundamental concepts of IFML, and its modeling potential. More information may be found at the official site of the language.² As IFML is still evolving as a pattern, the reader may expect to find some discrepancies between this text and the current specification of IFML, which may have changed after this book was published.

12.2 Interaction flow modeling language (IFML)

IFML is suitable for modeling interfaces, especially those that make intensive use of data such as information systems. The development method proposes the building of five models in order to define an application:

- *Structural model*: This represents the data structure and data organization. Ceri et al. use the entity-relationship diagram to define the structural model. However, the UML class diagram (Chapters 6 and 7) may be used for the same purpose.
- *Derivation model*: This was originally proposed for modeling data that can be calculated from other data. The derivation model may be understood as the set of definitions of derived attributes and associations that usually are defined in the conceptual model and may be written in OCL (Sections 6.2.3 and 6.4.3).
- *Composition model*: This includes the definition of containers as aggregates of basic *view components* (Section 12.3), *pages* (Section 12.4), and *areas* (Section 12.6.2).
- *Navigation model*: Where the flows between pages and view components are defined (Sections 12.5 and 12.6).
- *Presentation model*: Where the positioning of view components at the interface and their appearance are defined.

This chapter addresses particularly the composition and navigation models. The structural and derivation models were already discussed in Chapters 6 and 7). For the presentation model, the WebRatio® tool³ provides the use of XSL Style Sheets⁴. The code generator of the tool uses these presentation rules to produce pages according to the design produced with other tools. For the examples presented in this chapter, the standard interfaces rendered by the tool, without the use of styles, are shown. Even names of fields were left identical to the names of class attributes in the aim of promoting understandability instead of graphic quality.

12.3 View components

View components are the most basic elements in an IFML specification. View component may be directly associated to design classes and allow the viewing of instances of these classes.

There are many types of view components in IFML. Some examples are:

- *Details*: Displays information about a single object.
- *Multiple details*: Displays information about a collection of instances of the same class.
- *Simple list*: Displays multiple instances of a class in the form of a list.
- *List*: An enhancement of *simple list* that allows scrolling and dynamic sorting.
- *Checkable list*: A list that allows multiple selection.
- *Hierarchy*: Displays multiple instances of different classes organized in a hierarchy, such as master-detail or whole-part.
- *Recursive hierarchy*: Displays multiple instances of a single class that are organized hierarchically, such as organizational structures.
- *Scroller*: Allows the typical scroll operations on a sequence of objects.
- *Form*: Allows form-based data entry.

- *Multiple form*: Similar to form, but allows editing multiple objects at once.
- *Message*: Allows printing messages on the page.
- *Calendar*: Shows a perpetual calendar.

There are other view components and new ones may be defined. This chapter presents details on the most fundamental ones. For each kind of view component a graphical definition is presented, as well as the possible appearance of the web page rendered by WebRatio for the definition. All examples are based on the DCD of [Figure 12.1](#).

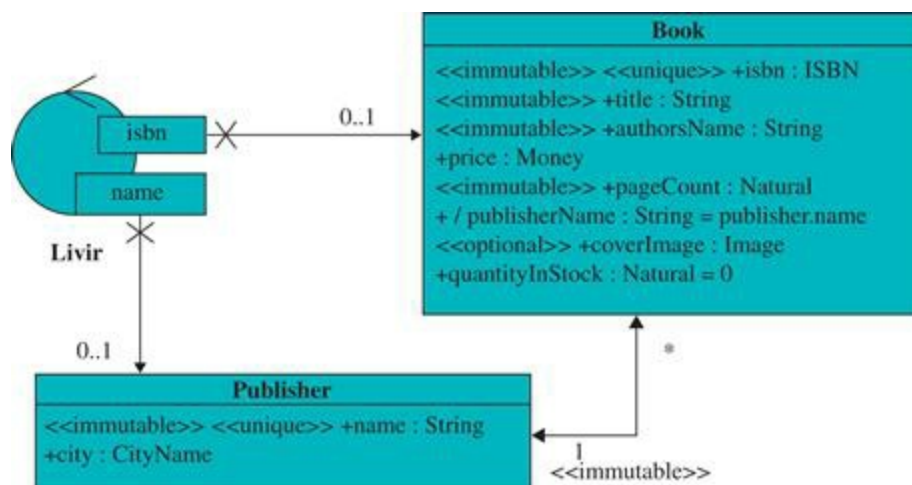


FIGURE 12.1 Reference design class diagram.

12.3.1 Details

The *details* view component presents information about a single object of a given class. It is defined by the following properties:

- A *name*, which is chosen by the designer.
- An *entity* or *source of data*, which is usually a class from the design class diagram.
- *Display attributes*, which is a list of attributes from the class to be displayed when the details view is rendered.

Most view components such as *details* also allow for the definition of one or more *conditional expressions* that work like selectors or filters over the set of instances to be shown. There are three kinds of conditional expressions:

- *Key condition*: Each object in the data model is identified by an *OID (Object Identifier)* – an attribute that is unique, mandatory, and immutable. It must not be confused with any of the attributes of a conceptual class, even the ones stereotyped with `<<unique>>` and `<<immutable>>`, such as ISBN and Social Security number. The OID is an internally generated code that corresponds to the *primary key* of an object. OIDs *cannot be known or updated by the user*. A key condition allows one to select one or more instances from a given class based on the value of its OID attribute.
- *Attribute condition*: This allows for the selection of one or more instances based on the values of their attributes. Operations such as *greater than*, *equal*, or *contains* may be used to compare the value of an attribute to a given parameter.

- *Relationship role condition*: This allows for the selection of one or more instances based on their association links.

Figure 12.2 presents a details view component intended to show data about an instance of a book for a given ISBN. The conditional expression `isbn=0517149257` determines which instance is shown. If such an instance does not exist, then no information is rendered by that component.



FIGURE 12.2 A details view component with a simple attribute condition.

The *attributes* property usually does not appear in the graphical representation. If a modeling tool is being used, this property may probably be accessed by browsing the component. As an ISBN is an attribute that is unique and mandatory, it is safe to assume that no more than one book is presented by this details view.

Figure 12.3 shows a possible web rendering for the details view component shown in Figure 12.2.

Book Details	
title	The Ultimate Hitchhikers Guide
authorsName	Douglas N. Adams
isbn	0517149257
price	122
pageCount	815
coverImage	
quantityInStock	4

FIGURE 12.3 A possible rendering for a details view component with all attributes of class *Book*.

The final rendered page may vary, because it is possible to add style definitions so that interfaces with different designs and appearance are produced.

If the details view component of Figure 12.2 is changed to show only the *title* and *authorsName* attributes, it would produce the rendered result shown in Figure 12.4.

Book Details	
title	The Ultimate Hitchhikers Guide
authorsName	Douglas N. Adams

FIGURE 12.4 A possible rendering for a details view with fewer attributes selected.

12.3.2 Multiple details

Multiple details presents a group of instances of a single class at once. Besides the three properties already shown for details view components, they add new optional properties:

- *Sort attributes*: Defines the attributes used for sorting the instances.
- *Max results*: Specifies the maximum number of instances to be retrieved. By default, all instances are retrieved.
- *Distinct*: Specifies that duplicate elements will not be shown more than once.

Multiple details also allows for the definition and use of conditional expressions in order to select which elements are to be shown. [Figure 12.5](#) presents an example of multiple details.

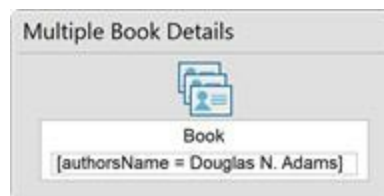


FIGURE 12.5 A multiple details view component.

[Figure 12.6](#) presents a possible rendering for the multiple details of [Figure 12.5](#) considering that only three books satisfy the conditional expression and that sorting is by title.

Multiple Book Details						
isbn	title	authorsName	price	pageCount	coverImage	quantityInStock
> 0671746723	Dirk Gently's Holistic Detective Agency	Douglas N. Adams	6.99	306		12
> 0671742515	The Long Dark Tea-Time of the Soul	Douglas N. Adams	6.99	307		4
> 0517149257	The Ultimate Hitchhikers Guide	Douglas N. Adams	122	815		4

FIGURE 12.6 A possible rendering for a multiple details view component.

As opposed to the details view component that shows only one instance at a time, multiple details presents all the objects that satisfy the conditional expression.

If it is necessary to use more than one attribute to determine the objects to be presented by the component, then a combination of conditional expressions may be used. This combination may use the operands *and* and *or*. [Figure 12.7](#) presents a multiple details view component that shows instances that have *pageCount* higher than 100 *and* price lower than US\$150.00.



FIGURE 12.7 Multiple details view with a composite conditional expression.

12.3.3 Simple list

The *simple list* presents one or more attributes of a set of instances of a class in the form of a list. While multiple details are normally used to visualize the details of many objects in the same page, a simple list and its variations are used when a list or menu is necessary for the user to select one or more elements and perform actions with them. The properties of simple lists are the same as those of the multiple details view component.

It is possible, for example, to define simple lists for selecting books based on their titles. [Figure 12.8](#) presents the view component for a simple list of book titles, and [Figure 12.9](#) shows a possible rendering. The absence of a conditional expression in the simple list view component means that all instances of *Book* are listed.

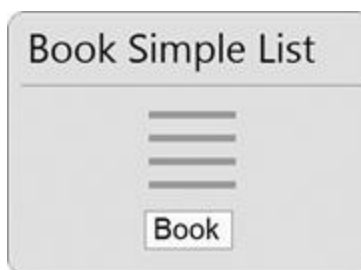


FIGURE 12.8 Simple list.



FIGURE 12.9 Possible rendering of a simple list.

12.3.4 List

A *list* is an enhanced version of the simple list that allows one to scroll elements and dynamically reorder them by a simple click on the head of the respective attribute column. Besides the properties already defined for simple lists, it includes the following specific properties:

- *Sortable*: A Boolean attribute that when true defines that the list may be dynamically sorted.
- *Default sort attributes*: If the list is sortable it defines the sorting attributes that may be selected by the user to sort the list and also the default sorting criteria when the list is rendered for the first time.
- *Sort history size*: Specifies how many sort actions from the user must be remembered by the list.
- *Checkable*: If true, a check box is rendered for each element of the list.
- *Block factor*: Specifies how many elements are shown at a time. If unspecified or set to -1 , all elements are shown. If the block factor is defined, then scrolling commands are rendered for the list.

Figure 12.10 shows the IFML view component for a list. This list is defined for the *Book* class with three attributes: *authorsName*, *title*, and *price*. The block factor is defined as 3. Figure 12.11 shows a rendered version of that list.

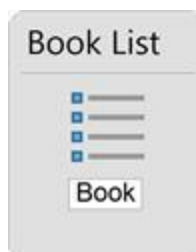


FIGURE 12.10 List.

Book List			
firstprevious 1 to 3 of 8nextlast			
jump to 1 2 3			
	title	authorsName	price
>	Dirk Gently's Holistic Detective Agency	Douglas N. Adams	6.99
>	Foundation and Empire	Isaac Asimov	5.99
>	Rama II	Arthur C. Clarke	6.99

FIGURE 12.11 Possible rendering for a list.

Observe at the top of [Figure 12.11](#) the commands to scroll the list from the first to the last group of three elements. Currently the first three elements are being shown, sorted by their title.

If the user clicks one of the links at the top of each data column, then the elements on the list are rearranged and sorted by that column. If the user clicks, for example, on *authorsName*, the lines will be sorted by the author's names, as seen in [Figure 12.12](#).

Book List			
firstprevious 1 to 3 of 8nextlast			
jump to 1 2 3			
	title	authorsName	price
>	The Hammer of God	Arthur C. Clarke	6.99
>	Rama II	Arthur C. Clarke	6.99
>	The Revenge of the Baby-Sat	Bill Watterson	8.95

FIGURE 12.12 Rendering of [Figure 12.11](#) after a user action.

Notice that the elements shown are different because the entire list is sorted again and the elements that are being viewed are the first three.

12.3.5 Checkable List

The *checkable list* allows the user to select a set of elements from a list. Otherwise it is very similar to a simple list. [Figure 12.13](#) presents the IFML definition of a checkable list and [Figure 12.14](#) presents its rendering.



FIGURE 12.13 A checkable list.

Checkable Book List			
	title	authorsName	price
<input type="checkbox"/>	Shave the Whales	Scott Adams	6.99
<input checked="" type="checkbox"/>	The Revenge of the Baby-Sat	Bill Watterson	8.95
<input type="checkbox"/>	Foundation and Empire	Isaac Asimov	5.99
<input checked="" type="checkbox"/>	Rama II	Arthur C. Clarke	6.99
<input checked="" type="checkbox"/>	The Hammer of God	Arthur C. Clarke	6.99
<input type="checkbox"/>	The Long Dark Tea-Time of the Soul	Douglas N. Adams	6.99
<input type="checkbox"/>	Dirk Gently's Holistic Detective Agency	Douglas N. Adams	6.99
<input type="checkbox"/>	The Ultimate Hitchhikers Guide	Douglas N. Adams	122

FIGURE 12.14 Rendering for a checkable list.

While *simple lists* allow the selection of only one object at a time, checkable lists allow the selection of any quantity of objects.

12.3.6 Forms

A *form* is used for data input. It is very useful for creating new instances, and also to provide parameters for searches, queries, and commands.

A form is composed of a set of *fields*. Each field holds an alphanumeric value. There are text fields, selection fields, and multiselection fields.

A form may or may not be associated to a class. If it is associated to a class, then its fields may be associated to the attributes of a class.

Forms that are used to collect parameters for a query or command usually do not need to be associated to classes. For example, a form that collects keywords for searching books does not need to be associated to any class.

The fields of a form have the following properties (among others):

- *Name*: The name of the field, given by the designer.
- *Attribute*: If the form is associated to a class, then the field may be associated to one of its attributes.
- *Content type*: Defines the type of the field, which may be *string*, *text*, *blob*, or *url*.

- *Preloaded*: The field may be preloaded with values that are shown to the user when the form is rendered.
- *Modifiable*: Defines whether the initial value of the field may be changed or not. The default is that a field value may be changed.

Figure 12.15 presents a form that is associated to the *Book* class, and Figure 12.16 presents its rendering.



FIGURE 12.15 A form.

FIGURE 12.16 Rendering of a form.

12.3.7 Hierarchies

Hierarchies are used to show object dependencies with two or more levels. They are useful, for example, for showing objects that are related by a one-to-many association such as *Publisher* and *Book*, or whole-part associations such as *Book* and *Chapter*. Figure 12.17 shows a definition of a two-level hierarchy with *Publisher* at the top and *Book* at the second level.

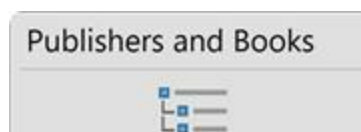




FIGURE 12.17 A hierarchy.

The second level usually defines a *role-based conditional expression*. In the example, the role-based conditional expression defines that only books that are linked to a given publisher are shown below it. In the absence of a conditional expression, all books would be shown below each publisher, but that is not the case now.

Figure 12.18 shows a rendering for the hierarchy of Figure 12.17 with books linked to their respective publishers.



FIGURE 12.18 Rendering of a hierarchy.

There are situations that require the application of the *organizational hierarchy* pattern explained in Section 7.8, where a concept has subconcepts that belong to the same class, which may be recursively nested. For these cases, IFML provides a *recursive hierarchy* view component. Figure 12.19 shows a recursive hierarchy defined for the *OrganizationalStructure* class shown in Figure 7.15.



FIGURE 12.19 A recursive hierarchy.

The definition and rendering of a recursive hierarchy is similar to that of a normal hierarchy. The only differences are that the detail class and the master class are the same and the number of levels is undetermined.

12.4 Pages

Pages are the interface elements that are effectively accessed by the user. Typically, a page contains one or more view components and possibly some other pages.

The IFML representation of a page contains a name and an optional set of included view components and subpages. For example, [Figure 12.20](#) presents a page that contains a list of publishers and a list of books, and [Figure 12.21](#) presents its rendering.

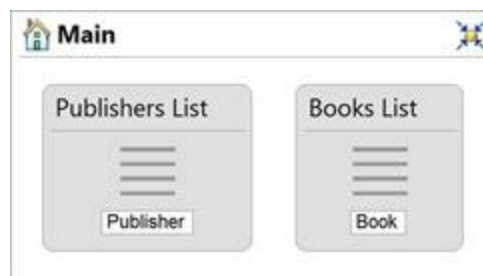



FIGURE 12.20 A page with two view components.

> Main

Publishers List	
name	city
> Andrews and McMeel	Kansas City
> Bantam	New York
> Boxtree	London
> Pocket Books	New York
> Random House	New York

Books List					
isbn	title	authorsName	price	pageCount	quantityInStock
> 0671746723	Dirk Getly's Holistic Detective Agency	Douglas N. Adams	6.99	306	12
> 0553293370	Foundation and Empire	Isaac Asimov	5.99	282	3
> 0553286587	Rama II	Arthur C. Clarke and Gentry Lee	6.99	466	2
> 0752208497	Shave the Whales	Scott Adams	6.99	128	7
> 055356871X	The Hammer of God	Arthur C. Clarke	6.99	240	1
> 0671742515	The Long Dark Tea-Time of the Soul	Douglas N. Adams	6.99	307	21
> 0836218663	The Revenge of the Baby-Sat	Bill Waterson	8.95	127	4
> 0517149257	The Ultimate Hitchhikers Guide	Douglas N. Adams	129	813	6

FIGURE 12.21 Rendering of a page with two view components.

The  symbol in [Figure 12.20](#) indicates that this page is a *homepage*, that is, the initial page of an application. The other symbol in the upper-right corner of the page does not have any special meaning in IFML; it is just a tool-specific command to highlight and obscure graphic elements.

The view components that appear inside the page in [Figure 12.20](#) are independent of each other because there are no flows between them.

By default, view components defined in the same page are rendered one on top of the other, and occupy the entire width of the page. But it is possible to use the *presentation model*, as mentioned before, to define view components in different positions and sizes. The presentation model consists basically of a grid in which the view components are positioned and dimensioned inside a page. In the example of [Figure 12.22](#) the view components were placed side to side.



Publishers List	
name	city
➤ Andrews and McMeel	Kansas City
➤ Bantam	New York
➤ Bostree	London
➤ Pocket Books	New York
➤ Random House	New York

Books List						
isbn	title	authorsName	price	pageCount	quantityInStock	
➤ 0671746723	Dirk Getty's Holistic Detective Agency	Douglas H. Adams	6.99	306	12	
➤ 0553293370	Foundation and Empire	Isaac Asimov	5.99	262	3	
➤ 0553286587	Rama II	Arthur C. Clarke and Gentry Lee	6.99	466	2	
➤ 0752208497	Shave the Whales	Scott Adams	6.99	128	7	
➤ 055356871X	The Hammer of God	Arthur C. Clarke	6.99	240	1	
➤ 0671742515	The Long Dark Tea-Time of the Soul	Douglas H. Adams	6.99	307	21	
➤ 0836218663	The Revenge of the Baby-Sat	Bill Waterson	8.95	127	4	
➤ 0517149257	The Ultimate Hitchhikers Guide	Douglas H. Adams	129	813	6	

FIGURE 12.22 A different rendering arrangement for the page shown in [Figure 12.21](#).

12.5 Flows

A *flow* is an oriented connection between two pages or view components. Flows may be used not only to define navigation possibilities between pages, but also to define data dependencies. For example, selecting an element in a list view component may cause information about the element to appear in a details view component. This can be accomplished by defining a flow from the list to the details.

12.5.1 Normal navigation flow

Normal navigation flows are a type of flow that is rendered as a clickable link in its origin page or view component. When the link is clicked it navigates to the destination page or view component. It is represented by an arrow.

[Figure 12.23](#) shows an example of a normal navigation flow between two pages. The resulting rendering of the *Publishers* page contains a link to the *Books* page (at the right top corner of [Figure 12.24](#)).

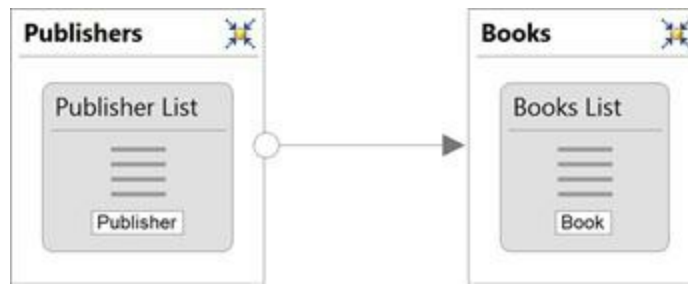


FIGURE 12.23 A normal navigation flow between pages.



FIGURE 12.24 Rendering of a normal navigation flow as a link.

We can see in [Figure 12.23](#) that the flow is not attached to the view components, but to the pages. Therefore, the link will be rendered only once in the page where the list of books is visualized.

If the flow had its origin at the list view component as in [Figure 12.25](#), then each line in the list would have an individual link to the *Books* page.

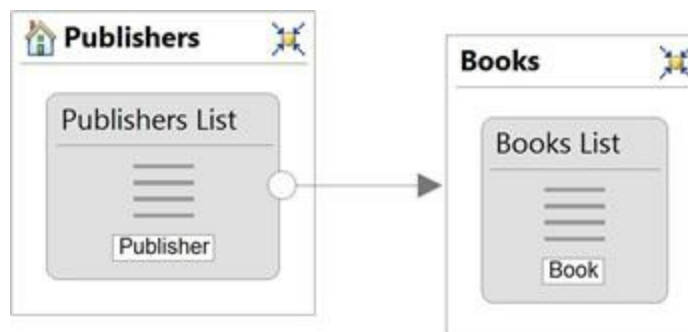


FIGURE 12.25 A flow from a view component to a page.

[Figure 12.26](#) shows how the model in [Figure 12.25](#) would be rendered.

Publishers List		
name	city	
> Bantam	New York	Books
> Pocket Books	New York	Books
> Boxtree	London	Books
> Andrews and McMeel	Kansas City	Books
> Random House	New York	Books

FIGURE 12.26 Rendering a link for each element of a list.

This list is still just a set of links that have the same effect: navigating to the *Books* page. The following sections show how this kind of flow can be used to transport information from one view component to another, making it possible, for example, to visualize only the books of a given publisher.

12.5.2 Data flow

Data flows are a kind of flow that is not rendered, but that define data dependency between view components.⁵ Dashed arrows represent data flows.

Data flows do not define navigation between view components. They are used to get values from one component without navigating from it. They are particularly useful when a component needs data from more than one component. The following sections present examples of data flows that get data from different sources.

12.5.3 Parameter binding

A *parameter binding* is a piece of information that is transmitted from the origin to the destination of a flow. A parameter binding has a *name* and a *value*, which is obtained at its origin component.

A flow may have a multiple parameter binding, passing a set of values instead of a single one.

Usually, object keys or attribute values are passed by parameter bindings. By default, a flow from a view component binds the key of the selected object at the origin component and takes it to the destination, where it can be used, for example, by a conditional expression.

In previous examples only constants were used in conditional expressions to select the elements to be shown in view components such as lists and details. Now, with flows and parameter binding, it is possible to use a view component to define which elements are shown in another view component.

The IFML diagram shown in [Figure 12.27](#) shows a list of books with a flow to a details

view component. The details view has a key condition `[oid=?]`. As the key of the selected book at the list is bound to the flow that goes to the details view, only that book is shown when the details view is rendered.

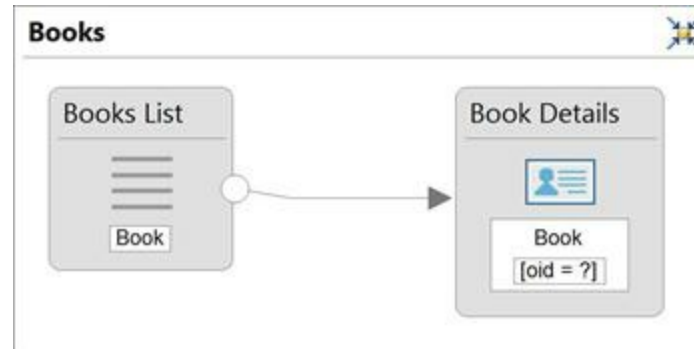


FIGURE 12.27 A normal navigation flow connecting two view components.

When the user selects a book in the list, by clicking the respective link, the details view presents information about the book. [Figure 12.28](#) shows the state of the page when the link associated to the first book is clicked.

> Books

Books List

title	authorsName	
> Foundation and Empire	Isaac Asimov	Details
> Rama II	Arthur C. Clarke and Gentry Lee	Details
> Shave the Whales	Scott Adams	Details
> The Hammer of God	Arthur C. Clarke	Details
> The Long Dark Tea-Time of the Soul	Douglas N. Adams	Details
> The Revenge of the Baby-Sat	Bill Waterson	Details
> The Ultimate Hitchhikers Guide	Douglas N. Adams	Details

Book Details

isbn	0553293370
title	Foundation and Empire
authorsName	Isaac Asimov
price	5.99
pageCount	282
coverImage	
quantityInStock	7

FIGURE 12.28 Rendering of the model of [Figure 12.27](#) when the first book is selected in the list.

Parameter binding may also be used to associate a form to another view component. For example, the form may provide a search field, and a list would present only the elements that satisfy some search criteria.

[Figure 12.29](#) shows an example where a search field is used to find the books of a given author. The user may type the author's name or part of it in the search field, and the results are shown in the *Books* list.

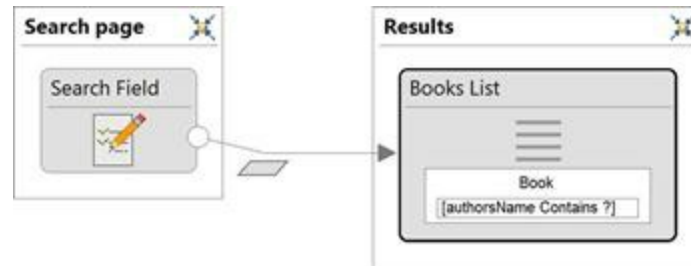


FIGURE 12.29 A flow that transports data from a form to a list.

The parallelogram that is next to the flow indicates that this flow has a parameter binding that is not the default object key.

In the rendering shown in [Figure 12.30](#), the user just typed “Clarke” in the search field, and the results produced are shown in [Figure 12.31](#).

The screenshot shows a web interface for a search page. At the top, there is a dark blue header. Below it, the text '> Search page' is displayed. The main content area is titled 'Search Field' and contains a text input field with the text 'Clarke' and a 'Go!' button.

FIGURE 12.30 Rendering of the model of [Figure 12.29](#) before the search is performed.

The screenshot shows the results page with a table of books. The table has the following columns: isbn, title, authorsName, price, pageCount, coverImage, and quantityInStock. The data rows are:

isbn	title	authorsName	price	pageCount	coverImage	quantityInStock
> 0553286587	Rama II	Arthur C. Clarke and Gentry Lee	6.99	466		12
> 055356871X	The Hammer of God	Arthur C. Clarke	6.99	240		2

FIGURE 12.31 Rendering of the model of [Figure 12.29](#) after the search is performed.

12.5.4 Multivalued parameter binding

It is possible to define flows from multiple-choice components such as checkable lists. In these cases, a parameter binding passes a set of values (checked elements keys, for example).

If a checkable list is linked to a multiple details view like in [Figure 12.32](#), the user may select a number of elements on the checkable list as in [Figure 12.33](#) and press the button on the bottom to navigate to the page that shows the details of the selected books only ([Figure 12.34](#)). The multiple details view has a key condition that states that only books whose OID belong to the set that is bound to the flow are shown.

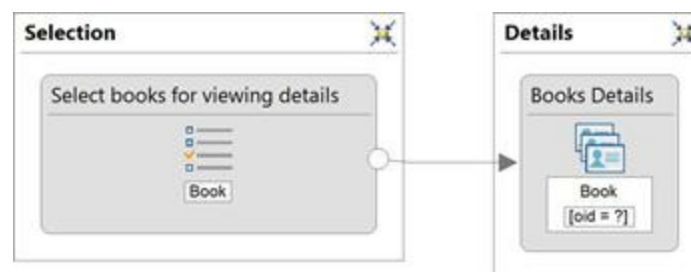


FIGURE 12.32 A flow with multivalued parameter binding.

The screenshot shows the 'Selection' view with a table of books. The table has two columns: 'title' and 'authorsName'. The books listed are: Foundation and Empire (Isaac Asimov), Rama II (Arthur C. Clarke and Gentry Lee), The Hammer of God (Arthur C. Clarke), The Long Dark Tea-Time of the Soul (Douglas N. Adams), Shave the Whales (Scott Adams), The Revenge of the Baby-Sat (Bill Waterson), and The Ultimate Hitchhikers Guide (Douglas N. Adams). The checkboxes for Rama II, Shave the Whales, and The Revenge of the Baby-Sat are checked. A 'Go to details' button is located at the bottom of the table.

title	authorsName
<input type="checkbox"/> Foundation and Empire	Isaac Asimov
<input checked="" type="checkbox"/> Rama II	Arthur C. Clarke and Gentry Lee
<input type="checkbox"/> The Hammer of God	Arthur C. Clarke
<input type="checkbox"/> The Long Dark Tea-Time of the Soul	Douglas N. Adams
<input checked="" type="checkbox"/> Shave the Whales	Scott Adams
<input checked="" type="checkbox"/> The Revenge of the Baby-Sat	Bill Waterson
<input type="checkbox"/> The Ultimate Hitchhikers Guide	Douglas N. Adams

FIGURE 12.33 Rendering of the model in [Figure 12.32](#) before clicking on the button.

> Details

Books Details						
isbn	title	authorsName	price	pageCount	coverImage	quantityInStock
> 0553286587	Rama II	Arthur C. Clarke and Gentry Lee	6.99	466		12
> 0752208497	Shave the Whales	Scott Adams	6.99	128		0
> 0836218663	The Revenge of the Baby-Sat	Bill Waterson	8.95	127		21

FIGURE 12.34 Rendering of the model in [Figure 12.32](#) after clicking on the button.

12.6 Hypertext organization

View components, flows, and their organization inside pages constitute the most local interface models. However, it is also necessary to create wider models for an interface, including groups of interrelated pages, navigation regions, landmarks, and so on. This kind of model is described in this section and is referred to as *hypertext organization*.

12.6.1 Site views

A *site view* is a package that contains an entire application that may be delivered for users. It usually contains a group of *areas* (see next section) and *pages* for the application. [Figure 12.35](#) presents an example of a site view with a page (*Main*) and two areas (*Administrative* and *Customer*).

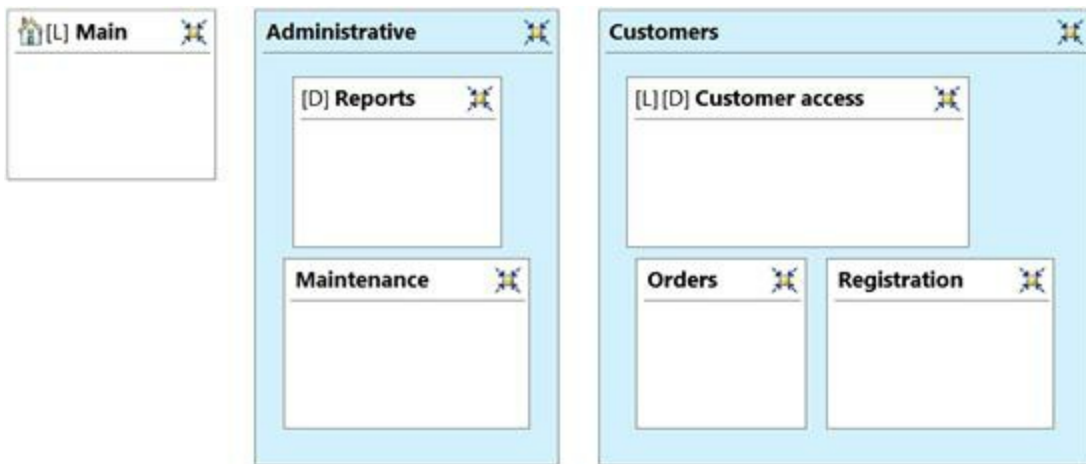


FIGURE 12.35 A site view.

This example has not yet been fully developed, though. Only the top-level structure is shown in [Figure 12.35](#).

12.6.2 Areas


A site view may be organized in *areas*. Areas and pages are referenced in IFML a

containers. Areas are groups of containers that have affinities, such as, for example, a set of functions related to a given user goal.

In the IFML model of [Figure 12.35](#), the areas appear as shadowed rectangles. Areas provide hierarchical organization to a site view.

12.6.3 Home, landmark, and default

In [Figure 12.35](#) some containers have special properties:

- [H] or  indicates the *homepage*. The homepage is the initial page of an application. When a user accesses the website she will access the home page by default. Areas do not have this property.
- [D] indicates the *default* page of an area. When a user navigates to a given area, she will land on the default page. Thus, the default page is a kind of “homepage” for a specific area. Areas do not have this property.
- [L] represents a *landmark* page or area. This means that the page or area is directly accessible from any other page included in the same area. Thus, it avoids the need to create lots of flows to pages that must be accessed from many places.

Only one homepage may exist for a given site view and only one default page for a given area. However, any number of landmark areas or pages may exist.

12.7 Web Interface Patterns

Most web interface patterns presented in this section are adapted from [Ceri et al. \(2003\)](#) and [Tidwell \(2005\)](#).

12.7.1 Cascade Index

A *cascade index* is a sequence of menus based on lists that eventually reaches a details view. For example, a user could begin by selecting a publisher. Then, the user could select a book from a publisher’s list, and finally access the book details. [Figure 12.36](#) illustrates that pattern.



FIGURE 12.36 Example of a cascade index.

In the diagram of [Figure 12.36](#), as the parameter bindings are default, the flows carry the OIDs of the selected elements. Those OIDs may be used by the conditional expressions at the destination view component to define which elements are to be shown.

The view component *Books List* uses a role-based conditional expression that shows only the books linked to the publisher whose OID is bound to the entry flow that comes from *Publishers List*.

Notice that the *Books List* has a conditional expression that is based on the role *PublisherToBook*, that is, only books linked to the publisher at the origin of the flow are shown by *Books List*.

The *Book* details view shows the complete information of the book selected in *Books List*. The selected book's OID is bound to the flow that comes from *Books List* to the details view. [Figures 12.37](#) to [12.39](#) show a sequence of rendered interfaces that illustrates that pattern. In [Figure 12.37](#) a publisher may be selected. In [Figure 12.38](#), after selecting the second publisher, one of its books may be selected. Finally in [Figure 12.39](#) the third book was selected and its details are shown.

> Select Publisher

Publishers List		
name		
>	Andrews and McMeel	Books
>	Bantam	Books
>	Boxtree	Books
>	Pocket Books	Books
>	Random House	Books

FIGURE 12.37 Rendering of *Publishers List*.

> Select Book

Books List		
title	authorsName	
>	Dirk Getly's Holistic Detective Agency	Douglas N. Adams Details
>	Foundation and Empire	Isaac Asimov Details
>	The Hammer of God	Arthur C. Clarke Details

FIGURE 12.38 Rendering of *Books List*.



FIGURE 12.39 Rendering of *Book* details.

One variant of this pattern consists of showing some data from the object selected in one of the intermediary lists. [Figure 12.40](#) illustrates that situation.

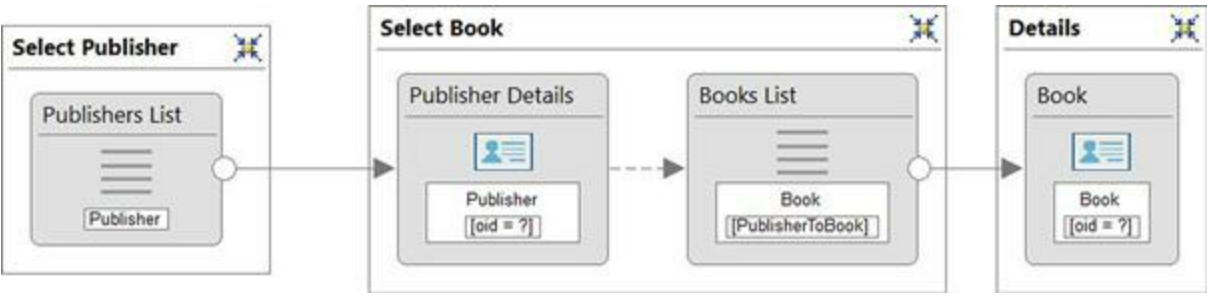


FIGURE 12.40 Cascade index with intermediary data presentation.

There, when a publisher is selected, the interface shows the publisher’s details and a list of books for further selection. Notice that in the intermediary page *Publisher Details* is connected to *Books List* by a data flow. It is not necessary for the user to navigate from *Publisher Details* to its list of books: both view components are shown at the same time when the *Select Book* page is accessed. [Figure 12.41](#) shows how the intermediary page could be rendered.



FIGURE 12.41 Rendering of the intermediary page of a cascade index with intermediary data.

12.7.2 Filtered index

In some cases it may be necessary to show a list with selected elements. For example, if the bookstore has thousands of books in its catalog, it would not be feasible for a user to simply browse for a book in that list. A filtered list could be used instead, for example, asking the user to provide a keyword in order to show a reduced book list.

This pattern is accomplished by a connection between a form, a list, and a details component, as shown in [Figure 12.42](#).

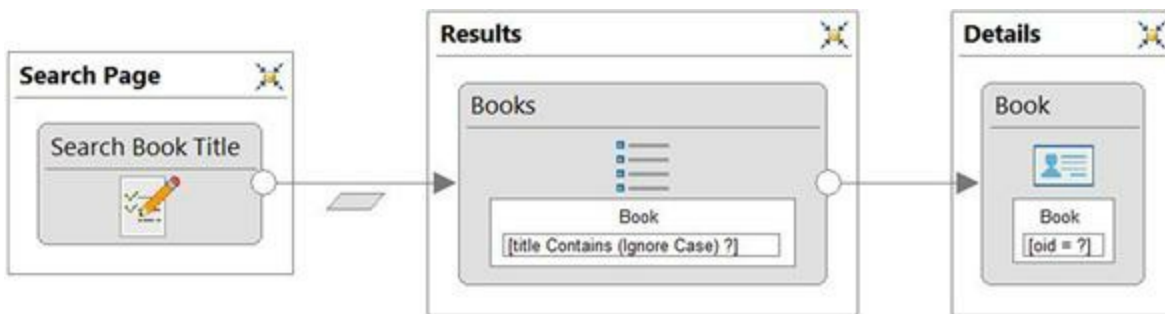


FIGURE 12.42 An example of a filtered index.

As the results are shown by a list (not a *simple* list), a scroll block factor may be defined as well as the possibility of dynamically sorting the results. Furthermore, a maximum number of results may be defined for this list so that unnecessarily large lists are not presented.

12.7.3 Guided tour

A *guided tour* is a pattern that allows one to visualize details of a set of objects one at a time by using scroll operations. It is implemented by using a scroller component with *blockFactor* 1 linked to a details view, as seen in [Figure 12.43](#).

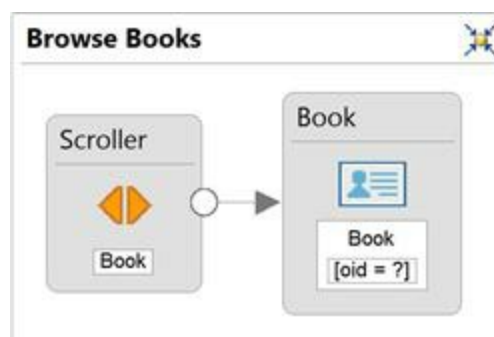


FIGURE 12.43 Specification of a guided tour.

Figure 12.44 shows a rendering of that specification where the fifth of eight books was selected by the user.

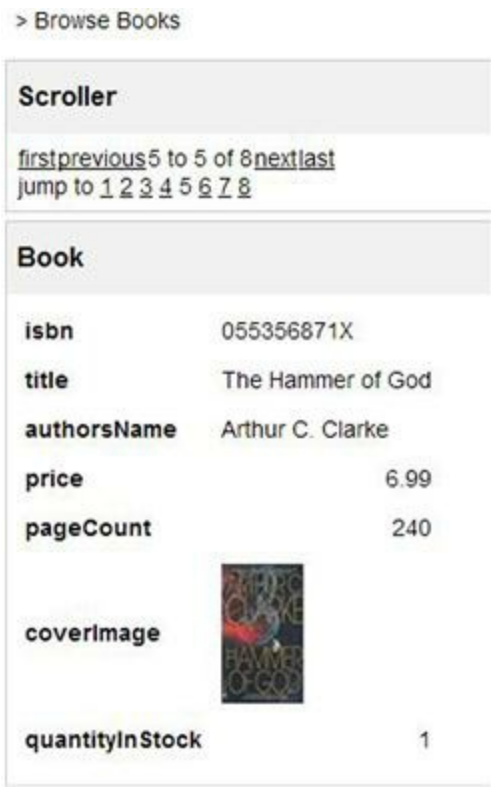


FIGURE 12.44 Rendering of a guided tour.

12.7.4 Viewpoints

Sometimes, it may be interesting to present different faces of certain objects in different moments. For example, summarized data about a book may be presented by default, but that data may be expanded or contracted again at the user's will.

In order to implement the viewpoints pattern, two or more details components may be defined for the same class, with different sets of attributes. Flows may be defined between the different components to allow the user to change from one view to another, as shown in Figure 12.45.

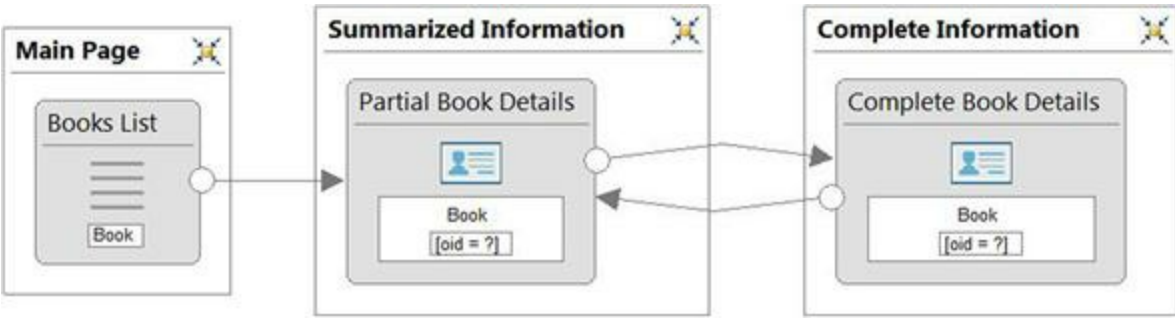


FIGURE 12.45 Specification of viewpoints.

Figure 12.46 shows the appearance of the summarized viewpoint and Figure 12.47 shows the complete information viewpoint.

> Summarized Information

Partial Book Details	
title	Dirk Getly's Holistic Detective Agency
authorsName	Douglas N. Adams
price	6.99
more details	

FIGURE 12.46 Summarized viewpoint.

> Complete Information


Complete Book Details	
isbn	0671746723
title	Dirk Getly's Holistic Detective Agency
authorsName	Douglas N. Adams
price	6.99
pageCount	306
coverImage	
quantityInStock	12
less details	

FIGURE 12.47 Complete information viewpoint.

12.7.5 Overview plus detail

The *overview plus detail* pattern is very common in applications such as email browsers, but it can be applied to a variety of other situations. The idea is to have in the same page a scrolling list of elements and the details view of one selected element in a region next to the list.

Figure 12.48 shows an IFML definition that uses this pattern with a list with block factor 3 and a details view component linked to it by a flow.

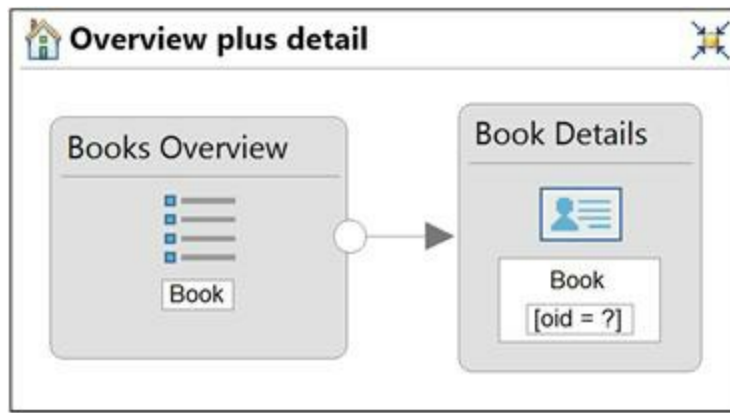


FIGURE 12.48 Specification of an overview plus detail pattern.

Figure 12.49 shows a rendering for the page specified in Figure 12.48.

> Overview plus detail

Books Overview

[first](#)
[previous](#)
4 to 6 of 8
[next](#)
[last](#)
[jump to 1](#)
[2](#)
[3](#)

	title	authorsName	price	
>	The Long Dark Tea-Time of the Soul	Douglas N. Adams	6.99	show detail below
>	Rama II	Arthur C. Clarke and Gentry Lee	6.99	show detail below
>	The Hammer of God	Arthur C. Clarke	6.99	show detail below

Book Details


isbn 0553286587
title Rama II
authorsName Arthur C. Clarke and Gentry Lee
price 6.99
pageCount 466
coverImage 
quantityInStock 2

FIGURE 12.49 Example of rendering of a page using the overview plus detail pattern.

12.7.6 Top-level navigation

Top-level navigation is a frequent pattern in web pages where a top-level menu allows access to different areas of the site. The example in Figure 12.50 illustrates top-level navigation being defined by a set of landmark areas, each containing one page (they could contain more pages if needed).

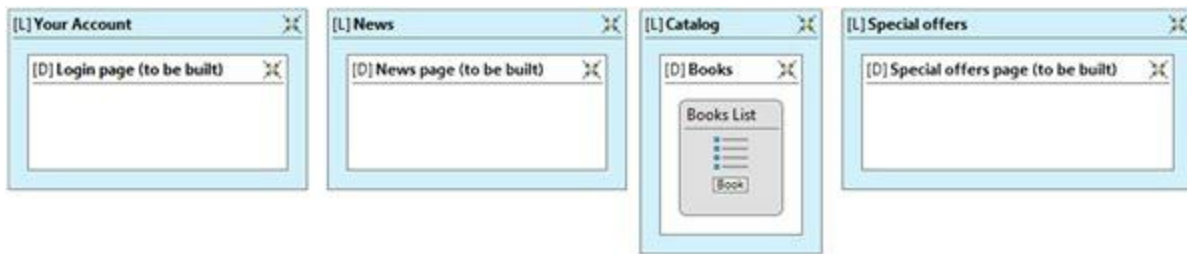


FIGURE 12.50 Example of specification of top-level navigation.

For the example, only the third area was partially defined. A rendering of that model is shown in [Figure 12.51](#) where the third area, “Catalog,” is selected from the top menu showing the default page *Books*.

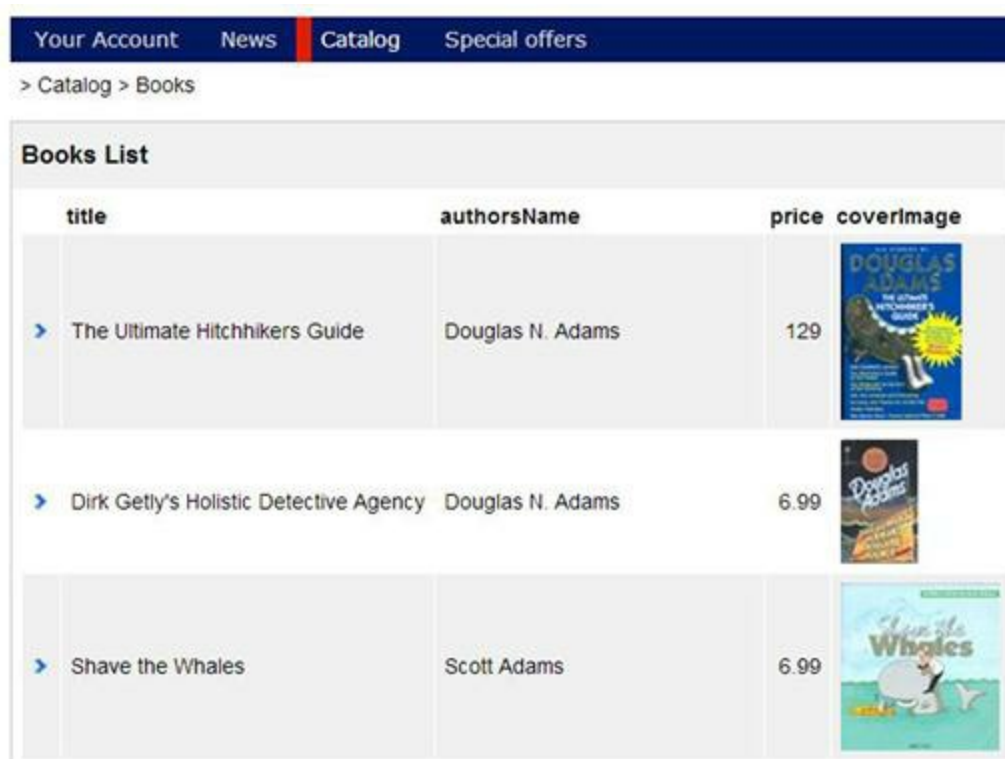


FIGURE 12.51 Example of rendering of a page using top-level navigation.

A variation of this pattern is menu-based top level navigation. This may be accomplished by adding a set of landmark pages to each of the top landmark areas, as shown in [Figure 12.52](#).

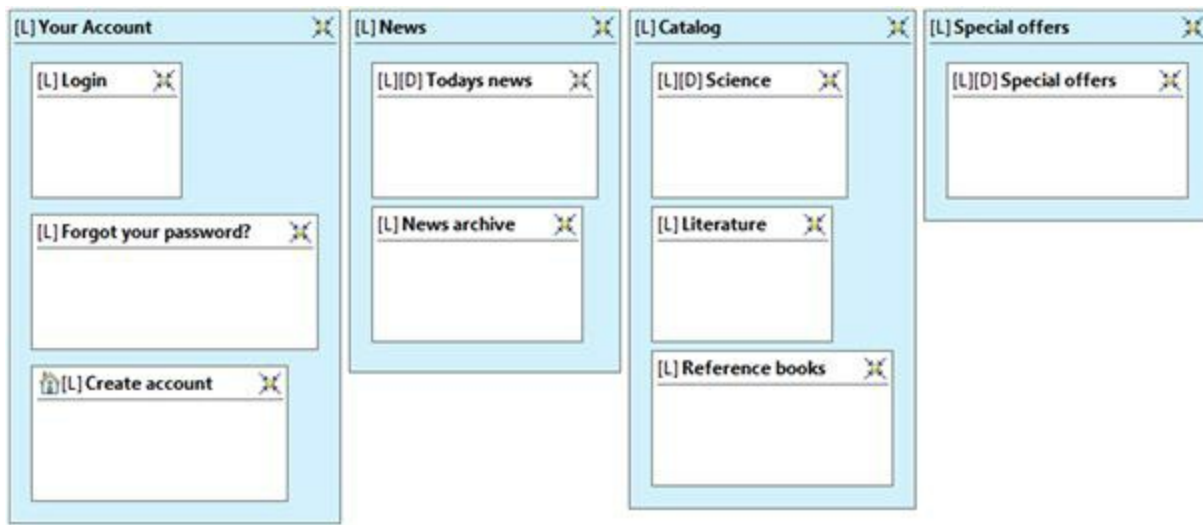


FIGURE 12.52 Example of specification of menu-based top-level navigation.

Figure 12.53 shows the rendering when the mouse is positioned over the menu item “Catalog.”

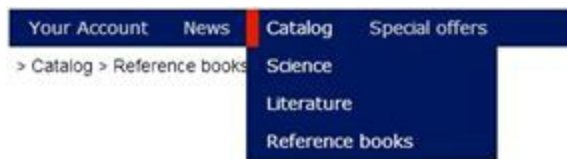


FIGURE 12.53 Example of rendering of a page using menu-based top-level navigation.

12.8 Modeling operations in the interface

IFML allows basic operations to be modeled by using *operation components* that allow for the creation, deletion, update, connection, disconnection, and reconnection⁶ of instances. These components perform most of the basic operations that were presented in Section 8.5. Figure 12.54 shows the graphical representation of those operations.



FIGURE 12.54 Operation components: create, delete, modify, connect, disconnect, and reconnect.

The operation components do not hold or render information; they just *process* information. Therefore, they are represented graphically outside the pages.

Every operation component should have at least one input flow. Sometimes all the information the operation needs is obtained from a single view component, but sometimes more than one view component may be necessary to supply all the data

needed by an operation.

Operation components have special output flows. Two different kinds of output flows exist for operations:

- *OK flow*, which defines where the focus goes when the operation is successfully performed for every object.
- *KO flow*, which defines where the focus goes when the operation fails for at least one of the objects affected by it.

The operation components are based on classes or associations that are their data sources. *Create*, *delete*, and *update* operations are defined over a single class, while *connect*, *disconnect*, and *reconnect* are defined over an association, and therefore they have a *source* and a *target* class.

For example, a delete operation could be defined over the *Book* class as in [Figure 12.54](#); this operation would be responsible for deleting instances of *Book*.

A connect operation, on the other hand, would be defined over an association such as *BookToPublisher*, as shown in [Figure 12.54](#). Therefore, it would be responsible for connecting (adding links between) instances of the *Book* class (source) and the *Publisher* class (target).

12.8.1 Create operation

The *create* operation creates new instances of a given class. A navigation or data flow may be used to provide initialization data for the new instance. Usually, the initialization data are obtained from a form based on the same class. A flow from the form to the create operation passes the values for all necessary attributes except for the OID, which is generated automatically by the create operation. For example, [Figure 12.55](#) shows a structure for creating instances of *Book*.

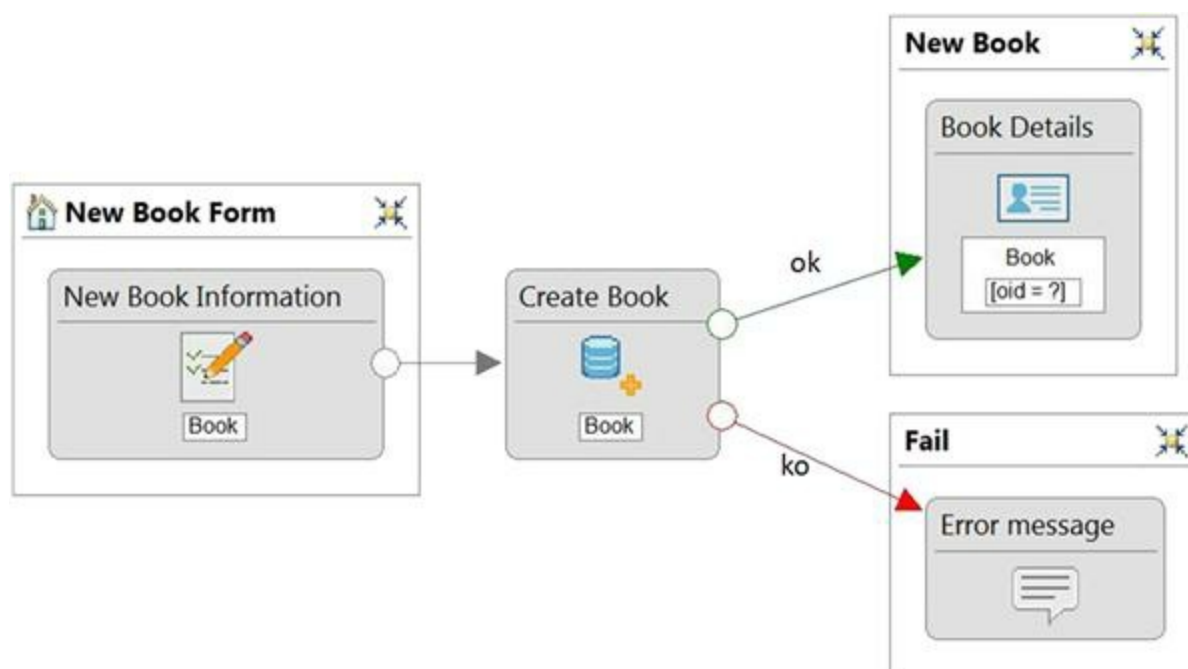



FIGURE 12.55 Example of application of a *create* operation.

Figure 12.56 shows the rendering of the form that is the origin of the flow to the *create* operation. The flow is rendered as the *Save* button.



The image shows a web form titled "New Book Information". It contains the following fields and controls:

- isbn**: A text input field.
- title**: A text input field.
- authorsName**: A text input field.
- price**: A text input field.
- pageCount**: A text input field.
- coverImage**: A file upload control with a green "Choose File" button and the text "No File Selected".
- quantityInStock**: A text input field.
- Save**: A grey button at the bottom of the form.

FIGURE 12.56 Rendering of a form with a flow to a create operation.

As we can see, the user may type in the form the initialization values for the new book. The flow from the form binds the data and takes the data to the create operation. The create operation then creates the new instance and defines its attributes with the values received from the flow.

If the operation is successful, then the OK flow is followed to a details component that shows the newly created book. The OK flow binds automatically the OID of the new book so that the conditional expression at the details may be used to define which book to present.

If the operation fails, the control goes to an error page following the KO flow.

12.8.2 Delete operation

The *delete operation* may be performed over one or more objects that satisfy the conditional expression defined for it.

The delete operation may be used in conjunction with a list (any kind of list), where the user may select the object(s) to be deleted. Figure 12.57 shows an example where the user selects a book to delete from a simple list. Notice that the selector of the delete operation is based on the book's OID, which is received as a parameter from the input flow.

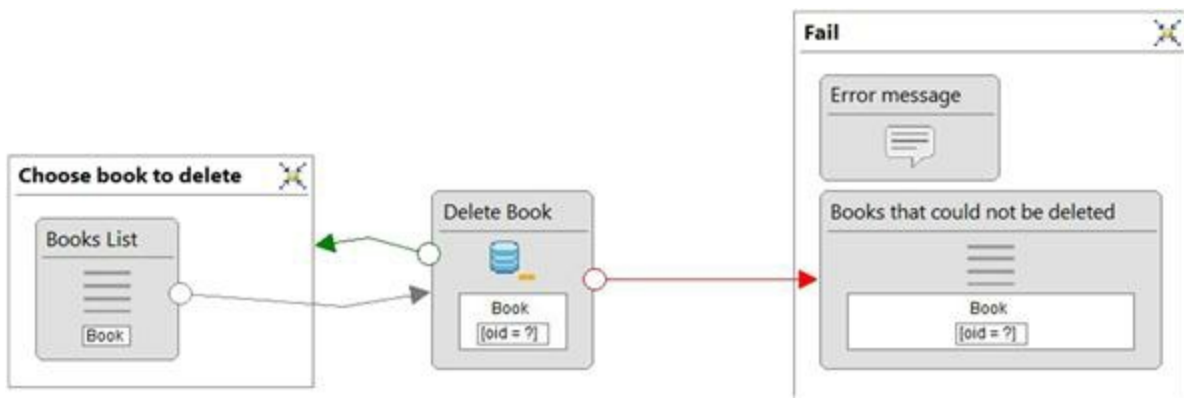


FIGURE 12.57 A delete operation being activated from a list.

This page may be rendered as in [Figure 12.58](#), where the word “delete” corresponds to the flow name.

> Choose book to delete

Books List		
title	authorsName	
> Dirk Gently's Holistic Detective Agency	Douglas N. Adams	delete
> Foundation and Empire	Isaac Asimov	delete
> Rama II	Arthur C. Clarke and Gentry Lee	delete
> Shave the Whales	Scott Adams	delete
> The Hammer of God	Arthur C. Clarke	delete
> The Long Dark Tea-Time of the Soul	Douglas N. Adams	delete
> The Revenge of the Baby-Sat	Bill Waterson	delete
> The Ultimate Hitchhikers Guide	Douglas N. Adams	delete

FIGURE 12.58 Rendering of a simple list associated to a delete operation by a flow named “delete.”

When all objects that qualify to be deleted are really deleted, the OK flow is followed. When at least one object cannot be deleted, the KO flow is followed and passes as a parameter binding the OIDs of the objects that could not be deleted. Thus, the KO flow could, for example, lead to a list or a multiple details view where the objects that could not be removed are shown, as seen in [Figure 12.57](#).

12.8.3 Update operation

An *update operation* contains a selector for one or more objects of the same class and a set of assignments for updating the attributes of those objects. Usually, new values for attributes are received by normal navigation or data flows.

[Figure 12.59](#) shows how to use an update operation to change the price of an existing book.

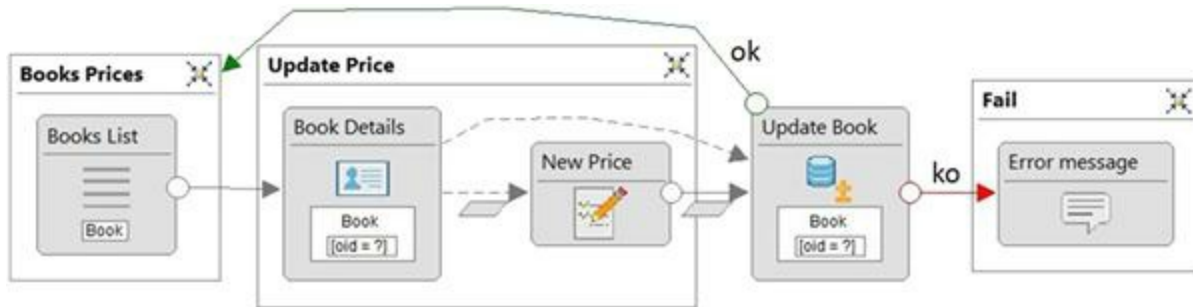


FIGURE 12.59 An update operation being used to change the price of a book.

On the *Book Prices* page, a book is selected from a list (Figure 12.60). Then a new page (*Update Price*) shows details of the book and a field preloaded with the old price. The user may change that value and update it by pressing “Save” (Figure 12.61).

> Books Prices

Books List			
	title	authorsName	price
>	The Ultimate Hitchhikers Guide	Douglas N. Adams	1.29 update price
>	Dirk Getty's Holistic Detective Agency	Douglas N. Adams	6.99 update price
>	Shave the Whales	Scott Adams	6.99 update price
>	The Long Dark Tea-Time of the Soul	Douglas N. Adams	6.99 update price
>	Rama II	Arthur C. Clarke and Gentry Lee	6.99 update price
>	The Hammer of God	Arthur C. Clarke	6.99 update price
>	Foundation and Empire	Isaac Asimov	5.99 update price
>	The Revenge of the Baby-Sat	Bill Waterson	8.95 update price

FIGURE 12.60 Rendering of the *Book Prices* page.

> Update Price

Book Details

title The Revenge of the Baby-Sat

authorsName Bill Waterson

New Price

New Price

FIGURE 12.61 Rendering of the *Update Price* page.

The old price is loaded in the *New Price* field by a data flow from *Book Details*. The

update operation is activated by the navigation flow that is rendered as the *Save* button on the *Update Price* page. That navigation flow has a parameter binding that passes the new price of the book from the respective field in the *New Price* form to the update operation. Other attributes of the book, including its OID, are received by a data flow from *Book Details*.

In [Figure 12.59](#), the OK flow from the update operation leads back to the list on the *Book Prices* page, where the new price may be visualized. The KO flow leads to an error window. Again, the OK flow is followed when all objects qualified by the conditional expression are changed. If at least one object cannot be updated, the KO flow is followed with the set of OIDs of the objects that could not be updated as binding parameters.

12.8.4 Connect, disconnect, and reconnect operations

The *connect*, *disconnect*, and *reconnect* operations have an association as a source of data. The connect and disconnect operations have two conditional expressions: one for the origin role and the other for the target role of the association. The reconnect operation may be understood as a combination of the former ones. It has three conditional expressions: one for the origin role, another for objects in the target role to be disconnected, and a third for objects in the target role to be connected to the source objects.

The connect, disconnect, and reconnect operations may add, remove, or replace links between sets of objects. If, for example, five objects satisfy the source conditional expression and three objects satisfy the target conditional expression, then a connect operation would create 15 links: each of the five origin objects will be linked to each of the three target objects.

The example in [Figure 12.62](#) shows how a link between a book and a publisher may be created. First of all, the link is mandatory from the book to the publisher. Thus, it is not possible simply to get a list of books and a list of publishers and choose which ones should be linked. A link between a book and a publisher must be created as soon as the book is created. Furthermore, that link is immutable and cannot change later.

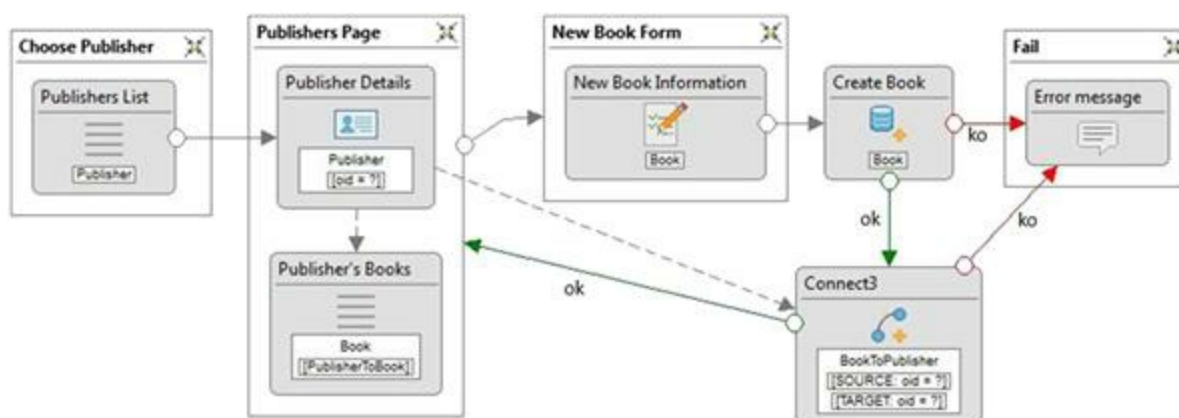


FIGURE 12.62 IFML model showing an interface in which books may be created and linked to publishers.

On the first page, *Choose Publisher*, there is a simple list of publishers. The user may choose one and be taken to *Publisher Page*. There, the publisher's details and a list of books already linked to it are shown. Notice that *Publisher Details* has a key condition and a flow from *Publishers List*, and therefore, it shows the publisher selected in the list. The *Publisher's Books* list has a data flow coming from *Publisher Details* and a role-based conditional expression so that only books that are linked to the publisher of *Publisher Details* are shown. [Figure 12.63](#) shows the *Publisher Page* rendered after the publisher "Bantam" was selected from the list.

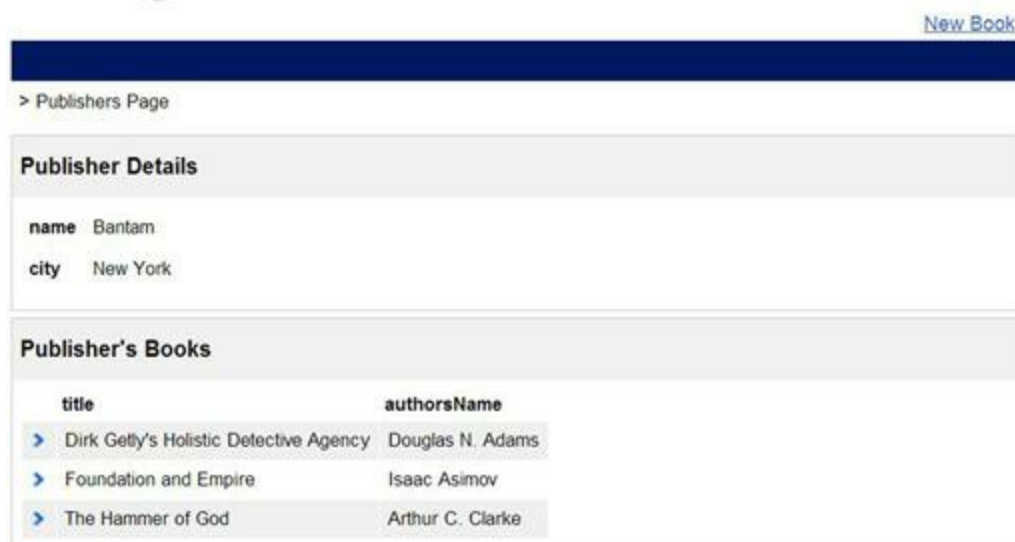


FIGURE 12.63 Rendering of the *Publisher Page*.

From the *Publisher Page* the user may navigate to the *New Book Form* by using a normal navigation flow between both pages, which is rendered as the *New Book* link. In this page the user may fill in the form with information about the new book, which will be automatically linked to the publisher presented in the *Publisher Page*. Initially the create operation creates the new instance of book. Its OK flow leads to a connect operation that is based on the association *BookToPublisher*. The flow that comes from the create operation has the new book key as a parameter binding; that key is used in the source conditional expression. The publisher is the target and it is obtained by a data flow coming from *Publisher Details*.

If the connect operation is successful, it returns focus to the *Publisher Page*, where the new book can be seen in the list of the current publisher's books. If the create or connect operation fail then focus goes to a page with an error message.

However, the design shown in [Figure 12.62](#) is not safe. If a book is created and the connect operation fails, that book would remain inconsistent regarding its definition. IFML allows a transaction to be defined as a group of operations. [Figure 12.64](#) shows an *operation group* being defined to include both the create and connect operations. As the operation group may be defined as a transaction ($[T]$), either all of the operations inside the group succeed or the whole group fails. In that case a single KO flow from the group is followed.

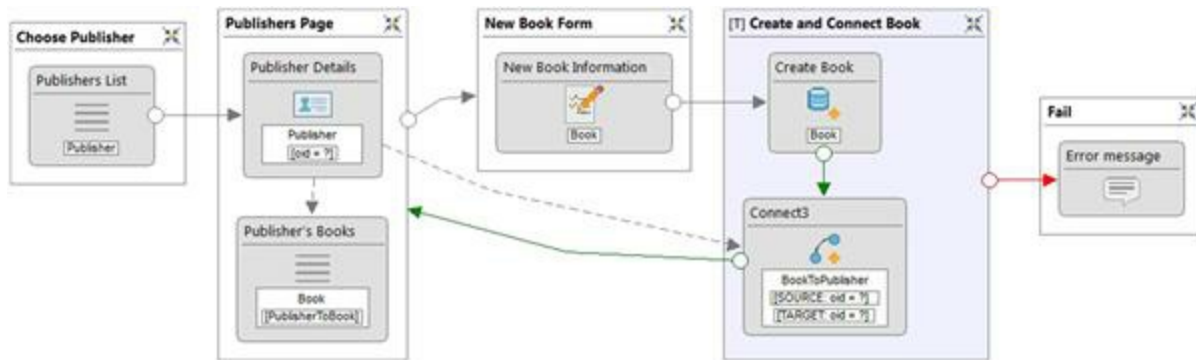


FIGURE 12.64 A safer design for Figure 12.62 using an operation group.

12.9 IFML models for CRUD operations

This section shows step-by-step how to implement a CRUD-style interface for managing information about publishers using IFML view components and operations.

The CRUD main page *Manage Publishers* presents a list of existing publishers. The creation of a new publisher is initiated by clicking on a link on the main page that gives access to a new page with a form. If the creation of a new publisher is successful, control returns to the main page, and the new publisher appears in the list. If there is an error, control goes to an error page, from which it is possible to return to the main page (Figure 12.65).

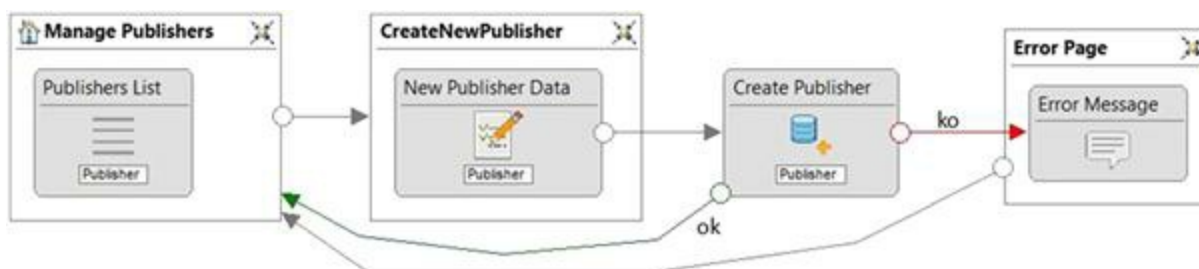


FIGURE 12.65 Defining an interface for “create”.

The second operation of a CRUD *retrieve*, is the query that allows the visualization of all attributes of a given publisher. This query may be modeled by a flow from the main list that leads to a page with a details view component, as shown in Figure 12.66.

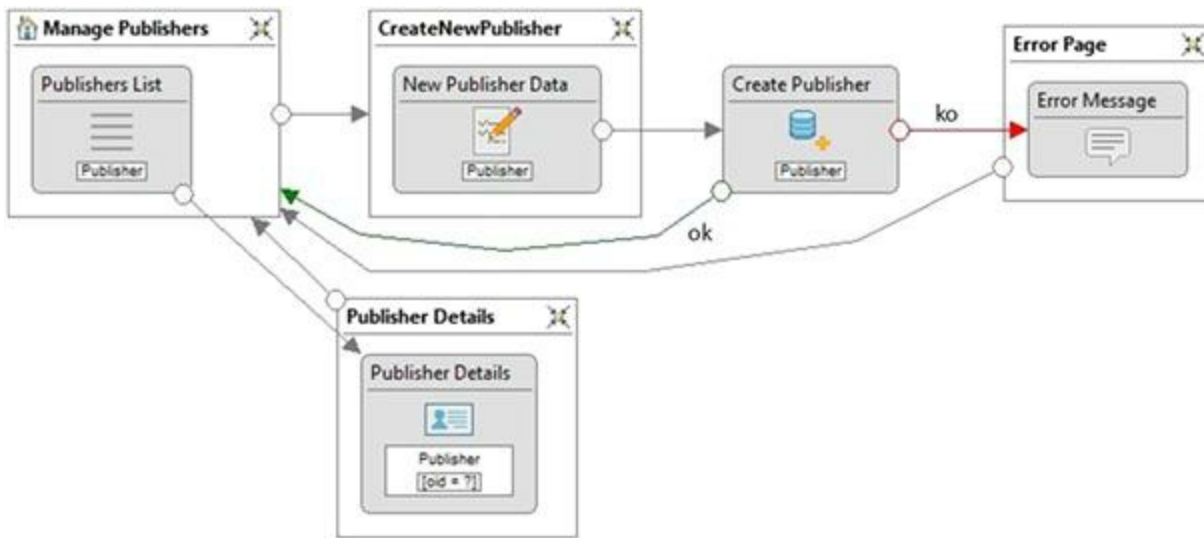


FIGURE 12.66 “Create” and “retrieve” defined.

The third operation, *update*, allows a publisher to be selected in the main list, and the attributes of the selected book are used to fill the fields of a form with *Publisher* as the source class. There they can be edited by the user and saved as shown in [Figure 12.67](#).

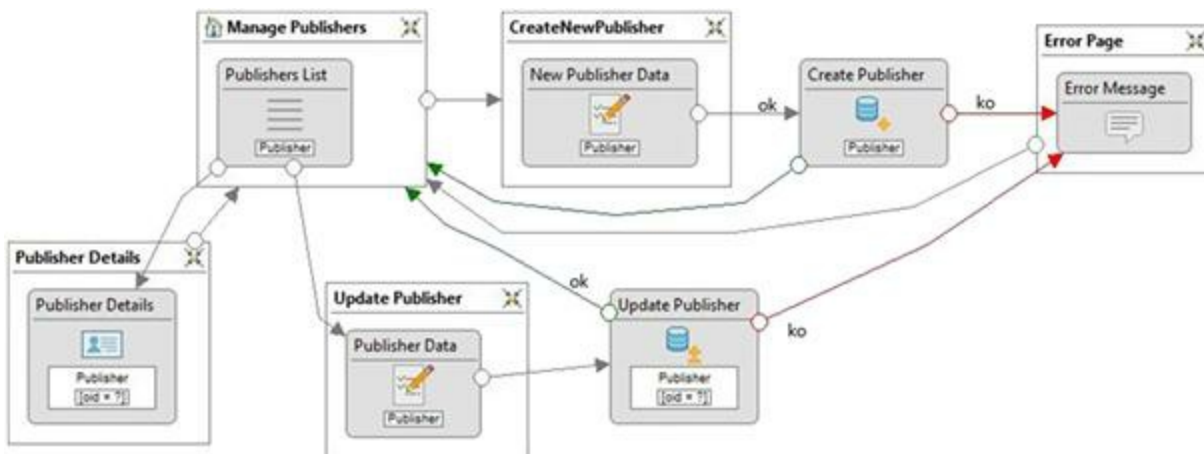


FIGURE 12.67 “Create,” “retrieve,” and “update” defined.

Finally, the operation for deleting a publisher is added, as shown in [Figure 12.68](#). A delete operation may be accessed from the main list.

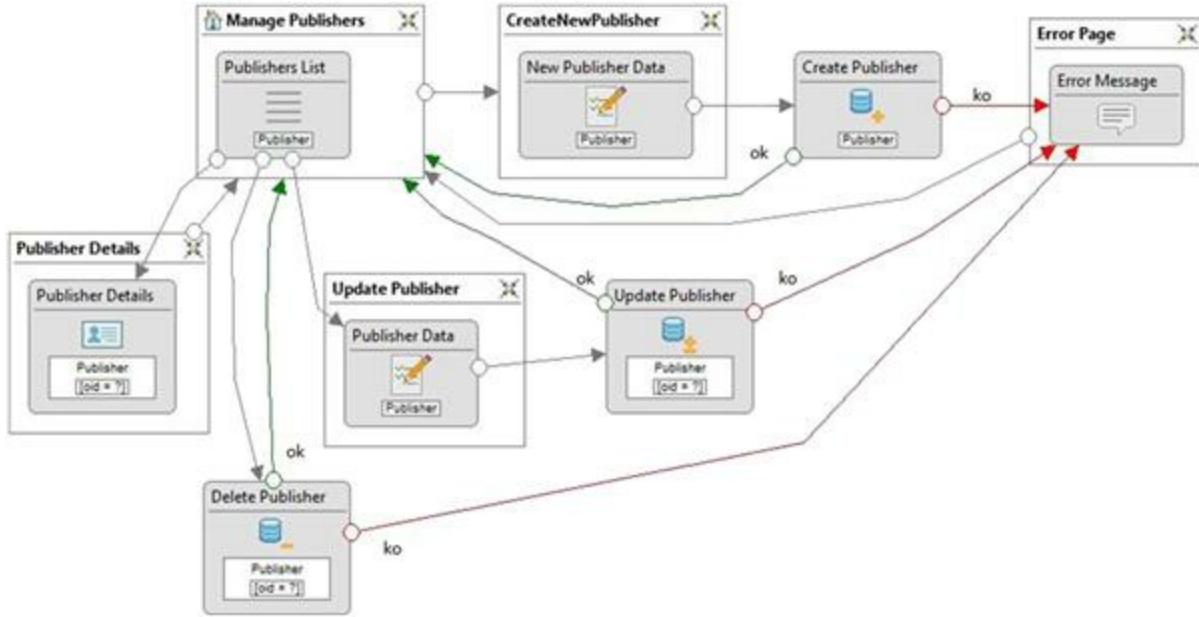


FIGURE 12.68 Complete CRUD interface specification.

Figure 12.69 shows the rendering of the main page of the CRUD of *Publisher*, as defined in Figure 12.68. At the top of the page, we can see the link for creating a new book, and just below is the main list, where the options for retrieving, updating, and deleting each publisher are available.

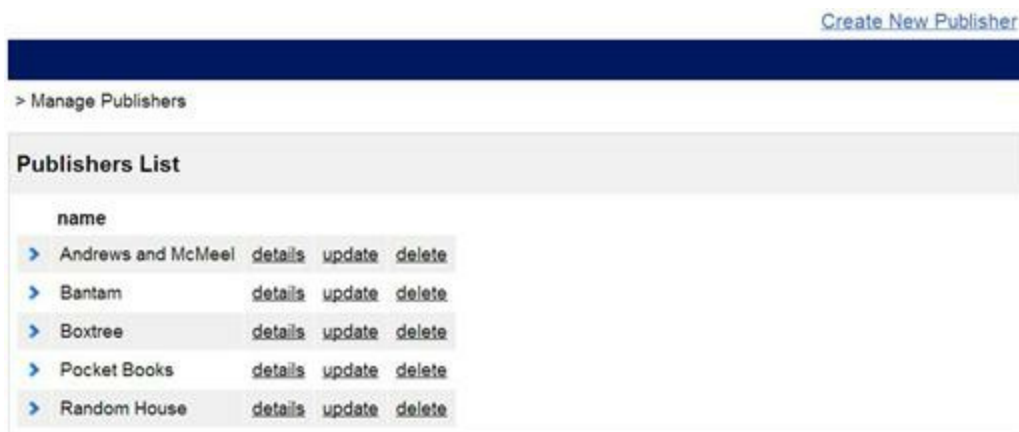


FIGURE 12.69 Main window for the CRUD for *Publisher*.

Figure 12.70 shows the rendering for the retrieve page, which is accessed by selecting the details option for the third publisher on the main page list. The *Back* link on the top of the page allows a user to return to the main page.



FIGURE 12.70 Rendering for the publisher retrieve page.

Figure 12.71 presents the publisher update page, which is accessed by pressing the *update* link on the main page list. This page is identical to the page for creating a new publisher except for the fact that it preloads data instead of being blank, and that as the *name* attribute is immutable, the respective field is defined as not modifiable.



FIGURE 12.71 Rendering for the publisher update page.

12.10 Use case interface modeling with IFML

As seen in the previous section, working with basic operations at this level may rapidly make the diagram too complex to be read easily. One option to deal with this complexity is to separate the interface project into different areas and deal with one at a time. This is possible at this time because use cases have already been explored and the team has a deeper understanding about the system structure and user needs, which allows them to design an acceptable interface.

The design of the customer interface may start with the use case diagram of Figure 3.11, part of which is presented in Figure 12.72.

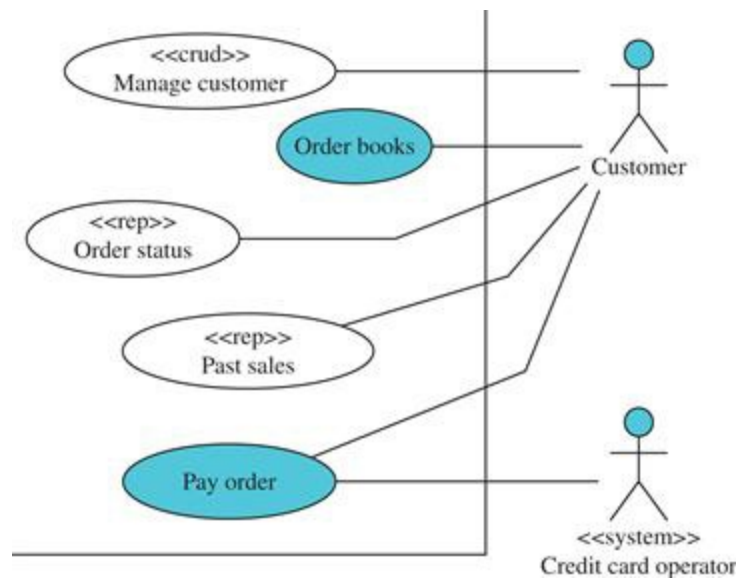


FIGURE 12.72 Partial use case diagram of Livir with the customer’s use cases.

Based on this use case diagram one possible interface design that could be prepared has three main areas:

- Shopping cart: orders and payment.
- My account: login and personal data.
- Reports.

This is still a very crude design, in the sense that no usability issues were raised in deciding on the best organization. However, it is effective. [Figure 12.73](#) shows the IFMI design of those areas.

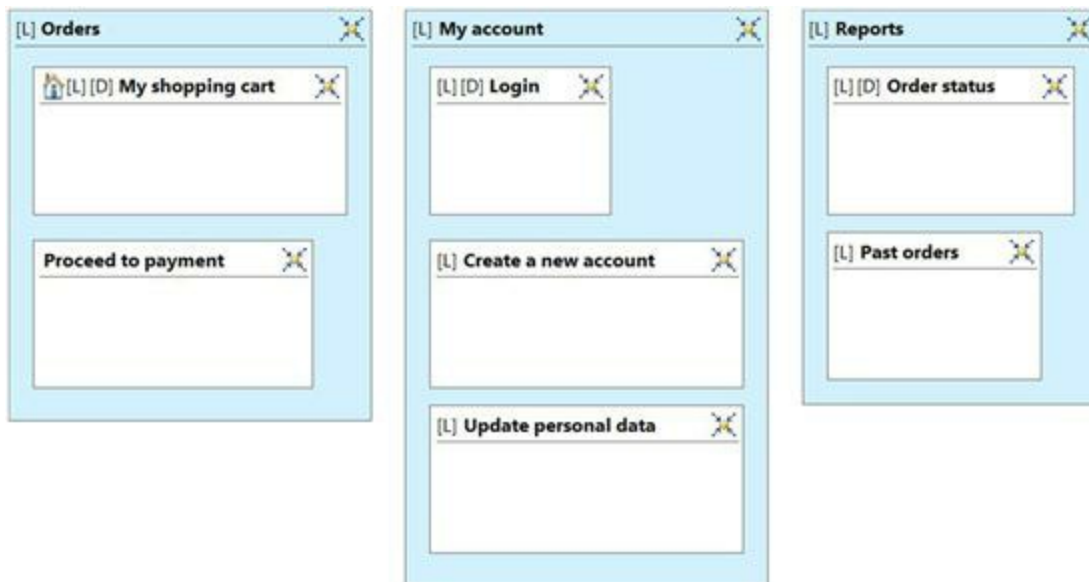


FIGURE 12.73 Initial high-level design for the customer site view of Livir.

Now let us take a look at the use case 01: *Order books*. Its system sequence diagram is depicted in [Figure 12.74](#).

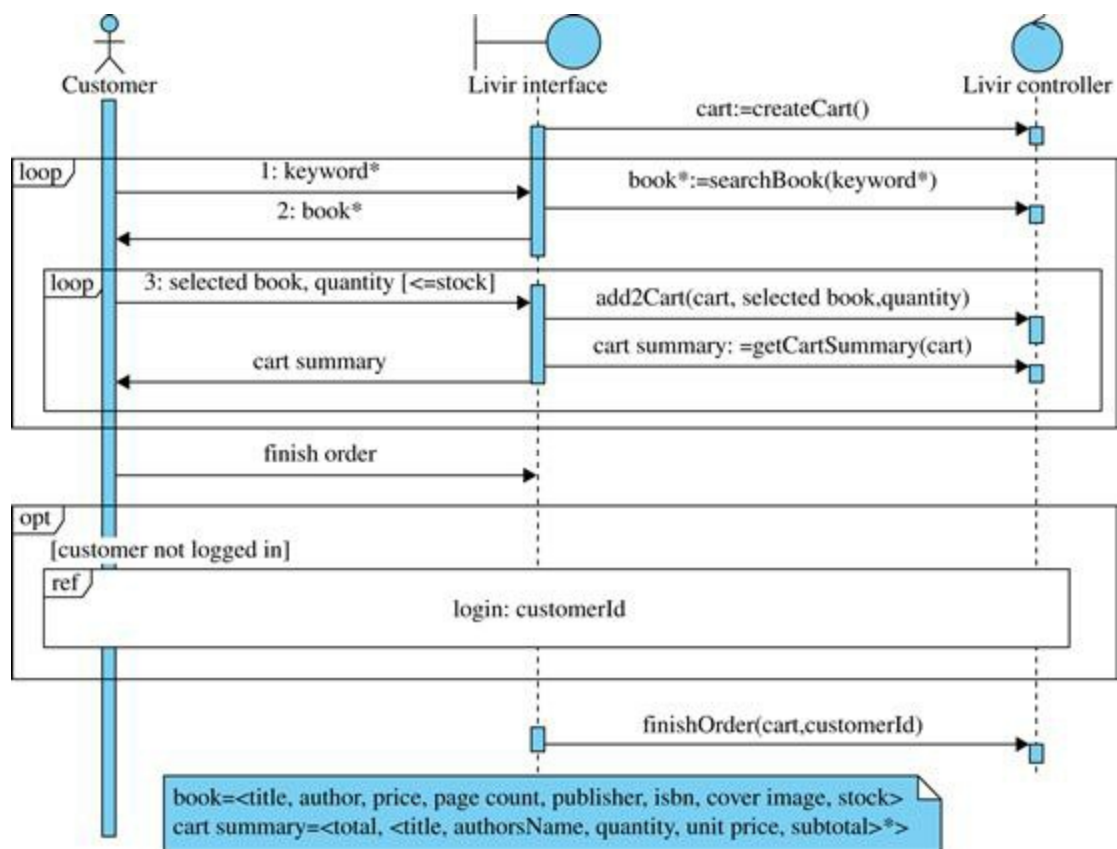


FIGURE 12.74 System sequence diagram for use case 01: *Order books*.

In order to prepare an interface design based on that sequence diagram, the team should look at the system operations, including their parameters and results:

- Any system operation, be it a command or query, must be activated somewhere in the interface. The activation can be obtained by clicking a button or a link, by selecting an item from a list, etc. Sometimes the same event may activate more than one system operation: if two or more system operations happen just after a system event (left side of the sequence diagram), then they are possibly activated by the same user action.
- Any parameter a system operation has must be obtained somewhere. Some parameters could be obtained from other sources (such as a clock, for example), but most parameters would be obtained in the interface. Thus, for most parameters there will be an interface view component (usually a form field) that allows the user to introduce the necessary data.
- Any result obtained by a system query must be presented somewhere. Some results may be hidden from the user in some cases, but usually they are presented. View components such as lists, details, and others may be used to show system query results.

In order to proceed with the interface design the recommendations above are followed and view components are created to hold all necessary parameters and results. System operation actions (shaped as hexagons) are added to the diagram as well. [Figure 12.75](#) presents the evolution of the interface design for the *My shopping cart* page, which implements part of the use case 01: *Order books*.

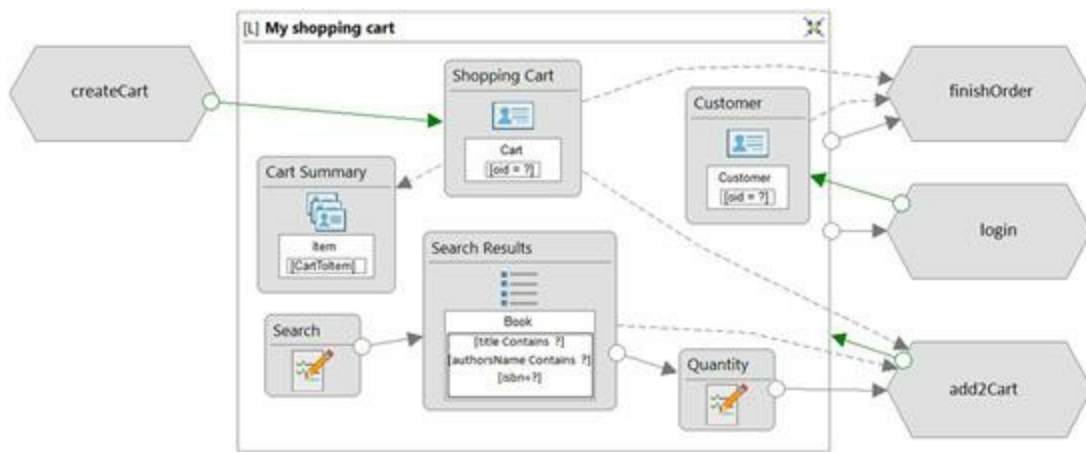


FIGURE 12.75 Further refinement of the interface design for the *My shopping cart* page.

The system operation *createCart* produces a new cart which is presented by the details *Shopping Cart*. The user can use the form *Search* to search for books, which are presented in list *Search Results*. There the user can select a book and define the desired quantity at the form *Quantity*. That activates the system operation *add2Cart*. That operation takes parameters from *Shopping Cart* (the *cartId*), *Search Results* (the *isbn*) and *Quantity* (the *quantity*). After a book is inserted in the cart it is automatically shown in the multiple details *Cart Summary*.

Before finishing the order, the customer should login if she has not yet done that. Finally, the order may be finished by performing the system operation *finishOrder*, which takes parameters from *Customer* (the *customerId*) and *Shopping Cart* (the *cartId*).

Thus, instead of using IFML operations for defining system operations, the team can use procedures defined externally as in [Figure 12.75](#). In this case, the team may implement them in their favorite programming language.

12.11 The process so far

	Inception	Elaboration
Business Modeling	<p>Build a general view of the system:</p> <ul style="list-style-type: none"> • Build a business use case diagram and determine the automation scope for the project. • Build preliminary activity diagrams for business use cases. • Build preliminary state machine diagrams for key business objects. 	
Requirements	<p>Prepare the system use case diagram (functional requirements):</p> <ul style="list-style-type: none"> • Identify the system actors from the business use case model. • Identify the system use cases from the business use case model, and the activity and state machine diagrams from business modeling. <p>Identify nonfunctional requirements as use case annotations:</p> <ul style="list-style-type: none"> • Identify the main business rules associated to use cases. • Identify the main quality issues associated to use cases. <p>Identify supplementary requirements.</p>	<p>Detail requirements by expanding use cases:</p> <ul style="list-style-type: none"> • Identify the main flow. • Identify the alternate flows: variants and exception handlers.
Analysis and Design	<p>Prepare the preliminary conceptual model by observing system use cases and the concepts needed by them.</p>	<p>Elaborate the system sequence diagrams:</p> <ul style="list-style-type: none"> • Represent the main flow of a use case as a system sequence diagram. • Represent the system commands and queries using stateful or stateless strategies. • Complete system sequence diagrams with alternate flows. <p>Refine the conceptual model:</p> <ul style="list-style-type: none"> • Identify concepts, attributes, and associations in the text of expanded use cases. • Detail attributes and associations with stereotypes, multiplicity, and constraints as needed.

(Continued)

	Inception	Elaboration
Implementation		<ul style="list-style-type: none"> • Organize the model by using inheritance, association classes, and temporal specifications. • Add invariants as needed. • Improve the conceptual model with the application of analysis patterns. <p>Write system operation contracts for commands and queries in system sequence diagrams:</p> <ul style="list-style-type: none"> • Identify preconditions and exceptions based on invalid parameters and complementary constraints. • Identify postconditions for commands and returns for queries. <p>Design the domain tier:</p> <ul style="list-style-type: none"> • Create a sequence or communication diagram for each system command contract. • Use those diagrams to decide which methods to implement in each class. • Inspect those diagrams to discover which associations can be unidirectional and define them as such. • Look at system query contracts and decide which delegate queries could be implemented; some of them could be derived attributes or derived associations. • Produce or refine the design class diagram. <p>Design the interface tier:</p> <ul style="list-style-type: none"> • Design the top-level interface organization based on areas and pages. • Refine pages, including view components for parameters of system operations and results for system queries. • Associate operation activation to interface events such as buttons and menus. <p>Generate code:</p> <ul style="list-style-type: none"> • Generate code for classes. • Generate code for attributes. • Generate code for associations. • Generate code for necessary basic methods such as get, set, add, remove, create, destroy, and others. • Generate code for system operations and delegate methods using dynamic models as reference.

(Continued)

	Inception	Elaboration
Test		Plan and execute tests: <ul style="list-style-type: none"> • Perform unit tests for methods and classes that were not generated automatically. • Perform system operation tests based on system operation contracts. • Perform use case tests based on use case scenarios.
Project Management	Estimate total effort, ideal calendar time, and average team size for the project. Estimate the duration and quantity of iterations for each phase. Prepare the phase plan and iteration plan for the first iteration.	

12.12 Questions

1. Chose at least three use cases of the Livir system (Table 4.7): a CRUD, a report, and a nonstereotyped use case. Create a complete interface model for them.
2. Research some sources concerning the MVC (Model-View-Controller) and MVP (Model-View-Presenter) design patterns. How many different definitions and variations for those models are you able to find? How do they differ from each other?

¹IFML is currently an ongoing standards project, and the first draft was published in March 2013. It may be accessed at <http://www.omg.org/spec/IFML/>.

²Available at www.IFML.org/.

³The first IFML-compliant tool.

⁴www.w3.org/Style/XSL/

⁵And *operations* as explained later.

⁶Replacing a link.

CHAPTER

E 13

Data Persistence

This chapter explains how to transform object-oriented models into relational databases and also how to load and save data automatically. Initially the object-relational mapping (ORM) technique, used to convert objects into relational database records, is explained. Then, we present a persistence mechanism based on virtual proxies, brokers, and caches.

Keywords

broker; commit; lazy load; object-relational mapping; ORM; relational database; rollback
virtual cache; virtual proxy

Key Topics in this Chapter

- Object-relational mapping (ORM)
- Virtual proxy
- Brokers
- Virtual caches

13.1 Introduction to data persistence

The availability of persistence mechanisms for commercial languages¹ has made database design much easier for many projects. With adequate tools, it is possible to automatically generate the persistence tier for a great number of information systems. For some critical and legacy systems, however, database adjustments may be still necessary in order to accommodate special features or to satisfy performance or security requirements.

Usually, object-oriented systems are implemented in object-oriented languages, but persistent² data storage is accomplished with *relational databases*. Although other techniques such as object-oriented databases ([Won, 1990](#)) and XML databases ([Bourret, 2010](#)) are also options for implementing permanent storage of data, *ORM, object-relational mapping*, is still the preferred approach.

The goal of this chapter is to explain what happens inside the persistence tier of a system when a persistence mechanism based on ORM is used. First of all, a good persistence mechanism requires domain and data storage to be separated into different tiers within the application. Remember that the interface tier is modeled with essential use cases and the domain tier is designed using the conceptual model and contracts as the basis. None of the aforementioned tiers addresses data storage or persistence. Persistence should be designed as a separate concern: a background system that will

keep data securely and permanently in its place without interfering on the domain or interface logic.

The persistence mechanism should assure that the objects are saved in a permanent memory device, and that they are loaded from there when necessary. Domain and interface logic should not be polluted by persistence concerns.

Object-oriented design provides lots of good concepts such as encapsulation, responsibility, and delegation that help designers deal with the complexities of the logic for accessing and transforming information. But when data must be stored in a more permanent way – disks and tapes, for example – the relational database is a good option because it is very efficient in terms of time performance and there are lots of optimization techniques to improve relational database operations. Thus, if the team wants to work in the best of two worlds (object and relational), the mapping between them must be understood.

The literature refers to the problem of *object-relational impedance mismatch* (Ireland, Keynes, Bowers, Newton, & Waugh, 2009), because most of the good features obtained by object-oriented design are lost when a flat relational database is used.

The implementation of a persistence mechanism minimizes that problem by using a set of predefined classes not belonging to the domain tier that take care of all logic involving domain objects being saved and retrieved from a relational database.

13.2 Object-relational mapping (ORM)

A complete and detailed design class diagram (DCD) allow for the automatic generation of a relational database structure that reflects in secondary memory³ the information that the objects represent in main memory. The following sections present some equivalence rules that should be observed when using ORM.

13.2.1 Classes and attributes

The first set of rules addresses classes and their attributes. Each persistent class of the DCD corresponds to a *relational table*. Each attribute is a *column* of the table, and each instance is a *line* or *record* of the table.

The stereotypes of some attributes such as `<<unique>>`, `<<optional>>`, and `<<immutable>>` affect the properties of the columns of the relational tables. Some of the features described here may not be implemented by some commercial database management systems. In that case, adjustments might be necessary in order to provide a safe implementation free from the object-relational impedance mismatch.

Attributes stereotyped with `<<unique>>` are represented as columns marked as *unique* or *uniq*, which cannot repeat values. However, even objects with no unique attribute have an *identity* that distinguishes them from other objects. Even objects with the same value for every single attribute may be differentiated by their identity. In the case of relational tables, this is accomplished by defining a *primary key* to each relational table. A primary key, just like a unique column, may not repeat elements. An element in a primary key

column may also not be null. Primary keys usually are simply sequential numbers generated automatically by the application in such a way that the same number is never generated twice for the same table. *Primary keys do not correspond to any of the attributes of an object*: they correspond to the object identity.

Figure e13.1 shows the class that is the basis for the following examples, and Table e13.1 shows the equivalent relational table with three instances of that class represented.

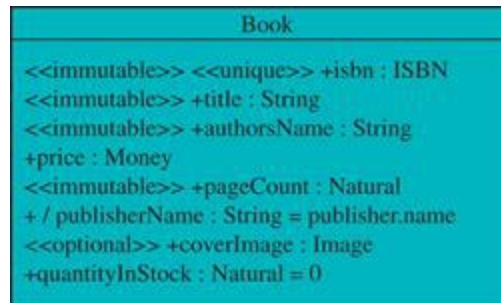


FIGURE E13.1 Reference class.

Table e13.1

Relational Table Equivalent to the Class of Figure e13.1

Table: Book							
PK,Uniq, Im,NN	Uniq,Im,NN	Im,NN	Im,NN	NN	Im,NN		NN
pkBook	isbn	title	authorsName	price	pageCount	cover Image	quantity InStock
10001	0553286587	Rama II	Arthur C. Clarke and Gentry Lee	6.99	466		2
10002	0553293370	Foundation and Empire	Isaac Asimov	5.99	282		3
10003	0671742515	The Long Dark Tea-Time of the Soul	Douglas N. Adams	6.99	307		21

Although other formats could be used to specify the constraints on the columns here they are presented as acronyms for quick reference:

- *Uniq*, or *unique*, means that the column cannot repeat values.
- *Im*, or *immutable*, means that the value in the column cannot be updated.
- *NN*, or *not null*, means that the column does not admit the null value. It is exactly the opposite of the *<<optional>>* stereotype used for design classes. It is used here because *not null* is a common constraint implemented in databases.
- *PK*, or *primary key*, means that the column is the primary key of the table. If a column

is *PK* it is necessarily immutable and not null. If only a *single column* is *PK* then it must be unique as well. However, if the *PK* is composite, spreading over more than one column, then each individual column may or may not be unique.

The first column in [Table e13.1](#) is *pkBook*. Notice that it does not correspond to any attribute of the reference class. Its value is artificially generated by a *number sequence generator* so that it is unique for the whole application or at least for that table. Some authors (Wieringa & Jonge, 1991) refer to it as a *surrogate key*, that is, a value that has no semantic meaning, and is unique systemwide, never reused, system generated, and not handled by the user or application.

The other columns correspond to the attributes of the reference class:

- *isbn* is a unique attribute. Therefore the equivalent column is unique, immutable, and not null.
- *title*, *authorsName*, and *pageCount* are normal attributes that are immutable and mandatory. Therefore the equivalent column is immutable and not null, but not unique.
- *price* and *quantityInStock* are normal attributes that may change but cannot be null. Therefore the equivalent column is not null only.
- *coverImage* is an optional attribute that may be updated. Therefore, the respective column has no constraint: it may be updated and may be null.
- *publisherName* is a derived attribute. Therefore it is not represented in the relational table.

If an attribute of the design class is marked with `<<transient>>` then it also should not appear in the relational table.

13.2.1.1 Number sequence generator

Most database management systems provide number sequence generators that generate numbers that never repeat. This is necessary to provide values for primary keys, especially when multiple users are producing new records at the same time. The number sequence generator must assure that different users would not produce the same number at the same time.

Number sequence generators may be associated directly to the column of the table that contains primary keys. Every time a new record is inserted in the table, a new number in the sequence is created.

The database designer may choose an initial value for the sequence as well as an increment. For example, beginning with 500 with an increment of 5 the sequence generated would be 500, 505, 510, 515, etc.

13.2.1.2 Index selection

By default, a relational table is just a set of records. Finding a given object at a table would require iterating over all elements until the desired element is found. For example, looking for a book given its ISBN would require an exhaustive search.

Databases usually provide, however, the possibility of indexing columns. An *indexed*

column has an auxiliary table that allows specific records to be found in almost constant time. For example, if the *isbn* column of the table is indexed, then when a book is searched based on its ISBN, no iteration is performed over the set of all records: the system simply would translate the value of the ISBN into a position in memory by using a hash function and retrieve the element from that position. If that is not the desired element, then it looks for the next, and so on until finding it. If the hash function is well implemented and the hash table has enough space to avoid collisions (two values being translated to the same hash value), then usually the desired element is really in the first place searched, or very close to it at least.

The use of indices improves query speed. However, it slows database updating because every time a record is updated, inserted, or deleted, the auxiliary table must be updated as well (Choenni, Blanken, & Chang, 1993). Furthermore, indices also require more storage space for accommodating the auxiliary tables.

A primary key is indexed by default. Other columns may be indexed if the designer chooses to do that. Given the restrictions mentioned before, creating other indices may be an advantage in the case of an attribute that is used as internal qualifier. In that case, finding the objects quickly may be crucial for the application's performance. Otherwise, indices should be avoided. For example, columns that are rarely used for searching purposes (for example, a book's page count) should not be indexed.

13.2.2 Associations

Generally, associations between classes (except transient associations that do not persist) correspond to *associative tables* in the relational model, that is, tables with a primary key composed by the primary keys values of the tables that represent the participating classes.

In this case, the primary key of the associative table is in fact composed of two (or more) columns. Each column may repeat values individually depending on the association multiplicity; but the pair (or tuple) of values that compound the primary key can never be repeated.

Depending on the multiplicity of the association roles, some rules must be observed. Many-to-many associations will have no individual unique restrictions. However one-to-many and one-to-one associations require primary keys in which one or both columns are unique. This is explained in further details in the following subsections.

13.2.2.1 Many-to-many associations

If the association is many to many with both role multiplicities defined as *, then there is no restriction on the columns that compose the primary key of the associative table. Figure e13.2 shows an example of a many-to-many association.



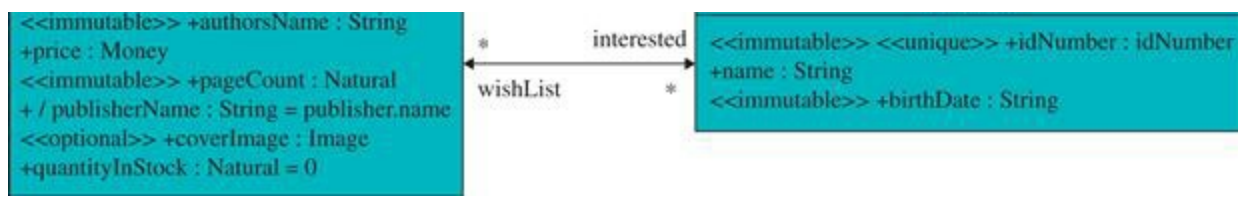


FIGURE E13.2 Many-to-many association.

Consider again the three books of [Table e13.1](#) and the three customers shown in [Table e13.2](#).

Table e13.2

Relational Table for the *Customer* Class

Table: Customer			
PK,Uniq,Im,NN	Uniq,Im,NN	NN	Im,NN
pkCustomer	idNumber	Name	birthDate
20001	987 - 65 - 4320	Abe	01/04/1970
20002	987 - 65 - 4329	Beth	02/23/1982
20003	987 - 65 - 4325	Charles	12/05/1979

Notice that in [Table e13.2](#) a customer has two codes: one is its *idNumber*, for example, a Social Security number. That value is known outside the information system. It is usually preformatted and although in normal conditions it does not change, nobody may prevent the government from changing all citizens' Social Security numbers if that is necessary. Therefore, though *idNumber* is unique for a customer, it does not qualify as a good primary key. The primary key for [Table e13.2](#) is *pkCustomer*, which is a number created automatically by the number sequence generator; it is unique and assured never to change, because it has no meaning outside the database.

[Table e13.3](#) shows an associative relational table that represents the links between some customers and some books desired by them. Notice that the PK spreads over two columns now. Each column contains a *foreign key* (FK), that is, a value that corresponds to the primary key of another table.

Table e13.3

Associative Table Representing a Many-to-Many Association

Table: interested_wishList	
PK	
NN	NN
fkCustomer	fkBook
20001	10001
20002	10001
20001	10003

In the associative table *interested_wishList*, the columns *fkCustomer* and *fkBook* together form the *composed primary key*. Both of them cannot be null, and pairs of *fkCustomer/fkBook* cannot be repeated. However, as the association is many to many, each individual column may repeat values, as seen in the table.

Table e13.3 shows that Abe (customer 20001) desires the books “Rama II” (10001) and “The Long Dark Tea-Time of the Soul” (10003). Beth (customer 20002) only desires “Rama II” (10001), and Charles (20003) has no wishes.

Instead of naming the associative table with the names of the classes (*Customer_Book*), it is preferable to name it with the names of the roles (*interested_wishList*), because more than one association may exist between two classes.

If the association is mandatory in one direction or both directions, then special considerations must be observed:

- If the association is mandatory on *one* side, then all instances from the *other* side must appear at least once in the associative table. If in the example of Figure e13.2, the role *interested* was mandatory (1..*), then each book should have at least one associated customer. Then each primary key value from the *Book* table should appear in the *interested_wishList* table at least once.
- If the association is mandatory on *both* sides, then all instances from both classes should appear at least once in the associative table. If in the example of Figure e13.2, both roles were mandatory (1..*), then each book from the *Book* table and each customer from the *Customer* table should appear at least once in the associative table.

More generally, considering that *A* has an association to *B*, and that the lower bound of the *B* role is *n* while the upper bound is *m* (multiplicity is *n..m*), the number of times that each instance of *A* must appear in the associative table is at least *n*, and no more than *m*. For example, if the multiplicity of role *B* is 2..5, then each instance of *A* must appear in the associative table at least twice and no more than five times. Unfortunately that constraint is not usually present in database management systems.

13.2.2.2 One-to-many associations

When the association is one to many, then the column on the *many* side must have a *unique* constraint. This means that the column may not repeat elements individually while the other column may repeat elements. The elements that cannot be repeated in

the associative table therefore can be linked to a single element of the other table; this constraint assures that the association is one to many. [Figure e13.3](#) shows an example of a one-to-many association.

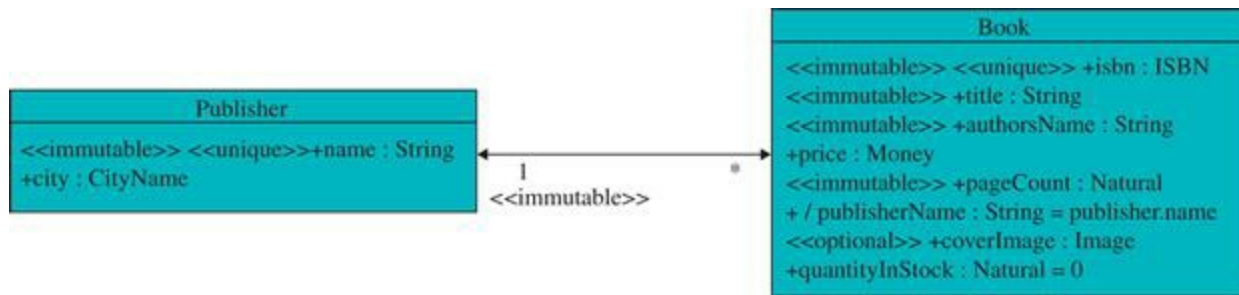


FIGURE E13.3 Example of a one-to-many association.

[Table e13.4](#) shows an associative table for the one-to-many association represented in [Figure e13.3](#). Notice that the *unique* constraint in the right column prevents the table from associating a book to more than one publisher.

Table e13.4
Associative Table Representing a One-to-Many Association

Table: publisher_book	
PK	
Im, NN	Uniq, NN
fkPublisher	fkBook
30002	10001
30001	10002
30002	10003

As seen in [Table e13.4](#), publisher 30001 has one book (10002) and publisher 30002 has two books (10001 and 10003). As books cannot be repeated in this table, no book can belong to more than one publisher.

It is also possible to represent associations from many to one as *foreign keys* in the table that represents the class at the *many* side of the association. For example, as each book may have only one publisher, then the association between book and publisher could be implemented as in [Table e13.5](#).

Table e13.5
Alternative Way to Implement a Many-to-One Association Without an Associative Table

Table: Book								
PK, Uniq, Im,NN	Uniq,Im, NN	Im,NN	Im,NN	NN	Im,NN		NN	Im,NN
pkBook	isbn	title	authors Name	price	page Count	cover Image	quantity InStock	fkpublisher
10001	0553286587	Rama II	Arthur C. Clarke and Gentry Lee	6.99	466		2	30002
10002	0553293370	Foundation and Empire	Isaac Asimov	5.99	282		3	30001
10003	0671742515	The Long Dark Tea- Time of the Soul	Douglas N. Adams	6.99	307		21	30002

The foreign key *publisher* is a direct reference to the primary key of the *Publisher* table. If the association is straight to 1, then the foreign key column must not be null. If the association is to 0..1, then the foreign key column should not have that constraint.

Although this approach is not as homogeneous as associative tables, it is usually preferred by designers because it avoids the need to implement a new table. One disadvantage is that in this case many-to-many associations are implemented as associative tables and many-to-one associations are implemented inside one of the original tables. Also, if the association is not mandatory (if its multiplicity is 0..1), and relatively few elements are associated, there would be lots of null values in the foreign key column, wasting storage space and degrading performance. However, if the association is straight to 1, this disadvantage does not apply.

13.2.2.3 One-to-one associations

One-to-one associations, mandatory or optional, require that the associative table have a *unique* constraint in both columns of the composed primary key in order to prevent any element on both sides from appearing more than once in the associative table. [Figure e13.4](#) shows an example of one-to-one association that is mandatory on one side and optional on the other.



FIGURE E13.4 Example of one-to-one association.

[Table e13.6](#) shows the relational table that implements the one-to-one association of [Figure e13.4](#).

Table e13.6

Associative Table Representing a One-to-One Association

Table: order_payment	
PK	
Im,Uniq,NN	Uniq,NN
fkOrder	fkPayment
50001	60001
50003	60002
50005	60003
50011	60004
50016	60005
50021	60006
50030	60007

As the role is mandatory for payments, all instances of *Payment* must appear in the associative table, but not all instances of *Order* must appear, because the role is not mandatory for them.

As in the case of many-to-one associations, one-to-one associations may also be implemented as foreign keys in one of the original tables. The foreign key column must necessarily be unique in that case. If the association role is also mandatory, then the foreign key column should not be null.

13.2.2.4 Ordered associations

An association with an ordered role (*sequence* or *ordered set*) may be implemented as an associative table with an extra column to represent the order of the element in the role's collection of elements. [Figure e13.5](#) shows an example with two situations: ordered set and sequence (a list in which elements may be repeated).

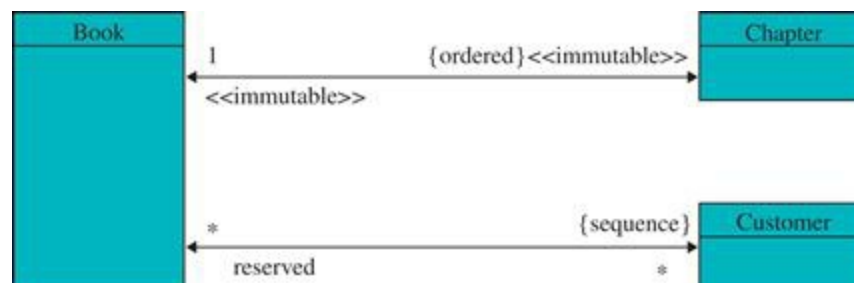


FIGURE E13.5 Example of ordered roles.

The difference between the relational implementation of an ordered set and a sequence is that in the case of the ordered set the *order* column must not be included in the composed primary key, as shown in [Table e13.7](#). However, in the case of a sequence (elements may be repeated), the *order* column must be part of the composed primary key, which in this case is composed of three columns ([Table e13.8](#)).

Table e13.7

Relational Table Representing an Ordered Set

Table: book_chapter		
PK		
Im,NN	Im,Uniq,NN	NN
fkBook	fkChapter	order
10001	130001	1
10001	130002	2
10001	130003	3
10002	130004	1
10002	130005	2
10003	130006	1
10003	130007	2

Table e13.8

Relational Table Representing a Sequence

Table: book_customer		
PK		
NN	NN	NN
fkBook	fkCustomer	order
10001	20001	1
10001	20003	2
10001	20002	3
10001	20001	4
10002	20003	1
10003	20001	1
10003	20002	2

In [Table e13.8](#), the fact that the *order* column is included in the primary key allows the same person to reserve the same book more than once (for example, 20001, Abe, has two reservations for book 10001, Rama II, in the first and fourth positions on the reservation list. A repetition like this would not be possible in [Table e13.7](#), because the primary key does not include the *order* column: the same pair *fkBook/fkChapter* cannot appear more than once in the table, regardless of its position.

In both cases, to prevent a given position to be occupied more than once, a *uniq* constraint spreading over the origin class and the position should be defined. In the case of [Table e13.7](#) and [Table e13.8](#) the pair *fkBook/order* should be unique.

13.2.2.5 Associations representing bags

In the case of *bags*, in which elements may be repeated but have no position, the usual solution is to add an extra column to the associative table with a counter for the number of times a given pair participates in the association. [Figure e13.6](#) shows an example of this kind of association.



FIGURE E13.6 Example of a bag.

[Table e13.9](#) shows the implementation of the associative table for the example in [Figure e13.6](#).

Table e13.9**Associative Table Representing a Bag**

Table: book_viewer		
PK		
NN	NN	NN
fkBook	fkCustomer	quantity
10002	20002	1
10001	20002	2
10002	20001	6
10003	20002	1

Table e13.9 specifies that Abe (20001) has viewed the book “Foundation and Empire” (10002) six times. Beth (20002) viewed “Rama II” (10001) twice, “Foundation and Empire” (10002) once, and “The Long Dark Tea-Time of the Soul” (10003) once. It is not necessary to represent in the table any pair whose quantity is zero; this is why Charles (20003) does not appear in the table: he has never viewed any book.

13.2.2.6 Qualified associations

In the case of a qualified association defined as a map (multiplicity 1 or 0..1) with an internal qualifier (the qualifier is an attribute of the qualified class), it is sufficient to implement the association as a regular one-to-many or many-to-many association depending on the multiplicity on the side of the qualifier, as explained in previous sections.

The only special care that the database designer must take in that case is to ensure that the column of the qualifier attribute is *unique* and *immutable*. It may be indexed if quick access to the records is necessary.

However, when the qualifier is *external*, it is necessary to add a third column to the associative table to allow for the representation of the qualifier. Figure e13.7 shows an example of that situation.

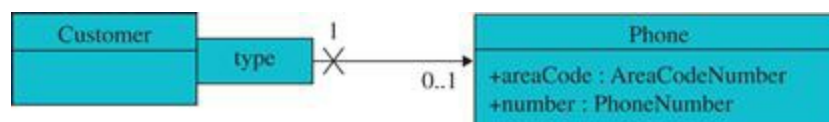
**FIGURE E13.7** Example of a map with an external qualifier.

Table e13.10 shows the implementation for the map defined in Figure e13.7. The associative table has a primary key that is composed only of the origin class key and the qualifier. The destination class is left out of the primary key. However, it must be marked as *unique* because, in the example, each phone has a single type.

Table e13.10

Associative Table Representing a Map With an External Qualifier

Table: customer_phone		
PK		
NN	NN	Uniq, NN
fkCustomer	type	fkPhone
20001	Home	70001
20001	Cellphone	70002
20002	Home	70003

If the external qualifier defines a partition (multiplicity *), as shown in Figure e13.8, it is implemented as shown in Table e13.10. But in the case of a partition with an external qualifier, the primary key must have three parts, including the origin and destination foreign keys as well as the qualifier. Also, as the origin role multiplicity is 1, the destination class column must be marked with *unique*. In the example shown in Table e13.11, this means that a book may not have more than one genre.

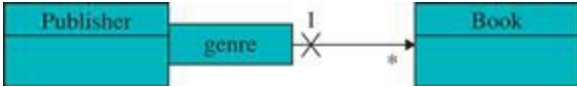


FIGURE E13.8 Example of a partition with an external qualifier.

Table e13.11

Associative Table Representing a Partition with an External Qualifier

Table: publisher_book		
PK		
NN	NN	Uniq, NN
fkPublisher	genre	fkBook
60001	sci-fi	10001
60001	sci-fi	10002
60002	humor	10003

If the role multiplicity at the origin were * (defining a relation where a book could have more than one genre), the implementation would basically be the same. The only difference is that the *unique* constraint in the destination column (*fkBook*) should not exist.

13.2.2.7 Association classes

An association with an association class is represented in two parts: a relational table for the association class with its attributes, and an associative table for the association with a reference to the association class table.

Figure e13.9 shows an association class, and Tables e13.12 and e13.13 implement its relational equivalent.

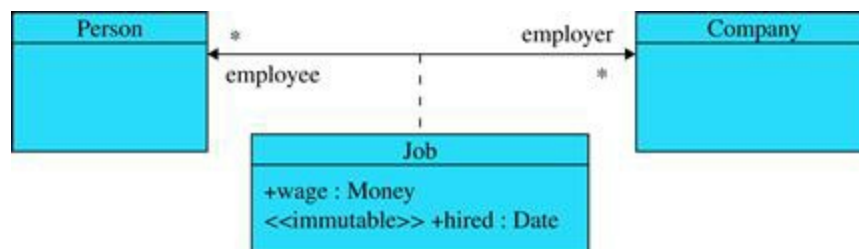


FIGURE E13.9 Example of an association class.

Table e13.12

Relational Representation of an Association with an Association Class – Part 1: The Association Class

Table: Job		
PK, NN, Im, Uniq	NN	NN, Im
pkJob	Wage	Hired
80001	1,500.00	02/15/2008
80002	1,200.00	03/01/1999
80003	2,000.00	04/16/2005
80004	900.00	01/17/2001

Table e13.13

Relational Representation of an Association with Association Class – Part 2: The Association

Table: employee_employer		
PK		
NN	NN	NN, Im, Uniq
fkPerson	fkCompany	fkJob
20001	70001	80001
20001	70005	80002
20002	70001	80003
20003	70002	80004

Thus, one way to represent associations with association classes is to use a table to represent the association class (Table e13.12), and an associative table (Table e13.13) with three columns: the primary keys of the participating classes (which forms the composite primary key of the association) and the primary key of the association class, which is not part of the composite primary key of the association, but must be immutable and unique in that table.

A further constraint is that all values for *pkJob* in the *Job* table must appear in the *fkJob* column of the *employee_employer* table.

13.2.2.8 *n*-ary associations

In the case of associations among three or more classes (n -ary), an associative table is defined in which the primary key is formed by the primary keys of all participating classes. [Figure e13.10](#) shows an example of a ternary association and [Table e13.14](#) shows its relational representation.

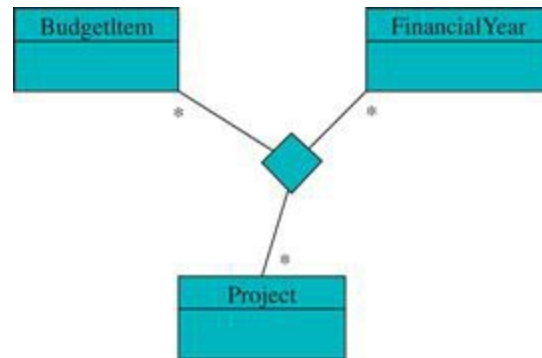


FIGURE E13.10 An example of a ternary association.

Table e13.14
Relational Equivalent of a Ternary Association

Table: budgetItem_financialYear_project		
PK		
NN	NN	NN
fkBudgetItem	fkFinancialYear	fkProject
90001	100001	110001
90001	100002	110002
90002	100001	110003

In [Table e13.14](#), pairs such as *fkBudgetItem/fkProject* may repeat elements (90001/100001, for instance). But the triple *fkBudgetItem/fkFinancialYear/fkProject* may never repeat.

13.2.2.9 Transient and façade controller associations

Transient associations are not represented in relational tables because, by their own definition, they exist only in primary memory, and it is not necessary neither desirable to persist them.

Some of the associations from the façade controller also do not need to persist. Associations from a controller to all instances of a class, which are mandatory for the

instances, do not need to be transformed into association tables, because they always repeat the same value for the controller and have all the elements of the other side. That information is already available in the primary key column of the table representing the conceptual class, and, therefore, that association table would be redundant. For example, in [Figure e13.11](#), the association between the controller and *Customer* with no explicit role name is mandatory for all customers. Thus, an associative table for representing that association would contain only the primary keys of all customers. As they are already represented in the *Customer* table, repeating them in a different table is unnecessary.



FIGURE E13.11 A mandatory and an optional association to a façade controller.

However, this observation is valid only if the association role is strictly 1 on the controller side. If the controller side has multiplicity 0..1, then the association should be represented separately. In [Figure e13.11](#) the *premium* association must be represented because not every customer belongs to it. Not every customer is a *premium* customer,⁴ and thus they must be listed somewhere. There are at least two choices for representing premium customers: adding a Boolean field to the *Customer* table indicating which customers are premium, or creating a single column table that contains the primary keys of premium customers.

The premium customer table does not have to include a column that represents the primary key of the controller (1) because the controller is not an entity (and therefore it has no primary key), and (2) because the controller is a singleton and even if a primary key is assigned to it, repeating the same value in all rows of the table would be unnecessary.

[Table e13.15](#) shows a possible implementation for this optional association.

Table e13.15

Implementation of an Optional Association from the Façade Controller

Table: premium

PK

NN

fkCustomer

20001

20002

According to [Table e13.15](#) only Abe (20001) and Bea (20002) are premium customers, and Charles (20003) is not.

13.2.3 Inheritance

Relational databases do not support inheritance directly. Mapping inheritance relations to relational databases is an issue that demands attention. There are many different approaches and this section discusses some of them. All examples in the following subsections are based on [Figure e13.12](#).

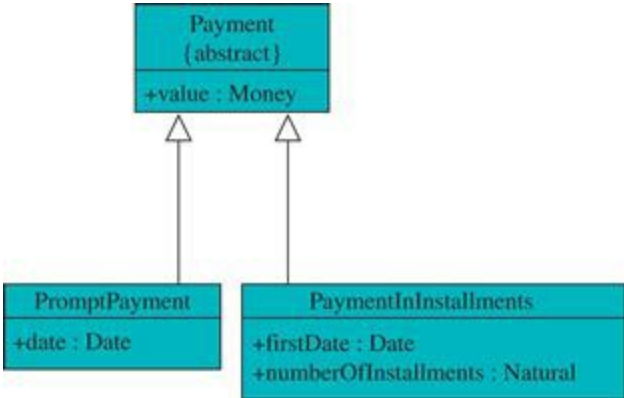


FIGURE E13.12 A situation with inheritance of attributes.

13.2.3.1 Implementing the entire hierarchy in a single table

One solution for representing inheritance that sounds straightforward at first glance is to implement the entire hierarchy in a single table. This table contains all the attributes of all classes in the hierarchy. It also has to identify which class is being represented in each record; this can be accomplished by adding a *type* column, as in [Table e13.16](#).

Table e13.16

Implementation of Inheritance in a Single Table with a *Type* Field

Table: Payment					
PK,NN,Uniq,Im	NN,Im	NN			
pkPayment	type	value	date	firstDate	numberOfInstallments
200001	promptPayment	300.00	04/09/2014		
200002	promptPayment	251.00	07/02/2014		
200003	paymentInInstallments	1,890.00		12/09/2013	12

Only attributes that belong to the class at the top of the hierarchy may not be null, because all other attributes could be null when the record belongs to one subclass and the attribute to another subclass. For example, if the type of the record is *promptPayment*, then *firstDate* and *numberOfInstallments* must necessarily be null. If it is assumed that an object cannot change its class after instantiation, then the *type* attribute should be immutable.

If multiple inheritance is used (for example, if a payment could be prompt and installments at the same time),⁵ then the *type* column should be replaced by a set of Boolean columns: *isPromptPayment* and *isPaymentInInstallments*. In this case, a payment could be prompt, in installments, or both (or neither, if they could also be instances of *Payment*, which in the present example cannot occur because it is an abstract class).

This approach has some disadvantages. It is hard to manage consistency between subclasses because all data is stored in the same place. Managing it adequately would require various complex control mechanisms. Also, lots of fields in the table would be null all the time and the big table would be a very sparse one, especially if a big and complex hierarchy is being represented.

13.2.3.2 Each concrete class as a single table

Another approach to represent inheritance is to represent each concrete class as a separate table. Each table would contain the attributes of the concrete class and the attributes of all of its superclasses. Tables e13.17 and e13.18 show a possible implementation using that approach.

Table e13.17

**Implementation of Inheritance Using a Table for Each Concrete Class:
*PromptPayment***

Table: PromptPayment		
PK, NN, Uniq, Im	NN	NN
pkPromptPayment	value	date
200001	300.00	04/09/2014
200002	251.00	07/02/2014

Table e13.18

Implementation of Inheritance Using a Table for Each Concrete Class: *PaymentInInstallments*

Table: PaymentInInstallments			
PK, NN, Uniq, Im	NN	NN	NN
pkPayment	value	firstDate	numberOfInstallments
200003	1,890.00	12/09/2013	12

If only attributes are inherited, this approach may work. However if superclasses define their own associations that must be inherited, then managing this becomes a headache because either an associative table should fill in data from different tables in a single column, or a single associative table should be implemented for each subclass that inherits the association.

13.2.3.3 Each class in a single table

A better choice for implementing inheritance given the aforementioned problems is to define one table for each class of the hierarchy, even the abstract ones. That way, attributes and association inheritance may be easily implemented. The main disadvantage is that for instantiating an object, a number of tables equal to the number of its superclasses plus one should be accessed. [Tables e13.19](#) to [e13.21](#) show how to implement this approach. There must be references from the subclasses' tables to each immediate superclass table.

Table e13.19

Implementation of Inheritance Using a Table for Each Class: *Payment*

Table: Payment	
PK,NN,Uniq,Im	NN
pkPayment	value
200001	300.00
200002	251.00
200003	1,890.00

Table e13.20

Implementation of Inheritance Using a Table for Each Class: *PromptPayment*

Table: PromptPayment		
PK,NN,Uniq,Im	NN,Uniq,Im	NN
pkPromptPayment	fkPayment	Date
300001	200001	04/09/2014
300002	200002	07/02/2014

Table e13.21

Implementation of Inheritance Using a Table for Each Class: *PaymentInInstallments*

Table: PaymentInInstallments			
PK,NN,Uniq,Im	NN,Uniq,Im	NN	NN
pkPaymentInInstallments	fkPayment	firstDate	numberOfInstallments
400001	200003	12/09/2013	12

13.3 Saving and loading objects

The equivalence between object-oriented design and the relational database is just part of the compatibility issue between these two models. It is necessary also to decide how and when objects will be loaded and saved to the database. Some designers prefer to determine themselves the moment when such operations should be performed. However, that *handcrafted* approach for saving and loading objects is subject to logic

errors, and usually it pollutes the domain-level code.

In addition, if the designer is the one who decides when to save and load objects, sometimes those operations could be performed unnecessarily (for example, loading objects that are already in memory and saving objects that were not changed). Controlling these issues case by case, method by method is not the most productive way to develop software.

It is possible to implement the processes for saving and loading objects with automatic mechanisms. In this case, the designer should only decide which classes, attributes, and associations are persistent, and a whole set of methods and data structures will be automatically created to allow those elements to be loaded and saved at the appropriate moments. Initially this section presents the basic or naïve implementation of this mechanism. Later, the limitations of the technique are explained and possible solutions drafted.

13.3.1 Virtual proxy

In order to implement an automatic mechanism for saving and loading objects, we can use a design pattern called *virtual proxy* (Gamma, Helm, Johnson, & Vlissides, 1995). A virtual proxy is a very simple object that implements only two responsibilities:

- It must know the value of the primary key of the real object it represents.
- It must redirect to the real object all messages it receives in its name.

Below is a draft of the way a virtual proxy works:

```
Class VirtualProxy
```

```
var realObjectPk:PrimaryKey  
for any message msg received do  
  realObject:=BrokerManager.get(realObjectPk)  
  realObject.msg()  
end for
```

```
end class
```

Later, in the following sections, the way the *BrokerManager* works is gradually explained.

Thus, the design with virtual proxies requires that instead of associating domain objects directly with other domain objects, they must be associated to their proxies. In this way, it is possible to bring into memory an instance of *Publisher* without loading all instances of *Book* that are linked to it. The instance of *Publisher* is associated to the *proxies of books*, which are very simple objects. The proxies are created in main memory and require much less space than the instances of the real class, such as those of *Book*. An instance of *Book* is loaded only if necessary, that is, only if a message is sent to it through its proxy. This economic way of using memory is called *lazy load*, and it is very efficient in terms of time and main memory in some situations.

To prevent the designer from worrying about when the objects must be loaded, the virtual proxy mechanism must be interposed to all persistent links in main memory. Real objects send messages to each other as if the proxies did not exist. But proxies intercept every message. The proxies ensure that the real object will be loaded if it is not in memory.

Figure e13.13 is an example of the lazy load mechanism. Initially (Figure e13.13a), only an instance of *Publisher* is in main memory. It is associated to three books. However, instead of having the books in memory, only their proxies are there. If the instance of *Publisher* must send a message to one of the books, it simply sends the message through the link; the message is intercepted by the proxy that calls the *BrokerManager*, which ensures that the book is loaded into the memory (Figure e13.13b). As the book is associated to some chapters, only the chapters' proxies are created in memory, not the real objects.

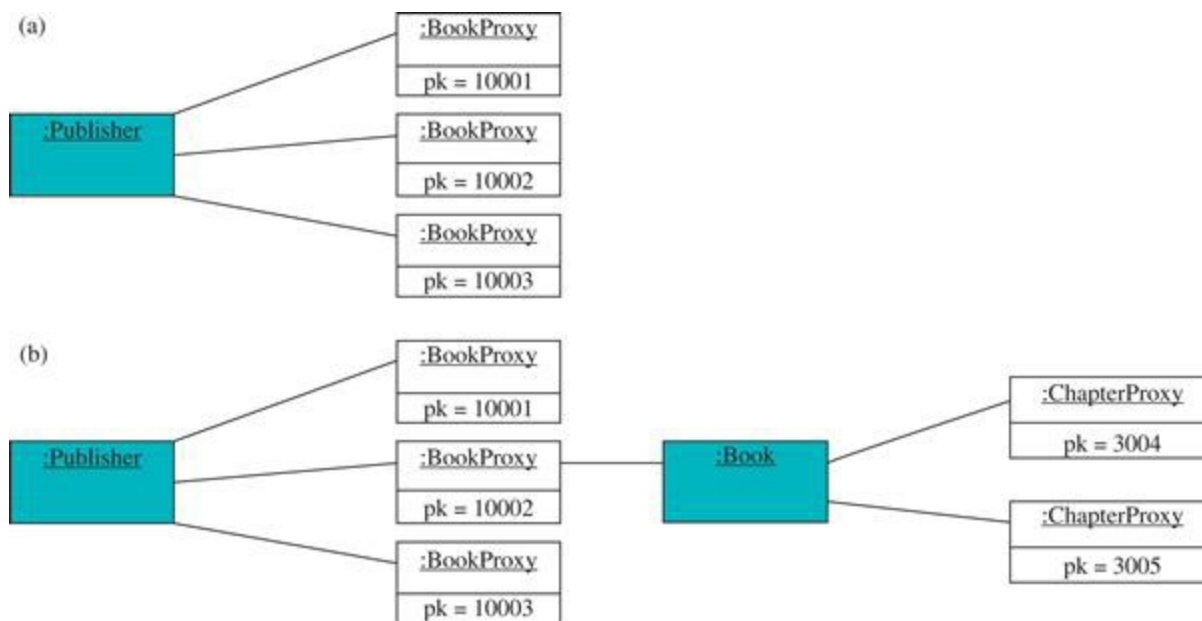


FIGURE E13.13 (a) Initial state in which only a publisher and book proxies are in main memory. (b) Situation after the publisher sends a message to one of the books.

If one of the chapters receives a message from the book, then only that chapter would be brought into memory.

13.3.1.1 Virtual data structures

The implementation of virtual proxies for each object may be very inefficient when an object has many links; for example, a publisher with 50,000 registered books would demand the instantiation of 50,000 proxies to be associated to it when it is brought into memory. Fortunately, there is a way to avoid instantiating large quantities of proxies, which is the implementation of virtual data structures to physically replace the implementation of the associations in main memory.

Thus, a publisher would not have 50,000 links to 50,000 proxies of *Book*, but a single link to a *VirtualSet* structure with 50,000 PKs. The *VirtualSet* implements regular

operations to add, remove, update, and query objects: the same operations a normal set must implement. The only difference is that it does not store the real objects, but only their primary keys. The virtual set does not bring objects into memory; it brings only their primary key numbers. The *VirtualSet* and its counterparts, *VirtualSequence*, *VirtualOrderedSet*, *VirtualBag*, *VirtualMap*, etc., use the *BrokerManager* to load real objects when necessary instead of holding all the real objects.

A virtual data structure may be implemented with the following principles:

- Instead of a physical representation of a collection of objects, it is a physical representation of a collection of the primary key values of the real objects.
- The method that adds an object to the collection must only add the primary key of the real object to the physical representation.
- The method that removes an object from the collection must only remove the primary key of the real object from the physical representation.
- Any method that performs a query on the data structure to return one or more objects receives the real object(s), which are requested from the broker manager.

Thus, adding and removing links between objects may be performed without having the real objects in memory (at least from one side of the link). An object is only brought into memory when information about it becomes necessary, that is, when it receives a message.

13.3.1.2 Discussion

Lazy load is useful because it only brings into memory objects that are going to receive a message. For example, if an instance of *Publisher* is in main memory and one of its books is going to be updated, then it is not necessary to load all instances of *Book* linked to the publisher, but only one.

On the other hand, if the instance of *Publisher* must search its books to find the most expensive of them, then all instances of *Book* that are linked to it must be loaded into main memory. This is the drawback of the technique. Performing all operations on objects in main memory can be an extraordinary waste of time. Imagine loading 50,000 instances of books to main memory just to discover which one of them has the highest price.

Virtual proxies work well when relatively few objects are brought into memory for each user operation. For example, a user searching for books to buy will only view a relatively small quantity of books in the list. That user would operate over just a single shopping cart and order. Those kinds of operations over small sets of objects are perfectly handled automatically by virtual proxies. However, when queries or commands involve iterating over large collections of objects, such as increasing the price of all books in the store, then using virtual proxies should be avoided, unless wasting processing time is not a problem; but that usually is not the case.

Thus, for performing queries or commands over large collections of objects, if the query or command uses just a few attributes of the objects, it would be advisable to consider replacing the virtual proxy mechanism in that specific case by a query or command directly performed over the database. Therefore, some messages, when they

reach specific objects, would be redirected to a specific encapsulated implementation that performs the necessary actions in the database and returns the results as necessary, rather than being redirected to a proxy.

13.3.2 Brokers and materialization

The process of loading an object from the database into main memory is called *materialization*. Materialization is usually requested by a proxy from a *broker manager*, which may in turn delegate materialization to a *specialized broker*. The broker manager looks if the requested object is in memory. If it is not, then the broker manager activates the specialized broker to materialize the object.

Each class may have its own specialized broker. However, a single broker may also be implemented to serve all classes. A specialized broker must implement a method called *materialize* that does the following:

1. It creates in main memory an instance of the persistent class.
2. It initializes the values of the attributes of the new instance with values taken from the respective line and column in the database.
3. It loads and initializes the virtual data structures that implement the associations of the object with the primary keys of the respective linked objects.

In order to obtain the values for the primary keys of the linked objects, the specialized broker must know what associations are attached to the object being loaded; then it searches the occurrences of the primary key of the object in the associative tables that implement those associations. The primary keys of other objects associated to the primary key of the object being materialized are added into the virtual data structure, which has the responsibility of holding the respective association.

For example, *Broker4Book* should materialize instances of *Book*, as defined in [Figure e13.14](#).

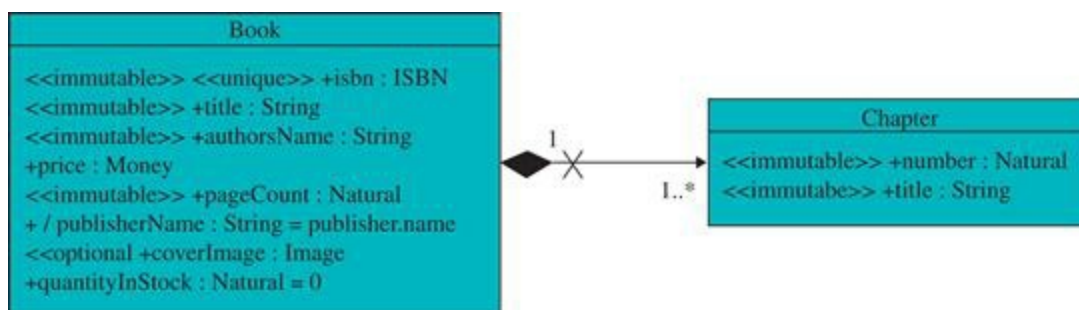


FIGURE E13.14 Reference conceptual model.

According to this conceptual model, *Broker4Book* must implement the *materialize* method by performing the following operations:

1. Create an instance of *Book*.
2. Fill in the *isbn*, *title*, *authorsName*, *price*, *pageCount*, *coverImage*, and *quantityInStock* attributes of the new instance with the values stored in the respective columns of the

Book column in the database.

3. Search the *book_chapter* table for occurrences of the primary key of the book in the *fkBook* column. For every occurrence, add the corresponding value found in the *fkChapter* column to the virtual set *chapters* of the new instance of *Book*, which implements the association role for the book.

The materialization performed by the specialized broker must not be confused with the creation of a new instance as defined by the system operation contracts ([Chapter 8](#)). In the contracts, the creation of an instance refers to the insertion of new information in the system, independent of the physical storage (main memory or secondary memory). The materialization performed by the *broker* only refers to the operation of bringing into the main memory an existing object that is not there physically yet. Materialization is, therefore, an operation that belongs exclusively to the persistence tier; it has nothing to do with the business tier.

13.3.3 Caches

Objects in main memory may be classified as follows regarding their state related to the database:

- *Clean* or *dirty*, depending on whether or not they are consistent with the version stored in the database.
- *Old* or *new*, depending on whether or not they already exist in the database.
- *Deleted* or *kept*, depending on whether they have been deleted in main memory but not yet deleted in the database.

A *cache* is a data structure, similar to a *map* or *dictionary*, that associates primary key values to the real objects it represents.

Although there are eight possible combinations for the features defined above, in practice only four combinations are sufficient to manage objects in main memory:

- *Old clean cache*: Keeps objects that are consistent with the database.
- *Old dirty cache*: Keeps objects that exist in the database but have been updated in main memory and are, therefore, inconsistent with the database.
- *New cache*: Keeps objects that have been created in main memory, but which are not yet in the database.
- *Delete cache*: Keeps objects that were deleted in memory, but still exist in the database.

The broker manager verifies that an object is in main memory by performing a query on the existing caches, asking for the primary key of the object being requested. If the broker manager finds a reference to that object in one of the caches, it returns the reference to the proxy that asked for it. On the other hand, if the broker manager does not find any reference to the requested object in any cache, then it asks a specialized broker (as, for example, *Broker4Book*) to materialize the object. The object is then materialized and inserted into the *old clean cache*.

If, by any chance, that object is updated in main memory, that is, if some of its attributes or links have been changed, it must be moved to the *old dirty cache*.

The persistence mechanism must have ways to assure that every time the inner state of

an object in the old clean cache is changed it is moved to the old dirty cache. This could be accomplished, for example, by adding a special command such as *BrokerManager.becomeDirty(self)* to any method that changes the object. If the object originally is in the old clean cache, that method will move it to the old dirty cache.

Objects that were created in memory as a result of a contract postcondition are stored into the *new cache*. An object that is in the new cache may not be moved to the old dirty cache, even if its attributes have been changed. It stays in the new cache until a *commit* is performed. After the commit it is moved to an old clean cache.

When an object is deleted from main memory, the result will depend on which cache the object came from. If it was in the old clean cache or in the old dirty cache, then it is moved to the *delete cache*. However, if it was in the new cache, then it may be simply deleted from main memory.

13.3.3.1 Commit and rollback

The *commit* and *rollback* operations are usually activated by the interface tier to indicate that a transaction was successful and confirmed, or that it was cancelled, respectively. These operations are implemented by the broker manager. In the case of a commit, the broker manager should do the following:

1. Perform an *update* in the database for every object in the old dirty cache, and move those objects to the old clean cache.
2. Perform an *insert* in the database for every object in the new cache, and move those objects to the old clean cache.
3. Perform a *remove* in the database for every object in the delete cache, and delete those objects from main memory.

In the case of a *rollback*, the broker manager must simply remove all objects from all caches, except those in the old clean cache.

As the old clean cache may grow indefinitely, it is necessary to implement some mechanism to remove the oldest objects from it every time its size reaches some previously established threshold.

The other caches only grow up to the moment a commit or rollback is performed. At that moment, they are emptied.

13.3.3.2 Cache control in a multiuser server

If more than one user connects to the system, it is necessary to determine how to share objects among different users. Assuming we have a client/server architecture with interface, domain, and persistence tiers, at least two approaches are possible:

- *The three tiers are executed in the client.* There is no main memory share in the server, only the database. The client is heavy and the server is used only to access or store data after a commit is performed. In this case, what travels in the network is data in the form of relational table records and SQL⁶ instructions. Information travels only when an object must be materialized or committed. The disadvantage of this design is that the client node is overloaded. However, client applications have recently

become richer and more complex: not depending on a server to process the logic of the application may be a crucial requirement.

- *The domain and persistence tiers are implemented in the server, and the interface tier is implemented in the client.* In this case, the objects will exist in main memory *only in the server*, and what travels through the network are the parameters of the system operations and the returns from queries. The advantage is that clients are lighter, and usually it is cheaper to upgrade a server than upgrading thousands of clients. However, for some applications that require fast processing of the data, waiting for the server may not be viable.

If the objects are physically in the server only, there are still more possibilities. In one case, all users share the four caches, with the disadvantage that a user could have access to objects that are being modified but have not yet been committed by another user. This option seems to be unadvisable for most applications.

The other option is to share only the objects in the old clean cache among the users. There should be multiple instances of other caches that are private to each user. If an object is in a private cache of one user, then the other users cannot access it. They must wait for a commit or rollback from the first user. Thus, the persistence mechanism for multiuser access could be implemented like this:

- An old clean cache is shared by all users.
- Each user has a private old dirty cache, delete cache, and new cache.

By doing this, it is possible to ensure that no user would have access to objects that are being modified by other users. Therefore, it is possible to use the caches to implement a *lock* mechanism, that is, when a user is updating an object, the other users cannot access it. Only when the user that has the object performs a commit or rollback, and the object is moved to the old clean cache or cleaned from main memory, can the other users gain access to it again.

An advantage of this method is the optimized usage of server main memory. All users share the objects in the old clean cache, which is the only one that grows indefinitely. The other four caches, which are specific for each user, only grow during a transaction. When a commit or rollback happens, those caches are emptied.

However, this may negatively affect scalability, because some users could be stuck while others are dealing with some objects. It is possible to implement a more sophisticated control mechanism based on optimistic merges: two or more users may edit the same object as long as they do not update the same attribute or association.

This must be used with care, however. Concurrency issues are always complicated concerns and many decisions may depend on business rules. For example, may the price of a book be updated while a user is finishing an order?

Some applications could even require a more pessimistic locking strategy: if a user is browsing an object, no other user may even have access to view it. This is the case for banking applications, for example, which require maximum data security.

13.4 The whole process

	Inception	Elaboration
Business Modeling	<p>Build a general view of the system:</p> <ul style="list-style-type: none">• Build a business use case diagram and determine the automation scope for the project.• Build preliminary activity diagrams for business use cases.• Build preliminary state machine diagrams for key business objects.	

(Continued)

	Inception	Elaboration
Requirements	<p>Prepare the system use case diagram (functional requirements):</p> <ul style="list-style-type: none"> Identify the system actors from the business use case model. Identify the system use cases from the business use case model, and the activity and state machine diagrams from business modeling. <p>Identify nonfunctional requirements as use case annotations:</p> <ul style="list-style-type: none"> Identify the main business rules associated to use cases. Identify the main quality issues associated to use cases. <p>Identify the supplementary requirements.</p>	<p>Detail requirements by expanding use cases:</p> <ul style="list-style-type: none"> Identify the main flow. Identify the alternate flows: variants and exception handlers.
Analysis and Design	<p>Prepare the preliminary conceptual model by observing system use cases and the concepts needed by them.</p>	<p>Elaborate the system sequence diagrams:</p> <ul style="list-style-type: none"> Represent the main flow of a use case as a system sequence diagram. Represent the system commands and queries using stateful or stateless strategies. Complete system sequence diagrams with alternate flows. <p>Refine the conceptual model:</p> <ul style="list-style-type: none"> Identify concepts, attributes, and associations in the text of expanded use cases. Detail attributes and associations with stereotypes, multiplicity, and constraints as needed. Organize the model by using inheritance, association classes, and temporal specifications. Add invariants as needed. Improve the conceptual model with the application of analysis patterns. <p>Write system operation contracts for commands and queries in system sequence diagrams:</p> <ul style="list-style-type: none"> Identify preconditions and exceptions based on invalid parameters and complementary constraints. Identify postconditions for commands and returns for queries.

(Continued)

	Inception	Elaboration
Implementation		<p>Design the domain tier:</p> <ul style="list-style-type: none"> • Create a sequence or communication diagram for each system command contract. • Use those diagrams to decide which methods to implement in each class. • Inspect those diagrams to discover which associations can be unidirectional and define them as such. • Look at system query contracts and decide which delegate queries could be implemented; some of them could be derived attributes or derived associations. • Produce or refine the design class diagram. <p>Design the interface tier:</p> <ul style="list-style-type: none"> • Design the top-level interface organization based on areas and pages. • Refine pages, including view components for parameters of system operations and results for system queries. • Associate operation activation to interface events such as buttons and menus. <p>Design the persistence mechanism:</p> <ul style="list-style-type: none"> • Reuse or build a persistence mechanism based on ORM, virtual proxies, caches, and brokers. • Decide where to apply the default persistence mechanism and where to implement direct database access for performance optimization. <p>Generate code:</p> <ul style="list-style-type: none"> • Generate code for classes. • Generate code for attributes. • Generate code for associations. • Generate code for necessary basic methods such as get, set, add, remove, create, destroy, and others. • Generate code for system operations and delegate methods using dynamic models as reference.

(Continued)

	Inception	Elaboration
Test		Plan and execute tests: <ul style="list-style-type: none"> • Perform unit tests for methods and classes that were not generated automatically. • Perform system operation tests based on system operation contracts. • Perform use case tests based on use case scenarios.
Project Management	Estimate total effort, ideal calendar time, and average team size for the project. Estimate the duration and quantity of iterations for each phase. Prepare the phase plan and iteration plan for the first iteration.	

13.5 Questions

1. What kind of associative table should be used to represent an association that is ordered in both directions? How do you represent it in the case of two ordered sets, or two sequences, or one ordered set and one sequence?
2. What kind of association table should be used to represent an association with an association class that is marked with `{bag}` on one side?

¹See, for example: <http://www.hibernate.org/>.

²*Persistent* in this context is the opposite of *transient*, that is, persistent information is information that must be kept until some user explicitly deletes it. Transient information is kept only during a session of use of the system.

³*Secondary memory* is usually slower than *main memory*. However it is also much cheaper, and thus it is widely used to store large quantities of data that do not fit in the more expensive main memory. Usually secondary memory consists of media such as magnetic or optical discs or tapes. Data stored in secondary memory usually cannot be processed directly by the computer unless it is loaded into main memory. Main memory is also known as *RAM* (*random access memory*) and physically it is usually implemented by electronic integrated circuits.

⁴Note that this is a normal association and not a derived one.

⁵Of course this cannot be true. It's just a supposition for the sake of the example.

⁶*Structured Query Language*, the dominant language for defining, accessing, and updating relational databases (Date, 1982).

Postface

Raul Sidnei Wazlawick

At this point the reader must have been asking why a picture of a loom is on the front cover of this book. Even if you are not asking that, I'll tell you anyway. The reason is twofold: symbolic and historical.

The symbolic reason is that there are similarities between the process of producing software and producing fabrics with a loom. First you plan, and you draft the result you intend to produce; and then you execute your work and see if it is going well. The many colors of string used to produce a fabric may be compared to the many aspects of information and technology that must be weaved to produce the *patterns* that are desired in the product. Work done on a loom can be undone as work done on software can. After that, it can be refactored.

Producing nice patterns in fabrics may appear complex and confusing for unskilled people; the same is true for producing software. But if the best practices, techniques, and tools are used, the process is smoothed.

A difference between these processes is that working on looms has been automated since the eighteenth century, while the software industry is just taking the first steps towards automation. Furthermore, software development is so complex that many people doubt that it could be successfully automated someday. However, although the complexity of software is growing each year, new techniques to improve our capacity to cope with that complexity are being learned and used.

The historical reason that links looms to this book is based on the six degrees of separation:

1. One of the first automatic complex machines were *looms* controlled by *punched cards* that were invented in France around 1725 and greatly improved by Joseph Marie Jacquard in 1801. They became a big commercial success and almost cost Jacquard his life because of the fierce opposition of silk weavers that were afraid of losing their jobs.
2. *Punched cards* were used by *Herman Hollerith* as a data entry mechanism for his Electric Tabulating Machine, which was used in the American census of 1890, reducing drastically the time to obtain results.
3. *Hollerith's* company, the Tabulating Machine Company, fused with other companies in 1911 to form the Computing Tabulating Recording Company (CTR), which in 1924 changed its name to the *International Business Machines Corporation (IBM)*.
4. In 2003, *IBM* acquired, for 2.3 billion dollars, the Rational Company, which was responsible, among other products, for the *Unified Process* and the *Unified Modeling*

Language (UML).

5. The present *book* addresses many aspects of the *Unified Process*, and uses *UML* and its byproducts OCL and IFML for modeling.

As far as I can see, five degrees of separation were sufficient.

September 12th, 2013.

References

1. Adams DN. *The Hitchhiker's Guide to the Galaxy* Completely Unexpected Productions Ltd. 1979;
2. Albrecht, A.J., 1979. Measuring application development productivity. Proceedings of the Joint SHARE/GUIDE and IBM Application Development Symposium, Chicago. pp. 83–92.
3. Albrecht AJ, Gaffney Jr JE. Software function, Source lines of code, and development effort prediction: a software science validation. *IEEE Trans Softw Eng.* 1983;9(6):639–648 November.
4. Alford M. *Requirements-Driven Software Design* McGraw-Hill 1991.
5. Ambler, S.W., 2000. Web Services Programming Tips and Tricks: Modeling essential use cases. Available at: <http://www.ibm.com/developerworks/library/ws-tip-essentialuse/index.html>.
6. Ambler SW. *The Object Primer: Agile Model-Driven Development with UML 2.0* third ed. Cambridge University Press 2004.
7. Ambler SW, Jeffries R. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process* John Wiley and Sons 2002.
8. Anda B. *Estimating software development effort based on use cases: Experiences from industry. Fourth International Conference on the Unified Modeling Language* Toronto: Springer-Verlag; 2001; pp. 487–502.
9. Beck, K., 1989. Simple Smalltalk Testing with Patterns. Available at: <http://www.xprogramming.com/testfram.htm> (accessed 31.08.13).
10. Beck K. *Test-Driven Development by Example* Addison-Wesley 2003.
11. Beck K, Andres C. *Extreme Programming Explained: Embrace Change* second ed. Addison-Wesley 2004.
12. Beizer B. *Software Testing Techniques* second ed. Van Nostrand Reinhold 1990.
13. Bloch A. *Murphy's Law, and Other Reasons Why Things Go Wrong* Los Angeles: Price/Stern/Sloan Publishers, Inc.; 1980.
14. Boehm BW. *Software Cost Estimation with COCOMO II* Prentice-Hall 2000.
15. Booch G, Maksimchuk RA, Engle MW, Young BJ, Conallen J, Houston KA. *Object-Oriented Analysis and Design with Applications* third ed. Addison-Wesley Professional 2007.
16. Bourret, R., 2010. XML Database Products. Available at, <http://www.rpbouret.com/xml/XMLDatabaseProds.htm> (accessed 11.09.13).
17. Braz, M., Vergilio, S., 2006. Software effort estimation based on use cases. Proceedings of the 30th Annual International Computer Software and Applications Conference, pp. 221–228.
18. Burnstein I. *Practical Software Testing* Springer-Verlag 2003.
19. Cabot J. From declarative to imperative UML/OCL operation specifications. *Lect*

- Notes Comput Sci.* 2007;198–213.
20. Cardoso IS. *Introducing support to the declaration of associations in UML 2 in an object-oriented language* Florianópolis, Brazil: Masters Dissertation. UFSC – Federal University of Santa Catarina, PPGCC – Computer Science; 2011.
 21. Carr MJ, Konda SL, Monarch I, Ulrich FC, Walker CF. *Taxonomy-Based Risk Identification SEI* Pittsburgh: Carnegie Mellon University; 1993.
 22. Center for Software Engineering, USC, 2000. COCOMO II – Model definition manual – version 2.1. Available at:
<<http://csse.usc.edu/csse/TECHRPTS/2000/usccse2000-500/usccse2000-500.pdf>>.
 23. Ceri S, Fraternali P, Bongio A, Brambilla M, Comai S, Matera M. *Designing Data-Intensive Web Applications* Morgan Kaufmann 2003.
 24. Choenni S, Blanken H, Chang T. Index selection in relational databases. *Proc Int Conf Comput Inf.* 1993;491–496.
 25. Cockburn A. *Writing Effective Use Cases* Addison-Wesley 2001.
 26. Crain A. *RUP Iteration Planning* IBM 2004; Available at:
<<http://www.ibm.com/developerworks/rational/library/5335.html>> (accessed 15.09.13).
 27. Date CJ. *An Introduction to Database Systems* Addison-Wesley 1982.
 28. Dustin E. Automate regression tests when feasible. In: Dustin E, ed. *Effective Software Testing: 50 Specific Ways to Improve Your Testing*. Addison-Wesley Professional 2002.
 29. Emam KE, Melo W, Drouin J-N. *Spice: The Theory and Practice of Software Process Improvement and Capability Determination* Los Alamitos, CA, USA: IEEE Computer Society Press; 1997.
 30. English, A.V., 2007. Business Modeling with UML: Understanding the Similarities and Differences between Business Use Cases and System Use Cases. Retrieved November 14, 2012, from
<<http://www.ibm.com/developerworks/rational/library/apr07/english/>>.
 31. Fowler M. *Patterns of Enterprise Application Architecture* Addison-Wesley 2003.
 32. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley 1995.
 33. Gause, D.C., Weinberg, G.M. *Exploring Requirements: Quality before Design*. Dorset House Pub.
 34. Goldberg A, Robson D. *Smalltalk 80: The Language* Addison-Wesley 1989.
 35. Grady R. *Practical Software Metrics for Project Management and Process Improvement* Prentice-Hall 1992.
 36. Ibarra, G., Vilain, P., 2010. Software estimation based on use case size. Brazilian Symposium on Software Engineering, Florianópolis. pp. 178–187.
 37. Ireland, C., Keynes, M. Bowers, D., Newton, M., Waugh, K., 2009. A classification of object-relational impedance mismatch. *Advances in Databases, Knowledge, and Data Applications, DBKDA '09*, Gosier pp. 36–43.
 38. Jacobson I. *The Object Advantage: Business Process Reengineering with Object Technology* Addison-Wesley 1994.

39. Jacobson I, Christenson M, Jonsson P, Övergaard G. *Object-Oriented Software Engineering: A Use Case Driven Approach* Addison-Wesley 1992.
40. Jacobson I, Booch G, Rumbaugh J. *The Unified Software Development Process* Addison-Wesley 1999.
41. Jacobson I, Spence I, Bittner K. *Use Case 2.0: The Guide to Succeeding with Use Cases* Ivar Jacobson International 2011.
42. Jones, C., 2007. Software Estimating Rules of Thumb – version 3. Available at: <http://www.compaid.com/caiinternet/ezone/capers-rules.pdf>.
43. Kamal MW, Ahmed MA. A proposed framework for use case based effort estimation using fuzzy logic: building upon the outcomes of a systematic literature review. *Int J New Comput Archit Appl*. 2011;1(4):976–999.
44. Karner G. *Use Case Points: Resource Estimation for Objectory Projects* Objective Systems 1993.
45. Krasner GE, Pope ST. *A cookbook for using the model-view controller user interface paradigm in smalltalk-80*. *The Journal of Object Technology – JOT SIGS Publications* 1988; August-September.
46. Kroll, P., 2004. Planning and Estimating a RUP Project Using IBM Rational SUMMIT Ascendant. Available at: <http://www.ibm.com/developerworks/rational/library/4772.html>.
47. Kroll P, Kruchten P. *The Rational Unified Process Made Easy: A Practitioner's Guide to RUP* Addison-Wesley 2003.
48. Kroll P, MacIsaac B. *Agility and Discipline Made Easy: Practices from OpenUp and RUP* Addison-Wesley Professional 2006.
49. Kruchten P. *The Rational Unified Process: An Introduction* third ed. Addison-Wesley 2003.
50. Larman C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process* third ed. Prentice-Hall 2004.
51. Lieberherr K, Holland I. Assuring good style for object-oriented programs. *IEEE Softw*. 1989; pp. 38–38, September.
52. Liskov, B., 1974. Programming with abstract data types, in Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages (pp. 50–59), Santa Monica, California.
53. McConnell S. *Rapid Development* Microsoft Press 1996.
54. Meszaros G. *xUnit Test Patterns: Refactoring Test Code* Addison-Wesley 2007.
55. Meyer B. *Object-Oriented Software Construction* Prentice Hall 1988.
56. Miles R, Hamilton K. *Learning UML 2.0* O'Reilly 2006.
57. Mohagheghi P, Anda B, Conradi R. Effort estimation of use cases for incremental large scale software development. *Int Conf Softw Eng*. 2005.
58. Molyneaux I. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance* O'Reilly Media 2009.
59. Myers GJ, Sandler C, Badgett T, Thomas TM. *The Art of Software Testing* second ed. New Jersey: John Wiley & Sons; 2004.
60. Nelson WB. *Accelerated Testing: Statistical Models, Test Plans, and Data Analysis*

- John Wiley & Sons 2004.
61. Nielsen J. *Usability Engineering* Academic Press 1994.
 62. Object Management Group. *OCML 2.3.1 Specification* OMG 2010.
 63. Object Management Group. *OMG Unified Modeling LanguageTM (OMG UML), Superstructure, Version 2.4.1* OMG 2011.
 64. Ockhan W. *Questions and decision in four books - Sentences of Petri Lombardi* Johannes Trechsel 1495.
 65. Potel, M., 1996. MVP: Model-View-Presenter: The Taligent Programming Model for C++ and Java. Available at: <<http://www.wildcrest.com/Potel/Portfolio/mvp.pdf>> (accessed 01.09.13).
 66. Pressman RS. *Web Engineering: A Practitioner's Approach* McGraw Hill 2008.
 67. Pressman RS. *Software Engineering: A Practitioner's Approach* McGraw Hill 2010.
 68. Probasco L. *What Makes a Good Use Case Name?* The Rational Edge 2001; March.
 69. Ribu, K., 2001. Estimating Object-Oriented Software Projects with Use Cases. Master of Science Thesis, University of Oslo, Department of Informatics.
 70. Robiolo G, Orosco R. Employing use cases to early estimate effort with simpler metrics. *Innov Systems Softw Eng.* 2008;4(1):31–43.
 71. Royce WW. *Managing the development of large software systems. Proceedings of the IEEE WESCON* Los Angeles: IEEE; 1970; pp. 1–9.
 72. Santos, C.C., 2007. Automatic generation of communication diagrams from OCL contracts. Masters Dissertation, UFSC, Florianópolis.
 73. Satzinger JW, Jackson RB, Burd DS. *Systems Analysis and Design in a Changing World* sixth ed. Course Technology 2011.
 74. Schneider G, Winters JP. *Applying Use Cases: A Practical Guide* Addison-Wesley 1998.
 75. Scott K. *The Unified Process Explained* Addison-Wesley 2001.
 76. SEI-CMU. *CMMI for Development version 1.3* Carnegie Mellon University 2010.
 77. Sommerville I. *Software Engineering* seventh ed. Addison-Wesley 2006.
 78. Symons CR. Function points analysis: Difficulties and improvements. *IEEE Trans Softw Eng.* 1988;14(1):2–11.
 79. Tidwell J. *Designing Interfaces: Patterns for Effective Interaction Design* O'Reilly Media 2005.
 80. UKSMA Metrics Practices Committee, 1998. Mk II Function Point Analysis Counting Practice Manual – Version 1.3.1. Available from <<http://uksma.co.uk/products.asp>> upon registration.
 81. Warmer J, Keppe A. *The Object Constraint Language: Precise Modeling with UML* Addison-Wesley 1998.
 82. West, D., 2002. Planning a Project with the Rational Unified Process. Rational White Paper. Available at: <http://www.nyu.edu/classes/jcf/CSCI-GA.2440-001/handouts/PlanningProjWithRUP.pdf> (accessed 15.09.13).
 83. Won K. *Introduction to Object-Oriented Databases* The MIT Press 1990.
 84. Woolf B. Null object. In: Martin R, Riehle D, Buschmann F, eds. *Pattern Languages of Program Design 3*. Addison-Wesley 1998.

Index

Note: Page numbers followed by “*f*”, “*t*” and “*b*” refers to figures, tables and boxes respectively.

A

- Abstract data types, 133
- Abstract superclasses, 139, 140*f*
- Acceptance tests, 272, 282–283
- Ad hoc techniques, e1–e3
- Adapted use case points, e26
- Adjusted use case points (UCP), e22
- Aggregation, 129–130, 130*f*
 - Aggregation
 - composite, 129, 130*f*
 - shared, 129, 130*f*
- Agile process, 5
- Algorithms, 228
- Analysis patterns, 165
- Apparent functionality, e3
- Association classes, 141–144
 - reservation modeled as, 143*f*
 - reservation modeled as intermediary concept, 143*f*
 - suitable representation of many records, 144*f*
 - suitable representation of roles, 142*f*
 - unsuitable representation of many records, 143*f*, 144*f*
 - unsuitable representation of roles, 141*f*

Associations, [115](#), [122–132](#)

derived, [126–129](#), [127f](#), [128f](#)

direction, [126](#)

n-ary, [130–132](#), [131f](#), [131f](#), [132f](#), [132f](#)

role multiplicity, [124–126](#), [125f](#)

Associative information, [157](#)

Attributes, [115](#), [115–120](#), [116f](#)

derived, [117–118](#), [117f](#)

enumerations, [118–119](#), [119f](#)

initial values, [116–117](#), [117f](#)

primitive types, [119–120](#), [120f](#)

types, [116](#), [116f](#)

Average team size, [e24–e25](#)

B

Backfire table, [e9](#)

Bag structure, [134](#), [134f](#)

Basic queries, [258](#)

Bidirectional associations, [e39](#), [e50–e60](#)

implementation

with intermediary object, [e59–e60](#)

unidirectional, [e57–e59](#)

many-to-many associations, [e52f](#)

mutual friends, [e51–e57](#)

both roles optional, [e51–e53](#)

one mandatory role, [e53–e55](#), [e53f](#)

two mandatory roles, [e55–e57](#), [e56f](#)

Brokers, [e90–e92](#)

Business activity diagram, [18–22](#)

control flow nodes, [20–22](#)

decision control flow, [20](#), [20f](#)

parallel control flow, [20](#), [21f](#)

elements, [18–19](#), [19f](#)

Business actors, [13](#), [15](#), [15f](#)

See also [System actors](#)

Business cycle test, [282–283](#)

Business modeling, [7–9](#)

See also [Business activity diagram](#), [Business use cases](#), [State machine diagram](#)

general view of system, [9–13](#)

deliverables, [10](#)

example of, [12f](#)

product acceptance criteria, [10–11](#)

scope declaration, [7–8](#), [10](#)

goals of, [8](#)

need for, [8–9](#), [9](#)

scenarios in

domain modeling, [8](#)

generic business model, [9](#)

new business, [9](#)

one company, many systems, [9](#)

organizational chart, [8](#)

renewal, [9](#)

Business Process Model and Notation (BPMN), [7–8](#)

Business renewal, [9](#)

Business use cases, [13–18](#), [13f](#)

See also [System use cases](#)

automation, [15–18](#)

candidates for automation, [17f](#)

system boundary, [16](#), [17f](#)

updated, [22f](#)

use case names, [13–14](#)

Business workers, [13](#), [15](#), [15f](#)

C

Caches, [e92–e94](#)

- commit and rollback operations, [e93](#)

- control, [e93–e94](#)

Calendar time, [e24–e25](#)

Calling operation, [194](#)

Cascade index, [309–311](#), [310f](#), [310f](#), [311f](#), [311f](#), [312f](#), [312f](#)

Casual use cases, [37](#)

COCOMO II, [10](#), [e3](#)

- scale factors, [9–10](#)

Code generation, [e35](#)

- for attributes, [e35–e38](#)

- automatic, [e35](#)

- for classes, [e35–e38](#), [e36f](#)

- for delegated methods, [e60–e62](#)

- for system operations, [e60–e62](#)

Command-Query Separation (CQS), [92](#)

Communication diagrams, [228](#)

Composition, [129–130](#), [130f](#)

Composition model, [290](#)

Concentration, [260](#), [263f](#)

Conceptual model

- organization of, [138–149](#)

 - associative, [138](#)

 - structural, [138](#)

 - temporal, [138](#)

- refinement of, [152–162](#)

 - associations, [157–158](#), [159f](#), [159f](#), [159f](#)

 - concepts and attributes, [152–155](#), [154f](#)

 - dependent concepts, [156–157](#)

example of, [159–162](#)

independent concepts, [156–157](#)

static aspect, [113](#)

Conceptual model patterns, [165](#)

account/transaction, [178–181](#), [178f](#), [180f](#), [181f](#), [182f](#)

composite, [173–174](#), [174f](#)

high cohesion, [165–168](#), [166f](#), [167f](#), [167f](#), [167f](#), [168f](#), [168f](#)

measure, [171](#), [171f](#)

object joining, [175–178](#)

copy and replace, [176](#)

essence/appearance, [177](#), [177f](#)

superseding, [176–177](#), [176f](#)

undoing join, [177–178](#)

organizational hierarchy, [174–175](#), [175f](#), [175f](#)

quantity, [170–171](#), [170f](#), [171f](#)

range, [182–183](#), [185f](#)

specification classes, [168–170](#), [169f](#), [169f](#)

strategy, [172](#), [173f](#)

temporal, [183–189](#)

bitemporal pattern, [187–189](#), [188f](#)

effectivity, [185](#), [185f](#)

history association, [185–186](#), [186f](#)

temporal association, [186–187](#), [187f](#), [187f](#), [188f](#)

Conceptual modeling, [113–115](#), [114f](#)

See also [Preliminary conceptual model](#)

associations, [115](#), [122–132](#)

attributes, [115](#), [115–120](#), [116f](#)

collections, [132–138](#), [133f](#), [133f](#)

concepts, [115](#), [120–122](#)

Concrete data types, [133](#)

Concrete use cases, [36](#)

Contingency plan, [e5](#), [e7](#)

Contracts

for CRUD, [216–222](#)

create contract, [217–218](#)

delete contract, [218–221](#), [221f](#), [221f](#)

retrieve contract, [221–222](#), [221f](#), [221f](#)

update contract, [218](#)

for listing objects, [222–223](#)

for use cases, [223–224](#)

CRUD (Create, Retrieve, Update, and Delete), [32](#), [55](#)

CRUD expanded use case, [86–89](#)

with access restrictions, [90f](#)

example of, [88f](#), [91f](#)

template for, [87f](#)

CRUD operations, IFML models for, [325–328](#), [326f](#), [326f](#), [327f](#), [327f](#), [328f](#), [328f](#)

D

Data flow, [304](#)

Data model, [113](#)

Data persistence, [e67–e68](#)

Database management system (DBMS), [30–31](#)

DCD building, [228](#)

DCD (design class diagram), [227](#)

Delegate methods, [e60–e62](#)

Delegation, [2](#), [259–262](#), [263f](#)

Derivation model, [290](#)

Derived associations, [117](#)

Design class diagram (DCD), [113](#), [227](#), [262–265](#), [265f](#)

Design patterns, [165](#)

Disaster recovery plan, [e5](#), [e7](#)

Domain modeling, [8](#)

Domain tier design, [227–229](#)

Dynamic model, [113](#)

Dynamic modeling, [228](#), [245–256](#)

- attribute value modification, [250](#), [251f](#)

- conditional postconditions, [253–255](#), [254f](#)

- exceptions, [255](#), [256f](#)

- instance creation, [246–248](#), [248f](#), [249f](#)

- instance destruction, [250–251](#), [252f](#)

- link addition, [249–250](#), [249f](#), [250f](#)

- postconditions over collections, [255–256](#), [256f](#)

- removing link, [251–253](#), [253f](#)

E

Effort, [e23–e24](#)

Effort estimation, [e1](#), [e7](#)

Effort per use case point, [e28](#)

Effort value, [e23](#)

Electronic focus groups, [11](#)

Equivalence partitioning, [272–273](#)

Essential use cases, [36](#), [75–76](#)

Estimation by analogy, [e2](#)

Exception handler

- corrective actions, [72](#)

- example of, [73f](#)

- exception, [72](#)

- finalization, [72](#)

- identification code, [72](#)

Exceptions, [69](#), [194](#), [214–216](#)

- in command contracts, [194](#)

- in query contracts, [194](#)

- transformed into precondition, [214](#)

Expanded use cases, [61–62](#)

alternate flows, [64–74](#)

exception handling, [69–74](#), [70f](#), [73f](#)

scenarios, [65–66](#)

variants, [66–69](#), [67f](#), [68f](#)

main flow of, [62–64](#)

exception handling, [63–64](#), [65f](#)

first draft, [63f](#), [64f](#)

open issues, [92](#)

preconditions, [91–92](#)

stakeholders, [91](#)

success post conditions, [92](#)

as system sequence diagrams, [95–97](#), [96f](#), [96f](#)

uses, [92](#)

writing recommendations, [74–83](#)

complementary steps, [81](#), [81f](#)

essential vs real use case, [75–76](#)

explicit information, [76–77](#)

identification and selection, [77](#)

mandatory steps, [77–81](#), [78f](#), [80f](#)

unsuitable steps, [82–83](#), [82f](#), [83f](#)

Expert judgment, [e1](#)

Exploratory surveys, [11](#)

F

Filter object, [e63](#), [e63f](#)

Filtered index, [312](#), [312f](#)

Filtered queries, patterns for, [e62–e64](#)

Full use case size method, [e26](#)

Fully dressed use cases, [37](#)

Function point analysis, [e3](#)

Function points, [10](#), [e3](#)

Functional modeling, [193–196](#)

Functional requirements, [41](#), [45](#)

Functional testing, [272–274](#)

Functional tests, [272](#)

FUSP (Fuzzy Use Case Size Points), [e26](#)

G

Generalization, [139–140](#), [139f](#), [140f](#)

Generic business model, [9](#)

Guided tour, [313](#), [313f](#), [314f](#)

H

Hardware attributes, [e4](#)

High-level requirements, [29–30](#)

Human actors, [30](#)

I

If-then-else-endif, [147](#)

IFML (Interaction Flow Modeling Language), [289](#), [289–290](#)

Immutable attributes, [e36](#)

Included fragments, [83–84](#), [84f](#), [84f](#)

Included use cases, [83–84](#)

Indexed column, [e71](#)

Inheritance, [139–140](#)

Instance variables, [e38](#)

Interaction Flow Modeling Language (IFML), [289](#), [289–290](#)

 areas, [308](#)

 default page, [309](#)

 flows, [302–305](#)

 homepage, [309](#)

hypertext organization, [305–309](#)

landmark page, [309](#)

pages, [301–302](#), [301f](#), [301f](#), [302f](#)

site view, [307–308](#), [309f](#)

view components, [290–300](#)

- checkable list, [297](#), [297f](#)

- details, [291–293](#), [292f](#), [292f](#), [293f](#)

- forms, [297–298](#), [299f](#)

- hierarchies, [298–300](#), [299f](#), [300f](#), [300f](#)

- list, [295–296](#), [296f](#), [296f](#), [297f](#)

- multiple details, [293–294](#), [293f](#), [294f](#), [294f](#)

- simple list, [294](#), [295f](#), [295f](#)

Interface tier design, [227](#), [289](#)

Intermediate queries, [258](#)

Invariants, [149–152](#)

- constraint, [149f](#)

- example of, [150f](#)

IsEmpty(), [147](#)

Iteration planning, [e30–e32](#)

Iterative surveys, [11](#)

K

Kilo source lines of code (KSLOC), [e3](#)

KO flow, [319](#)

L

Lazy load, [e88](#), [e89f](#), [e90](#)

Limit value analysis, [274](#)

Lines of code, [e3](#)

Logical design, [227](#), [227–228](#)

See also [Domain tier design](#)

Low coupling, [259–262](#)

M

Map, [135–136](#), [135f](#), [136f](#)

Mark II counting, [e7–e9](#)

Materialization, [e90–e92](#)

Modal classes, [144–149](#)

Modeling operations in interface, [317–325](#), [318f](#)

- connect, disconnect, and reconnect operations, [323–325](#), [324f](#), [324f](#), [325f](#)

- create operation, [319](#), [320f](#), [320f](#)

- delete operation, [319–320](#), [321f](#), [321f](#)

- update operation, [321–322](#), [322f](#), [322f](#), [323f](#)

Monotonic increasing transition, [145](#), [145–147](#), [146f](#)

- effective way to model, [147f](#)

- unsuitable way to model, [146f](#), [147f](#)

Multivalued parameter binding, [305](#), [307f](#), [308f](#), [308f](#)

N

Navigation model, [290](#)

New business, [9](#)

Nonfunctional requirements, [45–46](#)

Nonfunctional tests, [283–284](#)

Nonmonotonic transition, [145](#), [147–149](#)

- modeled with state machine diagram, [148f](#)

- modeled with the state design pattern, [149f](#)

Nonparametric estimation, [e1–e2](#)

Nonpattern use cases, [e11](#)

Normal navigation flow, [302–304](#), [303f](#), [303f](#), [303f](#), [304f](#)

Number sequence generator, [e71](#)

O

Object equivalence, [177](#)

Object-oriented CASE tools, [2](#)

Object-oriented databases, [e67](#)

Object-oriented design, [e67–e68](#)

Object-oriented systems development, [2](#)

Object-relational impedance mismatch, [e68](#)

Object-relational mapping (ORM), [e67](#), [e68–e87](#)

associations, [e71–e83](#)

association class, [e80–e82](#), [e81f](#), [e81t](#), [e81t](#)

bags, [e78](#), [e78f](#), [e79t](#)

façade controller, [e83f](#)

many-to-many associations, [e72–e73](#), [e72f](#), [e73t](#)

n-ary, [e82](#), [e82f](#), [e82t](#)

one-to-many associations, [e74–e76](#), [e74f](#), [e74t](#), [e75t](#)

one-to-one associations, [e76–e77](#), [e76f](#), [e76t](#)

ordered associations, [e77–e78](#), [e77f](#), [e77t](#)

qualified associations, [e79–e80](#), [e79f](#), [e80t](#)

transient, [e83](#)

attributes, [e68–e71](#)

classes, [e68–e71](#), [e69f](#), [e70t](#)

inheritance, [e83–e87](#), [e84f](#)

each class, [e86–e87](#), [e87t](#), [e87t](#), [e87t](#)

each concrete class, [e86](#), [e86t](#), [e86t](#)

implementation of, [e84–e86](#), [e85t](#)

Object responsibility distribution, [229–232](#), [231t](#)

Object visibility, [232–245](#)

by association, [232–239](#)

with association class, [237–239](#), [238f](#), [238f](#), [238f](#), [239f](#), [239f](#)

multiple objects, [233](#), [234f](#), [234f](#), [234f](#)

with ordered roles, [233–235](#), [234f](#), [235f](#), [235f](#), [235f](#)

preconditions, [239](#), [239f](#), [240f](#)

with qualifiers, [235–237](#), [236f](#), [236f](#), [236f](#), [236f](#), [237f](#), [237f](#)

single object, [233](#), [234f](#), [234f](#)

global, [244–245](#), [245f](#)

locally declared, [242–244](#), [244f](#), [244f](#)

by parameter, [240–242](#), [240f](#), [240f](#), [241f](#), [241f](#), [242f](#), [243f](#)

OCL (Object Constraint Language), [23](#), [116](#)

OK flow, [319](#)

One company, many systems, [9](#)

Open-close principle, [44](#)

Ordered set structure, [134](#), [134f](#)

Organizational chart, [8](#)

Overview plus detail pattern, [313](#), [316f](#), [316f](#)

P

Parameter binding, [304–305](#), [305f](#), [306f](#), [306f](#), [307f](#), [307f](#)

Parametric techniques, [e3–e4](#)

Parkinson's Law, [e2](#)

Partition, [136–137](#), [137f](#)

Pattern use cases, [e10](#), [84–89](#)

Persistence tier design, [227](#)

Personnel attributes, [e4](#)

Phase planning, [e29](#), [e32t](#)

Postconditions, [194](#), [203–214](#)

changing attribute value, [205](#)

collections of objects, [213–214](#)

combinations of, [211–212](#)

former values, [212–213](#)

instance creation, [206–207](#)

Instance destruction, [209](#)

link addition, [207–208](#)

and real-world events, [214](#)

removing link, [209–210](#)

well-formed, [210–211](#)

Preconditions, [194](#), [196–200](#)

assurance, [199–200](#)

complementary constraints, [198–199](#)

and exceptions *versus* invariants, [200](#)

parameter guarantee, [197–198](#)

Preliminary conceptual model, [52–56](#), [53f](#), [54f](#), [56f](#)

Presentation model, [290](#)

Pricing to win, [e2](#)

Primary actors, [31](#)

Product attributes, [e3](#)

Project attributes, [e4](#)

Project planning, [e26–e32](#)

duration of iterations, [e27](#)

number of iterations, [e27–e28](#)

Q

Query return, [201–203](#)

R

Real use cases, [75–76](#)

Reference class, [e69f](#)

Reference conceptual model, [195f](#), [e91f](#)

Refined conceptual model, [193](#)

Relation, [137–138](#), [137f](#)

Relational databases, [e67](#)

Relational table, [e68](#)

Report expanded use case, [85–86](#)

example of, [86f](#)

template for, [86f](#)

Reports, [e10](#)

Requirements, [41–51](#)

challenges, [44–45](#)

desired, [47](#)

functional, [45](#)

evident, [45](#)

hidden, [45](#)

mandatory, [47](#)

nonfunctional, [45–46](#)

logical issues, [45](#)

permanent, [46](#)

technological issues, [45](#)

transient, [46](#)

supplementary, [45](#), [47–51](#)

classification, [48t](#)

example of, [51t](#)

Requirements elicitation, [41–42](#)

Responsibility assignment, [2](#)

Risk, [e4–e5](#)

cause of, [e7](#)

control, [e5](#)

effects of, [e7](#)

exposure, [e6](#), [e6t](#)

identification, [e4](#)

impact, [e6](#)

planning, [e5](#), [e7](#), [e8t](#)

probability, [e6](#)

sources of, [e5–e6](#)

Risk analysis, [59–60](#), [e4–e7](#)

Risk mitigation plans, [e5](#), [e7](#)

S

Scenario building, [11](#)

Secondary actors, [31](#)

Semantic verification, [198](#)

Sequence diagrams, [93](#), [228](#)

Sequence structure, [134–135](#), [135f](#)

Smalltalk, [138](#), [139](#)

Source lines of code (SLOC), [e3](#)

Spaghetti code, [2](#)

Specialization, [139–140](#), [141f](#), [141f](#)

Stable state transition, [145](#), [145](#), [145f](#)

State machine diagram, [23–26](#), [25f](#), [26f](#), [26f](#)

Stereotyped use cases, [84–89](#)

Story points, [e2](#)

Structural model, [289–290](#)

Structural tests, [272](#)

Stubs and drivers, [274–275](#)

Subclass, [139](#)

Superclass, [139](#)

Supplementary requirements, [45](#), [47–51](#)

Surrogate key, [e69](#)

Syntactic verification, [198](#)

System actors, [30–31](#)

System command contract, [193](#)

 exceptions, [193](#)

 postconditions, [193](#)

 preconditions, [193](#)

System commands, [92](#), [97–98](#), [99f](#)

System control class, [121–122](#), [121f](#)

- System events, [79](#), [97](#), [98f](#)
- System operation contracts, [193](#)
- System operations testing, [272](#), [280–282](#), [282t](#)
- System queries, [92](#), [98](#), [99f](#), [257–259](#), [258f](#)
- System query contract, [194](#)
 - exceptions, [194](#)
 - preconditions, [194](#)
 - results, [194](#)
- System returns, [79](#), [97](#), [98f](#)
- System sequence diagrams, [92–109](#)
 - alternate flows in, [104–109](#), [105f](#), [106f](#), [107f](#)
 - complete, [109f](#)
 - connecting interface to façade-controller, [97–101](#), [100f](#), [101f](#), [101f](#), [104–109](#)
 - construction of, [95](#)
 - defined, [93](#)
 - elements of, [93–95](#), [93f](#)
 - inadequate, [94f](#)
 - main flow, [94–95](#), [102f](#), [104f](#)
 - stateful strategy, [103](#), [104f](#)
 - stateless strategy, [101–103](#), [102f](#)
 - user identification, [108f](#)
- System tests, [272](#), [282–283](#)
- System use cases, [29](#), [31–38](#), [32f](#)
 - avoiding fragments, [33f](#)
 - brief, [37](#)
 - and business use cases, [31](#), [38–41](#), [40f](#)
 - consistent result, [35–36](#), [35f](#), [36f](#)
 - essential, [36](#)
 - high-level, [29](#), [31](#)
 - interactive, [35](#)

single session, [33–34](#), [34f](#)

system boundary, [37–38](#)

uses, [29–30](#)

T

TCF (technical complexity factor), [e11–e17](#), [e12t](#), [e18t](#)

Team load capacity (TLC), [e29](#), [e29](#)

Team productivity index (TPI), [e23](#)

Technical adjustment factors, [e3](#)

Technical complexity factor (TCF), [e11–e17](#)

environmental factors, [e17–e22](#), [e19t](#)

technical factors, [e11–e17](#), [e12t](#), [e18t](#)

Technological design, [227](#)

Test-driven development, [275–276](#)

Testing, [271–272](#)

See also [Functional testing](#)

Top-level navigation, [314–316](#), [317f](#), [317f](#), [318f](#), [318f](#)

Traditional focus groups, [11](#)

Transient associations, [200–201](#), [201f](#)

U

UAW (unadjusted actor weight), [e9–e10](#)

UML (Unified Modeling Language), [3](#)

Unadjusted actor weight (UAW), [e9–e10](#)

Unadjusted use case points (UUCP), [e9](#)

Unadjusted use case weight (UUCW), [e10–e11](#)

Unidirectional associations, [e38–e50](#), [e39f](#), [e41t](#)

to 1, [e40–e44](#), [e42f](#)

advantages, [126](#)

with association class, [e49–e50](#), [e49f](#)

to many, [e44–e45](#), [e44f](#)

qualified association, [e45–e49](#), [e45f](#), [e47f](#), [e48f](#)

Unified Modeling Language (UML), [3](#)

diagrams

behavior diagrams, [3](#)

interaction diagrams, [3](#)

structure diagrams, [3](#)

Unified Process (UP), [3–5](#)

phases of, [4f](#)

construction, [5](#)

elaboration, [5](#)

inception, [4–5](#), [7](#)

transition, [5](#)

vs waterfall phases, [4](#)

principles of, [3–4](#)

Unique attributes, [121](#), [121f](#)

Unit testing, [276–280](#)

Unit tests, [271](#), [276](#)

UP (Unified Process), [3–5](#)

Use case complexity, [e29](#)

Use case diagram, [56f](#)

Use case interface modeling, [328–331](#), [329f](#), [330f](#), [330f](#), [331f](#)

Use case point analysis, [e7–e26](#)

Use case points, [10](#), [e7–e9](#), [e3](#)

Use case priority, [e29–e30](#), [e30t](#)

Use case testing, [282–284](#), [285t](#)

Use case tests, [272](#), [282](#)

Use cases

See also [Business use cases](#), [System use cases](#)

complex, [e10](#)

dependencies among, [e29](#)

detailed, counting methods, [e26](#)

expanded, *See* [Expanded use cases](#)

importance of, [61](#)

medium, [e10](#)

simple, [e10](#)

User stories, [11](#), [e2](#)

UUCP (unadjusted use case points), [e9](#), [e11](#)

UUCW (unadjusted use case weight), [e10–e11](#)

V

Viewpoints, [313](#), [314f](#), [315f](#), [315f](#)

Virtual proxy, [e88–e90](#)

data structures, [e89–e90](#)

W

Web interface patterns, [309–316](#)

WebRatio® tool, [290](#)

X

XML databases, [e67](#)



Object-Oriented Analysis and Design for Information Systems

Modeling with UML, OCL, and IFML

MK
MORGAN KAUFMANN

Raul Sidnei Wazlawick